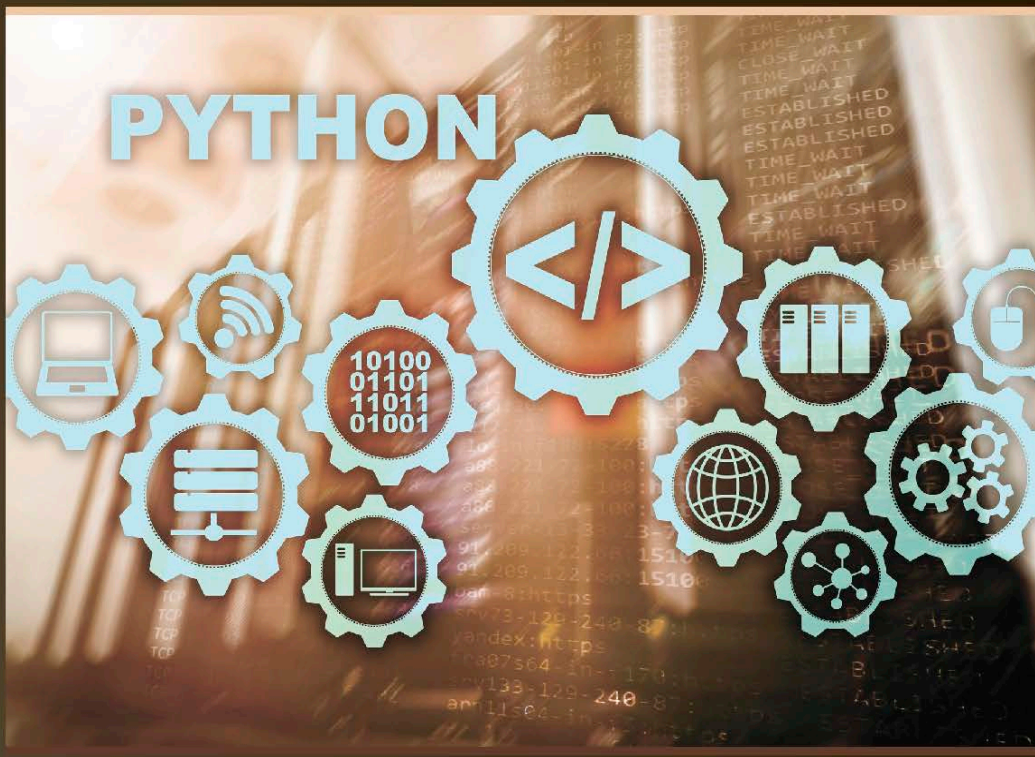
Мориц Ленц

Python

Непрерывная интеграция и доставка



ДМК ИЗДАТЕЛЬСТВО

Мориц Ленц

Python
Непрерывная интеграция и доставка

Python
Continuous Integration and Delivery

A Concise Guide with Examples

Moritz Lenz

Apress®

Python
Непрерывная интеграция и доставка

Краткое руководство с примерами

Мориц Ленц

Москва, 2020

УДК 004.438Python
ББК 32.973.22
Л33

Ленц М.
Л33 Python: Непрерывная интеграция и доставка / пер. с анг. А. Е. Мамонова, Д. А. Беликова. – М.: ДМК Пресс, 2020. – 168 с.: ил.

ISBN 978-5-97060-797-8

Язык Python используется во многих областях – веб-разработке, науке о данных и машинном обучении, интернете вещей (IoT), автоматизации систем. Морис Ленц, блогер, архитектор программного обеспечения с большим опытом работы, досконально рассматривает возможности Python, упрощающие и повышающие эффективность разработки ПО. В книге представлены различные виды тестирования; показано, как настроить автоматизированные системы, которые выполняют эти тесты, и устанавливать приложения в различных средах контролируемым способом. Представленный материал позволит разработчику успешно решать технические проблемы, которые обычно скрываются в программном коде.

Издание предназначено для технических специалистов, занимающихся доставкой программного обеспечения: разработчиков, архитекторов, инженеров по релизу и DevOps-специалистов.

УДК 004.438Python
ББК 32.973.22

ISBN 978-1-4842-4280-3 (анг.)
ISBN 978-5-97060-797-8 (рус.)

Содержание

Об авторе 10

О техническом рецензенте 11

Благодарность 12

Введение 13

Глава 1. Автоматическое тестирование 17
1.1. Что же мы хотим от тестов 17
Быстрая обратная связь 17
Уверенность 18
Помощь в отладке 19
Справка по проектированию 19
Спецификация продукта 20
1.2. Недостатки тестов 20
Усилия 20
Дополнительный код для поддержки 21
Хрупкость 21
Ложное чувство безопасности 22
1.3. Характеристики хорошего теста 22
1.4. Виды тестов 22
Модульные тесты (Unit Tests) 23
Интеграционные тесты (Integration Tests) 24
Системные тесты (System Tests) 24
Дымовые тесты (Smoke Test) 25
Тесты производительности 26
1.5. Резюме 27

Глава 2. Модульное тестирование в Python 28
2.1. Отступление: виртуальное окружение 29
2.2. Начало работы с модульными тестами 29
Первый тест 30
Пишем больше тестов 32
Тестируем неудачный случай 33
2.3. Работа с зависимостями 34
Отделение логики от внешних зависимостей 34
Внедрение зависимостей для тестирования 37
Поддельные объекты (Мок-объекты) 39

Исправление ... 41
2.4. Разделение кода и тестов ... 42
Настройка Python Path ... 42
2.5. Подробнее о модульном тестировании и Pytest ... 43
2.6. Запуск юнит-тестов в чистой среде ... 44
2.7. Другой пример проекта: matheval ... 45
Логика приложения ... 46
2.8. Резюме ... 48

Глава 3. Непрерывная интеграция с Jenkins ... 49
3.1. Серверы непрерывной интеграции ... 50
3.2. Начало работы с Jenkins ... 51
Запуск Jenkins в Docker ... 51
Настройка исходного кода репозитория ... 52
Создание первого задания Jenkins ... 53
3.3. Экспорт дополнительных сведений о тесте в Jenkins ... 56
3.4. Шаблоны для работы с Jenkins ... 57
Ответственности ... 58
Уведомления ... 58
Ветви функций и пул-запросы (pull requests) ... 59
3.5. Другие показатели в Jenkins ... 59
Покрытие кода ... 59
Сложность ... 60
Стиль кода ... 60
Проверка архитектурных ограничений ... 60
3.6. Резюме ... 61

Глава 4. Непрерывная доставка ... 62
4.1. Причины для CD и автоматизированных развертываний ... 63
Экономия времени ... 63
Сокращение цикла релиза ... 63
Сокращение цикла обратной связи ... 64
Надежность релизов ... 65
Меньшие приращения облегчают торговлю ... 65
Больше архитектурной свободы ... 66
Передовые методы обеспечения качества ... 66
4.2. План для CD ... 67
Архитектура конвейера ... 67
Антишаблон: отдельные сборки для каждой среды ... 69
Все зависит от формата упаковки ... 70
Технология управления репозиториями Debian ... 71
Инструменты для установки пакетов ... 72
Управление конвейером ... 73
4.3. Резюме ... 74

Глава 5. Сборка пакетов ... 75
5.1. Создание tar-архива с исходным кодом ... 75
5.2. Упаковка с помощью dh-virtualenv ... 76
Начало работы с упаковкой ... 77
5.3 Файл debian/control ... 77
Направление процесса сборки ... 78
Объявление зависимостей Python ... 78
Сборка пакета ... 79
Создание пакета python-matheval ... 79
Компромиссы dh-virtualenv ... 80
5.4. Резюме ... 81

Глава 6. Распространение пакетов Debian ... 82
6.1. Сигнатуры ... 82
6.2. Подготовка репозитория ... 83
6.3. Автоматизация создания репозитория и добавления пакета ... 84
6.4. Обслуживание репозиториев ... 86
Настройка компьютера для использования репозитория ... 87
6.5. Резюме ... 87

Глава 7. Развертывание пакетов ... 89
7.1. Ansible: основы ... 89
Соединения и файл инвентаризации ... 90
Модули ... 91
Модуль shell ... 92
Модуль copy ... 92
Модуль template ... 93
Модуль file ... 94
Модуль apt ... 94
Модули yum и zypper ... 95
Модуль package ... 95
Специализированные модули ... 95
Плейбуки ... 95
Переменные ... 98
Роли ... 100
7.2. Развертывание с помощью Ansible ... 102
7.3. Резюме ... 103

Глава 8. Виртуальная площадка для автоматизации развертываний ... 104
8.1. Требования и использование ресурсов ... 104
8.2. Знакомство с Vagrant ... 105
Настройка сети и Vagrant ... 106

8.3. Настройка машин ... 109
8.4. Резюме ... 114

Глава 9. Сборка в конвейере с помощью Go Continuous Delivery ... 115
9.1. О Go Continuous Delivery ... 115
Устройство конвейера ... 116
Соответствие заданий агентам ... 116
Одно слово по поводу среды ... 117
Материалы ... 118
Артефакты ... 118
9.2. Установка ... 119
Установка сервера GoCD в Debian ... 119
Установка агента GoCD в Debian ... 120
Первый контакт с XML-конфигурацией GoCD ... 121
Создание SSH-ключа ... 122
9.3. Сборка в конвейере ... 123
Макет каталога ... 124
Этапы, задания, задачи и артефакты ... 124
Конвейер в действии ... 126
Старый номер версии – это не полезно ... 126
Создание уникальных номеров версий ... 127
Еще кое-что по поводу сборки ... 128
Подключение к GoCD ... 129
9.4. Резюме ... 130

Глава 10. Распространение и развертывание пакетов в конвейере ... 131
10.1. Загрузка в конвейер ... 131
Учетные записи пользователей и безопасность ... 132
10.2. Развертывание в конвейере ... 134
10.3. Результаты ... 135
10.4. Проходя весь путь до реальных условий эксплуатации ... 136
10.5. Достижение разблокировано: базовая непрерывная доставка ... 138

Глава 11. Улучшаем конвейер ... 139
11.1. Откаты и установка определенных версий ... 139
Реализация ... 140
Давайте попробуем! ... 141
11.2. Проведение дымовых тестов в конвейере ... 142
Когда проводить такой тест? ... 142
Тестирование белого ящика ... 143
Образец дымового теста черного ящика ... 144

Добавление дымовых тестов в конвейер и роллинг-релизы 144
11.3. Шаблоны конфигурации 146
11.4. Как избежать шквала повторных сборок 148
11.5. Резюме 149

Глава 12. Безопасность 150
12.1. Опасности централизации 150
12.2. Время до выхода на рынок для исправлений безопасности 151
12.3. Аудит и спецификация ПО 152
12.4. Резюме 153

Глава 13. Управление состояниями 154
13.1. Синхронизация кода и версий базы данных 155
13.2. Разделение версий приложения и базы данных 155
Пример изменения схемы 156
Создание нового столбца, допускающего значение NULL 157
Миграция данных 159
Применение ограничений и очистка 159
Предварительные условия 160
Инструментарий 161
Структура 161
Единого решения не существует 162
13.3. Резюме 162

Глава 14. Выводы и перспективы 163
14.1. Что дальше? 163
Улучшенное обеспечение качества 163
Метрики 164
Автоматизация инфраструктуры 165
14.2. Заключение 167

Об авторе

Мориц Ленц (Moritz Lenz) – плодовитый блогер, автор и участник проектов с открытым исходным кодом. Он работает архитектором программного обеспечения и главным инженером-программистом для ИТ среднего бизнеса аутсорсинговой компании, где создал систему непрерывной интеграции и доставки для более пятидесяти программных библиотек и приложений.

О техническом рецензенте

Майкл Томас (Michael Thomas) более 20 лет трудился в области разработки программного обеспечения в качестве индивидуального участника, руководителя группы, менеджера программ и вице-президента по проектированию. Майкл имеет более 10 лет опыта работы с мобильными устройствами. В настоящее время он работает в медицинском секторе, используя мобильные устройства для ускорения обмена информацией между пациентами и поставщиками медицинских услуг.

Благодарность

Написание книги не является одиночным начинанием и возможно только с помощью многих людей и организаций. Я хотел бы поблагодарить всех моих бета-читателей (beta reader), которые предоставили обратную связь. К ним относятся, в произвольном порядке, Карл Фогель (Karl Vogel), Михаил Иткин (Mikhail Itkin), Карл Мясак (Carl Mäsak), Мартин Турн (Martin Thurn), Шломи Фиш (Shlomi Fish), Ричард Липпманн (Richard Lippmann), Ричард Фоли (Richard Foley) и Роман Филиппов (Roman Filippov). Пол Кокрейн (Paul Cochrane) заслуживает особой благодарности за рецензирование и предоставление обратной связи по сообщениям в блоге и рукописи, а также за доступность для обсуждения контента, идей и организационных вопросов.

Я также хочу поблагодарить мою издательскую команду в Apress: Стив Англин (Steve Anglin), Марк Пауэрс (Mark Powers) и Мэттью Муди (Matthew Moodie), – а также всех, кто делает потрясающую работу на заднем плане, например дизайн обложки, набор текста и маркетинг.

Наконец, спасибо моим родителям за то, что они разожгли мою любовь к книгам и технике. И самое главное, моей семье: Сигне (Signe) – моей жене, за постоянную поддержку; и моим дочерям – Иде (Ida) и Ронье (Ronja), за то, что они поддерживали меня в реальном мире и приносили радость в мою жизнь.

Введение

Одним из ключей к успешной разработке программного обеспечения является получение быстрой обратной связи. Это помогает разработчикам избежать тупиков, и в случае ошибки, которая быстро обнаруживается, ее можно исправить, пока код еще свеж в памяти разработчика.

С точки зрения бизнеса быстрая обратная связь также помогает заинтересованным сторонам и продуктовому менеджеру не создавать функциональность, которая оказывается бесполезной, что позволяет избежать напрасных усилий. Достижение взаимопонимания о желаемом продукте является очень сложной задачей в любом программном проекте. Показ работающего (хотя и частично) продукта на ранней стадии часто помогает устранить недопонимание между заинтересованными сторонами и разработчиками.

Существует множество способов добавления обратной связи на разных уровнях, от добавления компоновки (linting) и других проверок кода в IDE до гибких процессов (agile processes), которые подчеркивают повышенную стоимость доставки. Первая часть этой книги посвящена тестам программного обеспечения и их автоматическому выполнению, практике, известной как *непрерывная интеграция* (Continuous Integration – CI).

При реализации CI вы настраиваете сервер, который автоматически проверяет каждое изменение исходного кода, возможно, в нескольких средах, например в комбинациях версий операционной системы и языка программирования.

Следующим логическим шагом и темой второй части этой книги является *непрерывная доставка* (Continuous Delivery – CD). После создания и тестирования кода вы добавляете больше шагов к автоматизированному процессу: автоматическое развертывание в одной или нескольких тестовых средах, дополнительные тесты в установленном состоянии и, наконец, развертывание в производственной среде. Последний шаг обычно охраняется воротами ручного одобрения.

CD расширяет автоматизацию и таким образом предоставляет возможность быстрых итерационных циклов вплоть до производственной среды, где программное обеспечение может принести

пользу. С помощью такого механизма вы можете быстро получить обратную связь или данные об использовании от реальных клиентов и оценить, полезно ли расширять функциональность или обнаруживать баги, пока разработчики все еще помнят код, который они написали.

Примеры кода в этой книге используют Python. Благодаря своей динамической природе Python хорошо подходит для небольших экспериментов и быстрой обратной связи. Хорошо укомплектованная стандартная библиотека и обширная экосистема доступных библиотек и фреймворков, а также чистый синтаксис Python делают его хорошим выбором даже для более крупных приложений. Python обычно используется во многих областях, например в веб-разработке, науке о данных и машинном обучении, интернете вещей (IoT) и автоматизации систем. Это становится лингва франка профессиональных программистов и тех, кто просто коснулся автоматизировать какую-то часть своей работы или хобби.

Python поставляется в двух основных языковых версиях, 2 и 3. Поскольку поддержку Python 2 планируется завершить в 2020 году, и почти все основные библиотеки теперь поддерживают Python 3, новые проекты следует начинать в Python 3, а устаревшие приложения нужно перенести на эту языковую версию, если это возможно. Следовательно, в этой книге предполагается, что Python относится к Python 3, если явно не указано иное. Если вы знаете только Python 2, будьте уверены, что вы легко поймете исходный код, содержащийся в этой книге, и с легкостью перенесете знания в Python 3.

Целевая аудитория

Эта книга предназначена для технических специалистов, вовлеченных в процесс доставки программного обеспечения: разработчиков программного обеспечения, архитекторов, инженеров по релизу и инженеров DevOps.

Главы, в которых используются примеры исходного кода, предполагают базовое знакомство с языком программирования Python. Если вы знакомы с другими языками программирования, потратьте несколько часов на чтение вводного материала по Python. Вы, вероятно, достигнете уровня, на котором сможете легко следовать примерам кода в книге.

В примере инфраструктуры используется Debian GNU/Linux, поэтому знакомство с этой операционной системой полезно, хотя и не обязательно.

Примеры кода

Примеры кода, используемые в этой книге, доступны на GitHub под организацией `python-ci-cd` по адресу https://github.com/python-ci-cd или по ссылке **Download Source Code**, расположенной по адресу www.apress.com/9781484242803.

Поскольку некоторые примеры основаны на автоматическом извлечении кода из определенных репозиториев Git, необходимо разделить их на несколько репозиториев. Несколько глав ссылаются на отдельные репозитории в этом пространстве имен.


Так будут оформляться предупреждения и важные примечания.


Так будут оформляться советы или рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Автоматическое тестирование

Перед погружением в примеры тестирования кода Python необходимо более подробно обсудить саму природу тестов.

1.1. Что же мы хотим от тестов

Зачем вообще заморачиваться над написанием тестов? Существует множество причин, почему мы хотим написать или хотя бы иметь тесты.

Это не необычно иметь несколько тестов в одном наборе тестов (test suite), написанных в ответ различным потребностям.

Быстрая обратная связь

Каждое изменение в коде содержит риск внести баги. Исследования показывают, что от 7 до 20 % исправлений багов привносят новые баги[1].

Не лучше ли, если бы мы могли найти больше багов перед тем, как они найдут путь к клиенту? Или даже до того, как ваши коллеги увидят их? Это не просто вопрос тщеславия. Если вы быстро получаете обратную связь о том, что у вас баг, то проще вспомнить все детали той части кода, над которой только что работали, так что исправление багов будет в разы быстрее.

---

[1] *Jim Bird*. Bugs and Numbers: How Many Bugs Do You Have in Your Code? // Building Real Software: Developing and Maintaining Secure and Reliable Software in the Real World. URL: http://swreflections.blogspot.de/2011/08/bugs-and-numbers-how-many-bugs-do-you.html. August 23, 2011.

Много тест-кейсов (test case), чтобы предоставить цикл быстрых обратных связей. Вы можете запустить их до того, как отправить свои изменения в систему контроля версий, и они сделают вашу работу более эффективной и будут держать историю вашего кода чистой.

Уверенность

Ссылаясь на предыдущий абзац, следует упомянуть отдельно, можете быть уверены, зная, что простые ошибки за вас будут отловлены набором тестов. В большинстве компаний, ориентированных на разработку программного обеспечения, существуют критические зоны, в которых баги могут подвергнуть опасности весь бизнес. Только представьте, что разработчик случайно ошибся в системе входа (login system) в программе управления данными о здоровье и теперь клиенты видят чужие диагнозы. Или система автоматической выплаты перечислила на клиентскую кредитную карту неправильную сумму.

Даже компании, не разрабатывающие программное обеспечение, могут катастрофически пострадать от ошибок в программах. Оба – климатический орбитальный аппарат Mars[1] и первый запуск ракеты «Ariane 5»[2] – понесли потери соответствующего транспортного средства из-за проблем с программным обеспечением.

Ответственность их работы создает эмоциональный стресс для разработчиков программного обеспечения. Автоматизированные тесты (automated tests) и хорошая методология разработки могут помочь уменьшить этот стресс.

Даже если разрабатываемое программное обеспечение не является критически важным, неблагоприятные факторы риска могут привести к тому, что разработчики или сопровождающие лица внесут наименьшие возможные изменения и отложат необходимый рефакторинг (refactoring), который сохранит код в рабочем состоянии. Уверенность, которую обеспечивает хороший набор тестов, может позволить разработчикам сделать то, что необходимо, чтобы кодовая база не стала пресловутым большим комком грязи[3].

---

[1] Wikipedia. Mars Climate Orbiter // https://en.wikipedia.org/wiki/Mars_Climate_Orbiter. 2018.

[2] *J. L. Lions*. Ariane 5: Flight 501 Failure. Report by the Inquiry Board // http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html. July 1996.

[3] Wikipedia. Большой комок грязи // https://ru.wikipedia.org/wiki/Большой_комок_грязи. 2018.

Помощь в отладке

Когда разработчики изменяют код, что, в свою очередь, приводит к сбою теста, они хотят, чтобы тест помог найти ошибку. Если тест просто говорит «что-то не так», это знание лучше, чем незнание ошибки. Было бы еще полезнее, если бы тест мог дать подсказку для начала отладки.

Если, например, сбой теста показывает, что функция `find_Shorttest_path` вызвала исключение, вместо того чтобы возвращать путь, как ожидалось, и мы знаем, что или эта функция (либо та, которую она вызвала) сломалась, или она получила неправильный ввод. Это гораздо лучшее средство отладки.

Справка по проектированию

Движение Extreme Programming (XP)[1] отстаивает необходимость практической *разработки на основе тестов* (test-driven development – TDD). То есть прежде чем писать какой-либо код, который решает проблему, вы сначала пишете неудачный тест. Затем пишете достаточно кода, чтобы пройти тест. Либо вы закончили, либо пишете следующий тест. Промыть и повторить.

Это имеет очевидные преимущества: вы убедитесь, что весь код, который пишется, имеет тестовое покрытие, и что вы не пишете ненужный или недостижимый код. Тем не менее специалисты по TDD также сообщили, что подход, основанный на тестировании, помог им написать лучший код. Один из аспектов заключается в том, что написание теста заставляет задуматься о программном интерфейсе приложения (Application programming interface – API), который будет иметь реализацию, и поэтому вы начнете реализацию с лучшим представлением. Другая причина заключается в том, что чистые (pure) функции (функции, возвращаемое значение которых зависит только от ввода, они не вызывают побочных эффектов и не считывают данные из баз данных и т. д.) очень просто проверить. Таким образом, подход, основанный на тестировании, направляет разработчика к лучшему разделению алгоритмов или бизнес-логики от поддержки логики и вспомогательной. Такое разделение интересов является аспектом хорошего проектирования программного обеспечения.

---

[1] Wikipedia. Extreme programming // https://en.wikipedia.org/wiki/Extreme_programming. 2018.

Следует отметить, не все согласны с этими наблюдениями, учитывая опыт или аргументы о том, что тестировать некоторый код гораздо сложнее, чем писать, это приводит к трате усилий, требуя тестов для всего. Тем не менее помощь в разработке, которую могут предоставить тесты, является причиной, по которой разработчики пишут код, и поэтому не должна быть здесь упущена.

Спецификация продукта

Дни больших унифицированных технических документов для программных проектов в основном прошли. Большинство проектов следуют некоторой итеративной модели разработки, и даже если есть подробный документ спецификации, он часто устарел.

Когда нет подробной и актуальной спецификации, тестовый набор может взять на себя роль спецификации. Когда люди не уверены, как программа должна вести себя в определенной ситуации, тест может дать ответ. Для языков программирования, форматов данных, протоколов и прочего может даже иметь смысл предложить набор тестов, который можно использовать для проверки более чем одной реализации.

1.2. Недостатки тестов

Было бы нечестно молчать о недостатках тестов. Эти недостатки не должны отвлекать вас от написания тестов, но знание о них поможет решить: что тестировать, как писать тесты и, возможно, сколько тестов писать.

Усилия

Написание тестов требует времени и усилий. Таким образом, когда вам поручено реализовать функцию, нужно не только реализовать эту функцию, но и написать тесты для нее, что приведет к большей работе и меньшему количеству времени для выполнения других задач, которые могут принести прямую пользу бизнесу. Если, конечно, тесты не обеспечивают достаточной экономии времени (например, за счет того, что нет необходимости исправлять ошибки в производственной среде и очищать данные, которые были повреждены в результате ошибки), чтобы амортизировать время, затрачиваемое на написание тестов.

Дополнительный код для поддержки

Тесты сами по себе являются кодом и должны поддерживаться, как и код, который тестируется. В общем, вы хотите наименьшее количество кода, которое может решить вашу проблему, потому что чем меньше у вас кода, тем меньше кода нужно поддерживать. Думайте о коде (включая тестовый код) как об обязательстве, а не как об активе.

Если вы пишете тесты вместе с вашими функциями и исправлениями ошибок, то должны изменить эти тесты при изменении требований. Некоторые тесты также требуют изменения при рефакторинге, что затрудняет изменение базы кода.

Хрупкость

Некоторые тесты могут быть хрупкими, то есть они иногда дают неправильный результат. Тест, который не проходит, даже если рассматриваемый код является правильным, называется *ложноположительным*. Такой сбой теста требует времени для отладки, не предоставляя никакой ценности. *Ложноотрицательный тест* – это тест, который не дает сбоя, когда тестируемый код не работает. Ложноотрицательный тест также не имеет никакой ценности, но его гораздо труднее обнаружить, чем ложноположительный тест, поскольку большинство инструментов привлекают внимание к неудачным тестам.

Хрупкие тесты подрывают доверие к тестам. Если развертывание продукта с ошибочными тестами становится нормой – поскольку все считают, что эти неудачные тесты являются ложноположительными,– то значение результатов набора тестов упало до нуля. Вы можете по-прежнему использовать его для отслеживания информации, какой из тестов не прошел по сравнению с последним прогоном, но это приводит к большой ручной работе, которую никто не хочет делать.

К сожалению, некоторые виды тестов очень сложно выполнить надежно. Тесты графического пользовательского интерфейса (Graphical User Interface – GUI), как правило, довольно чувствительны к изменениям макета или технологии. Тесты, в которых используются компоненты, находящиеся вне вашего контроля, также могут быть источником хрупкости.

Ложное чувство безопасности

Безупречный запуск набора тестов может дать вам ложное чувство безопасности. Это может быть связано или с ложноотрицательными результатами (тесты, которые должны были провалиться, но не провалились), или с отсутствием тестовых сценариев. Даже если тестовый набор достигает 100%-го покрытия тестируемого кода, он может пропустить некоторые пути кода (code path) либо сценарии. Таким образом, вы видите успешно прошедший тестовый прогон и воспринимаете это как указание на то, что ваше программное обеспечение работает правильно только для того, чтобы быть заваленным отчетами об ошибках, после того как реальные клиенты вступают в контакт с продуктом.

Не существует прямого решения для чрезмерной уверенности, которую может обеспечить набор тестов. Только благодаря опыту работы с кодовой базой и ее тестами вы почувствуете реалистичные уровни достоверности, которые обеспечивает зеленый (т. е. проходящий) тестовый прогон.

1.3. Характеристики хорошего теста

Хороший тест – это тест, который сочетает в себе несколько причин написания тестов, избегая при этом как можно больше недостатков. Это означает, что тест должен быть быстрым, простым для понимания и поддержки, давать хорошую и конкретную обратную связь в случае неудачи и быть надежным.

Может быть, несколько удивительно, но иногда он должен давать сбои, хотя можно ожидать, что тест не пройден. Тест, который никогда не проходит, также никогда не дает обратной связи и не может помочь с отладкой. Это не означает, что вы должны удалить тест, для которого вы никогда не регистрировали сбой. Возможно, это не удалось на компьютере разработчика, и он или она исправили ошибку перед проверкой изменений.

Не все тесты могут соответствовать всем критериям хороших тестов, поэтому давайте рассмотрим некоторые из различных типов тестов и компромиссы, которые им присущи.

1.4. Виды тестов

Существует традиционная модель классификации тестов, основанная на их области действия (объем кода, который они охваты-

вают) и их назначении. Эта модель разделяет код тестов на модульные, интеграционные и системные тесты. Он также добавляет дымовое тестирование (smoke tests), тесты производительности (performance tests) и другие тесты для различных целей.

Модульные тесты (Unit Tests)

Модульный тест (выполняется изолированно) – наименьший блок программы, которую имеет смысл охватить. В процедурном или функциональном программировании язык, который имеет тенденцию быть подпрограммой или функцией. В объектно ориентированном языке, таком как Python, это может быть метод. В зависимости от того, насколько строго вы интерпретируете определение, это также может быть класс или модуль.

Модульный тест должен избегать запуска кода за пределами тестируемого модуля. Поэтому, если вы тестируете бизнес-приложение, интенсивно использующее базу данных, модульный тест по-прежнему не должен выполнять вызовы базы данных (доступ к сети для вызовов API) или файловой системы. Существуют способы замены таких внешних зависимостей для целей тестирования, о которых я расскажу позже, хотя если вы сможете структурировать свой код, чтобы избежать таких вызовов, по крайней мере в большинстве модулей, тем лучше.

Поскольку доступ к внешним зависимостям делает большую часть кода медленным, модульные тесты обычно бывают невероятно быстрыми. Это делает их подходящими для тестирования алгоритмов или основной бизнес-логики.

Например, если ваше приложение является помощником по навигации, в нем есть хотя бы один фрагмент алгоритмически сложного кода: маршрутизатор, который (учитывая карту, начальную точку и цель) создает маршрут или список возможных маршрутов, с такими показателями, как длина и ожидаемое время прибытия. Этот маршрутизатор или даже его части – это то, что вы хотите охватить модульными тестами как можно тщательнее, включая странные крайние случаи, которые могут вызвать бесконечные циклы, или проверить, что путешествие из Берлина в Мюнхен не отправляет вас через Рим.

Огромный объем тестовых случаев, которые вы хотите для такого устройства, делает другие виды тестов нецелесообразными. Кроме того, вы не хотите, чтобы такие тесты проваливались из-

за несвязанных компонентов, поэтому их сосредоточенность на устройстве повышает специфичность.

Интеграционные тесты (Integration Tests)

Если вы собрали сложную систему, такую как автомобиль или космический корабль, из отдельных компонентов, и каждый компонент работает нормально в отдельности, каковы шансы, что все это работает? Существует множество причин, по которым что-то может пойти не так: некоторые провода могут быть неисправны, компоненты хотят общаться по несовместимым протоколам, или, возможно, соединения не могут противостоять вибрации во время работы.

По программному обеспечению ничего не отличается, поэтому пишут интеграционные тесты. Интеграционный тест проверяет сразу несколько блоков. Это делает несоответствия на границах между блоками очевидными (через сбои теста), позволяя исправлять такие ошибки на ранней стадии.

Системные тесты (System Tests)

Системный тест помещает часть программного обеспечения в среду и проверяет ее там. Для классической трехуровневой архитектуры системный тест начинается с ввода через пользовательский интерфейс и тестирует все уровни вплоть до базы данных.

Если модульные тесты и интеграционные тесты являются тестами белого ящика (тесты, которые требуют и используют знания о том, как реализовано программное обеспечение), системные тесты, как правило, являются тестами черного ящика. Они принимают точку зрения пользователя и не заботятся о внутренностях системы.

Это делает системные тесты наиболее реалистичными с точки зрения того, как программное обеспечение подвергается тестированию, но они имеют несколько недостатков.

Во-первых, управление зависимостями для системных тестов может быть очень сложным. Например, если вы тестируете веб-приложение, обычно сначала требуется учетная запись, использующаяся для входа в систему, а затем для каждого тестового примера требуется фиксированный набор данных, с которыми он может работать.

Во-вторых, системные тесты часто используют столько компонентов одновременно, что сбой теста не дает четкого представле-

ния о том, в каком месте на самом деле неправильно, и требует, чтобы разработчик смотрел на каждый сбой теста, нередко для выяснения, что изменения не связаны с ошибками теста.

В-третьих, системные тесты обнаруживают сбои в компонентах, которые вы не собирались тестировать. Системный тест может завершиться неудачей из-за неправильно настроенного сертификата безопасности транспортного уровня (Transport Layer Security – TLS) в API, который используется программным обеспечением и который может быть полностью вне вашего контроля.

Наконец, системные тесты обычно намного медленнее, чем модульные и интеграционные тесты. Тесты белого ящика позволяют вам тестировать только те компоненты, которые вам нужны, поэтому вы можете избежать запуска неинтересного кода. В тесте системы для веб-приложения вам, возможно, придется выполнить вход в систему, перейти на страницу, ту, что хотите проверить, ввести некоторые данные, а затем, наконец, выполнить тест, который вы действительно хотите сделать. Системные тесты часто требуют гораздо больше настроек, чем модульные или интеграционные, увеличивая время их выполнения и более длительный интервал до получения обратной связи о коде.

Дымовые тесты (Smoke Test)

Дымовой тест похож на системный в том, что он тестирует каждый слой в вашем технологическом стеке, хотя это не является тщательным тестом для каждого. Обычно он написан не для проверки правильности какой-либо части вашего приложения, а скорее для того, чтобы приложение вообще работало в его текущем контексте.

Дымовой тест для веб-приложения может быть такой же простой, как вход в систему, после чего следует вызов страницы профиля пользователя, чтобы убедиться, что имя пользователя появляется где-то на этой странице. Это не проверяет какую-либо логику, но обнаруживает такие вещи, как неправильно настроенный веб-сервер или сервер базы данных, неверные файлы конфигурации или учетные данные.

Чтобы получить больше пользы от дымового теста, вы можете добавить страницу состояния или конечную точку API в свое приложение, которое выполняет дополнительные проверки, такие как наличие всех необходимых таблиц в базе данных, наличие зависимых служб и т. д. Только если все эти зависимости во время выпол-

нения будут реализованы, будет статус **OK**, который может легко определить дымовой тест. Обычно вы пишете только один или два дымовых теста для каждого развертываемого компонента, но запускаете их для каждого развертываемого экземпляра.

Тесты производительности

Обсуждаемые до сих пор тесты фокусируются на корректности, но нефункциональные качества, такие как производительность и безопасность, могут быть одинаково важны. В принципе, довольно просто запустить тест производительности: записать текущее время, выполнить определенное действие, снова записать текущее время. Разница между двумя временными записями заключается во времени выполнения этого действия. При необходимости повторите и рассчитайте некоторую статистику (например: медиана, среднее значение, стандартное отклонение) из этих значений.

Как обычно, дьявол кроется в деталях. Основными проблемами являются создание реалистичной и надежной тестовой среды, реалистичных тестовых данных и реалистичных тестовых сценариев.

Многие бизнес-приложения сильно зависят от баз данных. Итак, ваша среда тестирования производительности также требует базы данных. Репликация большого производственного экземпляра базы данных для среды тестирования может быть довольно дорогой, как с точки зрения аппаратного обеспечения, так и с точки зрения затрат на лицензирование. Таким образом, существует соблазн использовать уменьшенную базу данных тестирования, что может привести к аннулированию результатов. Если что-то идет медленно в тестах производительности, разработчики, как правило, говорят: «Это просто слабая база данных; Прод легко справится» (Прод – Production server, на котором будет произведен релиз. – *Прим. перев.*), и они могут быть правы. Или нет. Нет способа узнать.

Другим коварным аспектом настройки среды является множество движущихся частей, когда речь заходит о производительности. На виртуальной машине (ВМ) вы обычно не знаете, сколько циклов ЦП получила виртуальная машина от гипервизора, или среда виртуализации играла забавные трюки с памятью виртуальной машины (например, выгружая часть памяти виртуальной машины на диск), вызывая непредсказуемую производительность.

На физических машинах (которые также лежат в основе каждой виртуальной машины) вы сталкиваетесь с современными системами управления питанием, которые контролируют тактовую ча-

стоту, основываясь на тепловых соображениях, а в некоторых случаях даже на специальных инструкциях, используемых в ЦП[1].

Все эти факторы приводят к тому, что измерения производительности становятся гораздо более неопределенными, чем можно наивно ожидать от такой детерминированной системы, как компьютер.

1.5. Резюме

Как разработчики программного обеспечения мы хотим, чтобы автоматизированные тесты давали быструю обратную связь об изменениях, улавливали регрессии до того, как они достигли клиента, и давали достаточную уверенность в том, что мы сможем изменить код. Хороший тест – быстрый, надежный и имеет высокую диагностическую ценность в случае неудачи.

Модульные тесты обычно бывают быстрыми и имеют высокую диагностическую ценность, но охватывают только небольшие фрагменты кода. Чем больше кода охватывает тест, тем медленнее и хрупче он становится, и его диагностическая ценность уменьшается.

В следующей главе мы рассмотрим, как писать и запускать модульные тесты на Python. Затем разберем, как запускать их автоматически для каждого коммита.

---

[1] *Vlad Krasnov*. On the Dangers of Intel's Frequency Scaling // Cloudflare. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/. November 10, 2017.

Глава 2

Модульное тестирование в Python

Многие программисты вручную проверяют код, который они пишут, вызывая разрабатываемый код, выводят результат в консоль и визуально сканируют вывод на предмет корректности. Это работает для простых задач, но страдает от некоторых проблем:

- когда вывод увеличивается, становится все труднее обнаружить ошибки;
- когда программист устает, легко пропустить неверный вывод;
- когда реализованная функция становится больше, можно пропустить регрессии в частях, которые были «протестированы» ранее;
- поскольку неформальные тестовые сценарии, как правило, полезны только для программиста, который их написал, полезность теряется для других разработчиков.

Таким образом, было изобретено *модульное тестирование*, в котором каждый записывает примеры вызовов в части кода и сравнивает возвращаемое значение с ожидаемым значением.

Это сравнение обычно выполняется таким образом, что при прохождении теста вывода практически нет, а в противном случае – очень понятный вывод. Тестовую программу можно использовать для запуска тестов из нескольких тестовых сценариев для отчета об ошибках и статистической сводки пройденных тестов.

2.1. Отступление: виртуальное окружение

Для запуска юнит-тестов, которые мы собираемся написать, потребуются некоторые дополнительные инструменты, доступные в виде пакетов Python. Чтобы установить их, вы должны использовать инструмент под названием *virtualenv*. Это каталог Python, содержащий интерпретатор Python, программы управления пакетами, такие как `pip`, а также символические ссылки на базовые пакеты Python, что дает нетронутую среду Python, на которой можно создать настраиваемую изолированную виртуальную среду, содержащую именно нужные вам библиотеки. Virtualenv позволяет вам установить любой пакет Python, который хотите; вам не нужны привилегии администратора для установки зависимостей для вашего приложения. Можно активировать один virtualenv в рамках сессии и просто удалить директорию с ним, когда он больше не нужен.

Virtualenvs используется для изоляции отдельных сред разработки друг от друга и от Python, установленного системой. Чтобы создать его, вам нужен инструмент `virtualenv`, который обычно поставляется с установкой Python или – в дистрибутивах Linux – может быть установлен через менеджер пакетов. В системах на основе Debian вы можете установить его так:

```
$ sudo apt-get install virtualenv
```

Чтобы создать virtualenv с названием `venv`, выполните:

```
$ virtualenv -p python3 venv
```

Это подготовит директорию под названием `venv` с необходимыми файлами. Ваш следующий шаг должен быть активирован таким образом:

```
$ source venv/bin/activate
```

После того как активировали виртуальное окружение, вы можете установить в него пакеты, используя `pip`, например:

```
$ pip install pytest
```

Когда закончите, отключите его с помощью команды `deactivate`.

2.2. Начало работы с модульными тестами

Чтобы проиллюстрировать модульное тестирование, давайте начнем с одной функции и того, как ее тестировать. Функция, кото-

рую я хочу реализовать здесь, – это *бинарный поиск*. Получив отсортированный список номеров (назовем его стогом сена), найдите в нем другой номер (иглу). Если он присутствует, верните индекс, по которому он был найден. Если нет, создайте исключение типа `ValueError`. Вы можете найти код и тесты для этого примера по адресу https://github.com/python-ci-cd/binary-search.

Начнем со среднего элемента стога сена. Если он окажется равным игле, мы закончили. Если меньше иглы, можем повторить поиск в левой половине стога сена. Если он больше, мы продолжаем поиск в правой половине стога сена.

Чтобы отслеживать область внутри стога сена, которую мы должны искать, сохраняем два индекса – `left` и `right` – и в каждой итерации перемещаем один из них ближе к другому, сокращая пространство для поиска пополам на каждом шаге.

Вот как выглядит первая попытка реализации этой функции:

```
def search(needle, haystack):
    left = 0
    right = len(haystack) - 1

    while left <= right:
        middle = left + (right - left) // 2
        middle_element = haystack[middle]
        if middle_element == needle:
            return  middle
        elif middle_element < needle:
            left = middle
        else:
            right = middle
    raise ValueError("Value not in haystack")
```

Первый тест

Это работает? Кто знает? Давайте узнаем, написав тест.

```
def test_search():
    assert search(2, [1, 2, 3, 4]) == 1, \
        'found needle somewhere in the haystack'
```

Это простая функция, которая выполняет функцию поиска с примерами входных данных и использует `assert` для создания исключения, если ожидание не было выполнено. Вместо непосредственного вызова этой тестовой функции мы используем `pytest`, инструмент командной строки, предоставляемый пакетом Python с тем же именем. Если он недоступен в вашей среде разработки,

можете установить его с помощью следующей команды (не забудьте запустить ее внутри virtualenv):

```
pip install pytest
```

Когда pytest доступен, можете запустить его в файле, содержащем как функцию поиска, так и функцию тестирования, следующим образом:

```
$ pytest binary-search.py
==================== test session starts ====================
platform linux -- Python 3.5.2, pytest-3.3.2, py-1.5.2
rootdir: /home/moritz/examples, inifile:
collected 1 item
binary-search.py .                                      [100%]
================== 1 passed in 0.01 seconds ==================
```

В ходе тестового прогона выводится различная информация: она включает сведения о платформе и версии используемого программного обеспечения, рабочей директории и используемом файле конфигурации pytest (в данном примере – ни одного).

В строке collected показано одно значение, затем видно, что pytest нашел одну тестовую функцию. Точка за именем файла в следующей строке показывает прогресс, с одной точкой для каждого выполненного теста.

В терминале последняя строка отображается зеленым цветом, чтобы указать пройденный тестовый прогон. Если бы мы допустили ошибку, скажем, использовали 0 вместо 1, то в качестве ожидаемого результата получили бы некоторые диагностические данные, например:

```
========================= FAILURES ==========================
_____________________test_search_____________________________
     def test_search():
>        assert search(2, [1, 2, 3, 4]) == 0, \
             'found needle somewhere in the haystack'
E        AssertionError: found needle somewhere in the haystack
E        assert 1 == 0
E         + where 1 = search(2, [1, 2, 3, 4])
binary-search.py:17: AssertionError
================== 1 failed in 0.03 seconds ==================
```

Из примера видно, что тестовая функция не работает как в виде исходного кода, так и со значениями, подставленными в обе стороны от оператора == в вызове assert, показывая, что именно пошло не так. В терминале с поддержкой цвета неудачный тест и строка

состояния внизу будут показаны красным, чтобы сделать неудачные тесты очевидными.

Пишем больше тестов

Многие ошибки в коде проявляются в крайних случаях: с пустыми списками или строками в качестве входных данных, числами, равными нулю, доступом к первому и последнему элементам списков и т. д. Рекомендуется думать об этих случаях при написании тестов и освещать их. Начнем с поиска первого и последнего элементов.

```
def test_search_first_element():
     assert search(1, [1, 2, 3, 4]) == 0, \
         'search first element'
def test_search_last_element():
     assert search(4, [1, 2, 3, 4]) == 3, \
       'search last element'
```

Тест на поиск первого элемента проходит, но тест на последний элемент зависает, то есть выполняется бесконечно, без завершения. Вы можете прервать процесс Python, одновременно нажав клавиши **Ctrl** и **C**.

Если функция поиска может найти первый, но не последний элемент, в нем должна быть какая-то асимметрия. Действительно есть: для определения среднего элемента используется оператор целочисленного деления `//`, который округляет положительные числа до нуля. Например, `1 // 2 == 0`. Это объясняет, почему цикл может застрять: когда `right` равно `left + 1`, код устанавливает середину в значение `left`. Если выполняется ветвь `left = middle`, площадь стога сена, в которой ищет функция, не уменьшается в размере, и цикл застревает.

Это легко исправить. Поскольку код уже определил, что элемент с индексом `middle` не является иглой, его можно исключить из поиска.

```
def search(needle, haystack):
    left = 0
    right = len(haystack) - 1

    while left <= right:
        middle = left + (right - left) // 2
        middle_element = haystack[middle]
        if middle_element  == needle:
            return middle
        elif middle_element < needle:
```

```
            left = middle + 1
        else:
            right = middle - 1
    raise ValueError("Value not in haystack")
```

С данным исправлением все три теста пройдут.

Тестируем неудачный случай

До сих пор тесты были сосредоточены на «удачном случае», случае, в котором был найден элемент и не было обнаружено ошибок. Поскольку исключения не являются исключением (извините за каламбур) в нормальном потоке управления, они также должны быть проверены.

В `pytest` есть некоторые инструменты, которые помогут вам убедиться, что исключение вызвано фрагментом кода и имеет правильный тип.

```
def test_exception_not_found():
    from pytest import raises

    with raises(ValueError):
        search(-1, [1, 2, 3, 4])

    with raises(ValueError):
        search(5, [1, 2, 3, 4])
    with raises(ValueError):
        search(2, [1, 3, 4])
```

Здесь мы тестируем три сценария: значение было меньше, чем первый элемент в стоге сена; больше, чем последний; и наконец, значение находится между первым и последним размерами элемента, но просто не в стоге сена.

`pytest.raises` возвращает *менеджер контекста* (*context manager*). Менеджеры контекста, помимо прочего, представляют собой удобный способ обернуть код (внутри блока `with`) в некоторый другой код. В этом случае менеджер контекста ловит исключение из блока `with`, и тест проходит, если он имеет правильный тип. И наоборот, тест завершается неудачей, если либо не было сгенерировано исключение, либо произошел сбой неправильного типа, например `KeyError`.

Как и в случае с оператором `assert`, можете указывать метки тестов. Они полезны как для отладки сбоев тестов, так и для документирования тестов. С помощью функции `raises` вы можете передать метку теста в качестве именованного аргумента с именем `message`.

```
def test_exception_not_found():
    from pytest import raises
    with raises(ValueError, message= "left out of bounds"):
        search(-1, [1,2,3,4])
    with raises(ValueError, message= "right out of bounds"):
        search(5, [1,2,3,4])
    with raises(ValueError, message= "right out of bounds"):
        search(2, [1,3,4])
```

2.3. Работа с зависимостями

Не весь код так же прост для тестирования, как функция поиска из предыдущих разделов. Некоторые функции используют внешние библиотеки или взаимодействуют с базами данных, API или интернетом.

В юнит-тестах вы должны избегать выполнения этих внешних действий по нескольким причинам.

- Действия могут иметь нежелательные побочные эффекты, такие как отправка электронных писем клиентам или коллегам, запутывая или даже причиняя им вред.
- Обычно у вас нет контроля над внешними службами. Следовательно, вы не можете контролировать последовательные ответы, что значительно затрудняет написание надежных тестов.
- Выполнение внешних действий – таких как запись или удаление файлов – оставляет среду в измененном состоянии, что может привести к результатам теста, которые невозможно воспроизвести.
- Производительность страдает, что негативно влияет на цикл обратной связи с разработчиками.
- Часто внешним службам – таким как базы данных или API – требуются учетные данные, которые затрудняют управление и создают серьезный барьер для настройки среды разработки и запуска тестов.

Как же тогда избежать этих внешних зависимостей в своих юнит-тестах? Давайте рассмотрим некоторые варианты.

Отделение логики от внешних зависимостей

Многие приложения получают данные откуда-то, часто из разных источников, затем выполняют некоторую логику с ними и, возможно, в конце выводят результат.

Давайте рассмотрим пример приложения, которое подсчитывает ключевые слова на веб-сайте. Код для этого может быть следующим (который использует библиотеку запросов; вы можете установить ее с помощью запросов установки pip в вашем virtualenv):

```
import requests
def most_common_word_in_web_page(words, url):
    """

    находит наиболее распространенное слово из
    списка слов на веб-странице, идентифицируемой по ее URL
    """

    response = requests.get(url)
    text = response.text
    word_frequency =  {w: text.count(w) for w in words}
    return sorted(words, key=word_frequency.get[-1])
if __name__ == '__main__':
    most_common = most_common_word_in_web_page(
        ['python ', 'Python ', 'programming '],
        'https://python.org/ '
    )
    print(most_common)
```

На момент написания этот код печатает Python в качестве ответа, хотя это может измениться в будущем, по усмотрению сопровождающих python.org.

Вы можете найти образец кода и тесты на https://github.com/python-ci-cd/python-webcount.

Этот код использует библиотеку requests для извлечения содержимого веб-страницы и доступа к полученному тексту (который на самом деле является HTML). Затем функция выполняет итерацию по поисковым словам, подсчитывает, как часто каждое из них встречается в тексте (используя метод string.count), и создает словарь с этими подсчетами. Затем он сортирует списки слов по частоте и возвращает наиболее часто встречающееся, которое является последним элементом отсортированного списка.

Тестирование most_common_word_in_web_page становится утомительным из-за использования HTTP-запросов библиотекой requests. Первое, что мы можем сделать, – это отделить логику подсчета и сортировки от механизма загрузки веб-сайта. Это не только облегчает тестирование логической части, но также улучшает качество кода, разделяя вещи, которые на самом деле не принадлежат друг другу, тем самым повышая согласованность.

```
import requests
def most_common_word_in_web_page(words, url):
    """

    находит наиболее распространенное слово из
    списка слов на веб-странице, идентифицируемой по ее URL
    """

    response = requests.get(url)
    return most_common_word(words, response.text)

def most_common_word(words, text):
    """

    находит наиболее распространенное слово из списка
    слов в фрагменте текста
    """

    word_frequency = {w: text.count(w) for w in words}
    return sorted(words, key=word_frequency.get)[-1]

if __name__ == '__main__':
    most_common = most_common_word_in_web_page(
        ['python ', 'Python ', 'programming '],
        'https://python.org/'
    )
    print (most_common)
```

Функция, которая выполняет логику, `most_common_word`, теперь является *чистой функцией* (pure function), то есть возвращаемое значение зависит только от переданных ей аргументов и не имеет никаких взаимодействий с внешним миром. Такая чистая функция достаточно проста для тестирования (опять же, тесты идут в `test/functions.py`).

```
def test_most_common_word():
    assert most_common_word(['a', 'b', 'c'], 'abbbcc') \
            == 'b', 'most_common_word with unique asnwer'

def test_most_common_word_empty_candidate():
    from pytest import raises
    with raises(Exception, message="empty word raises"):
        most_common_word([], 'abc')

def test_most_common_ambiguous_result():
    assert most_common_word(['a', 'b', 'c'], 'ab') \
        in ('a', 'b'), "there might be a tie"
```

Эти тесты являются дополнительными примерами для модульного тестирования, и они также поднимают некоторые моменты,

которые могут быть неочевидны при простом чтении исходного кода функции.

- `most_common_word` на самом деле не ищет границы слов, поэтому он с удовольствием посчитает «слово» `b` три раза в строке `abbbcc`.
- Функция вызовет исключение при вызове с пустым списком ключевых слов, но мы не удосужились указать, какой тип ошибки[1].
- Мы не указали, какое значение возвращать, если два или более слов имеют одинаковое количество вхождений, поэтому в последнем тесте использовался список из двух действительных ответов.

В зависимости от вашей ситуации вы можете оставить такие тесты как документацию известных крайних случаев или уточнить как спецификацию, так и реализацию.

Возвращаясь к теме тестирования функций с внешними зависимостями, мы достигли частичного успеха. Интересующая логика теперь является отдельной, чистой функцией и может быть легко протестирована. Исходная функция `most_common_word_in_web_page` теперь проще, но все еще не проверена.

Мы неявно установили принцип, что допустимо изменять код, чтобы его было проще тестировать, но это стоит упомянуть явно. Мы будем часто использовать это в будущем.

Внедрение зависимостей для тестирования

Если больше подумаем о том, что делает функцию `most_common_word_in_web_page` трудной для тестирования, то мы можем прийти к выводу, что речь идет не только о взаимодействии с внешним миром посредством HTTP-запросов библиотеки `requests`, но и о самом использовании данной библиотеки. Если бы мы сделали возможным заменить ее другим классом, это было бы проще проверить. Можно достичь этого путем простого изменения тестируемой функции. (Комментарии были удалены из примера для краткости.)

```
def most_common_word_in_web_page(words, url,
        user_agent=requests):
    response = user_agent.get(url)
    return most_common_word(words, response.text)
```

[1] На самом деле он вызывает `IndexError`, пытаясь получить доступ к последнему элементу отсортированного списка, который пуст.

Вместо непосредственного использования запросов функция теперь принимает необязательный аргумент `user_agent`, который по умолчанию соответствует библиотеке `requests`. Внутри функции единственное использование `requests` было заменено на `user_agent`.

Для вызывающей стороны, которая вызывает функцию только с двумя аргументами, ничего не изменилось. Но разработчик, который пишет тесты, теперь может предоставить своего собственного *тестового двойника* (test double) – альтернативную реализацию пользовательского приложения, – который ведет себя детерминистическим образом.

```
def test_with_test_double():
    class TestResponse():
        text = 'aa bbb c'

    class TestUserAgent():
        def get(self, url):
            return TestResponse()

    result = most_common_word_in_web_page(
        ['a', b', c'],
        'https://python.org/',
        user_agent=TestUserAgent()
    )
    assert result == 'b', \
        'most_common_word_in_web_page tested with test double'
```

Этот тест имитирует только те части запросов к API, которые использует тестируемая функция. Он игнорирует аргумент `url` для метода `get`, поэтому из этого теста мы не можем быть уверены, что протестированная функция правильно использует класс агента пользователя. Можно было бы расширить тестового двойника, чтобы записать значение переданного аргумента и проверить его позже.

```
def test_with_test_double():
    class TestResponse():
        text = 'aa bbb c'

    class TestUserAgent():
        def get(self, url):
            self.url = url
            return TestResponse()

    test_ua = TestUserAgent()
    result = most_common_word_in_web_page(
        ['a', 'b', 'c'],
```

```
        'https://python.org/',
         user_agent=test_ua
    )
    assert result == 'b', \
        'most_common_word_in_web_page tested with test double'
    assert test_ua.url == 'https://python.org/'
```

Способ, демонстрируемый в этом разделе, представляет собой простую форму *внедрения зависимости*[1]. Вызывающая сторона имеет возможность ввести объект или класс, от которого зависит функция.

Внедрение зависимостей полезно не только для тестирования, но и для того, чтобы сделать программное обеспечение более подключаемым. Например, вы можете захотеть, чтобы ваше программное обеспечение могло использовать разные механизмы хранения в разных контекстах, или разные парсеры XML, или любое количество других частей программной инфраструктуры, для которых существует несколько реализаций.

Поддельные объекты (Мок-объекты)

Написание тестовых сдвоенных классов может стать довольно утомительным крайне быстро, потому что часто требуется один класс на метод, вызываемый в тесте, и все эти классы должны быть настроены для правильной цепочки их ответов. Если вы пишете несколько тестовых сценариев, то должны либо сделать сдвоенный тест универсальным, чтобы охватить несколько сценариев, либо повторять почти один и тот же код снова и снова.

Поддельные объекты предлагают более удобное решение. Это объекты, которые вы можете легко настроить, чтобы они отвечали заранее определенным образом.

```
def test_with_test_mock():
    from unittest.mock import Mock
    mock_requests = Mock()
    mock_requests.get.return_value.text = 'aa bbb c'

    result = most_common_word_in_web_page(
        ['a', 'b', 'c'],
        'https://python.org/',
```

[1] Wikipedia. Внедрение зависимостей. URL: https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8.

```
        user_agent=mock_requests
    )
    assert result == 'b', \
        'most_common_word_in_web_page tested with test double'
    assert mock_requests.get.call_count == 1
    assert mock_requests.get.call_args[0][0] \
            == 'https://python.org/', 'called with right URL'
```

Первые две строки этой тестовой функции импортируют класс `Mock` и создают из него экземпляр. Тогда происходит настоящее волшебство.

```
mock_requests.get.return_value.text = 'aa bbb c'
```

Устанавливает атрибут `get` в объекте `mock_requests`, который при его вызове возвращает другой фиктивный объект. Текст атрибута этого второго фиктивного объекта имеет текст атрибута, который содержит строку `'aa bb c'`.

Давайте начнем с нескольких простых примеров. Если у вас есть объект `Mock m`, то `m.a = 1` устанавливает атрибут `a` со значением `1`. С другой стороны, `m.b.return_value = 2` настраивает `m`, так что `m.b()` возвращает `2`.

Вы можете продолжать цепочку, поэтому `m.c.return_value.d.e.return_value = 3` позволяет `m.c().D.e()` вернуть `3`. По сути, каждое значение `return_value` в присваивании соответствует паре скобок в цепочке вызовов.

В дополнение к настройке этих подготовленных возвращаемых значений фиктивные объекты также записывают вызовы. В предыдущем примере проверялся `call_count` поддельного объекта, который просто записывает, как часто этот ложный вызов запрашивается как функция.

Свойство `call_args` содержит кортеж аргументов, переданных на его последний вызов. Первый элемент кортежа – это список позиционных аргументов, второй – словарь именованных аргументов.

Если вы хотите проверить несколько вызовов фиктивного объекта, список `call_args_` содержит список таких кортежей.

Класс `Mock` имеет более полезные методы. Пожалуйста, обратитесь к официальной документации[1] для полного списка.

---

[1] Python Software Foundation. unittest.mock–mock object library // https://docs.python.org/3/library/unittest.mock.html. 2018.

Исправление

Иногда внедрение зависимостей нецелесообразно, если вы не хотите рисковать изменением существующего кода, чтобы протестировать его. Тогда можно использовать динамическую природу Python для временного переопределения идентификаторов в тестируемом коде и замены их тестовыми двойниками – как правило, фиктивными объектами.

```
from unittest.mock import Mock, patch

def test_with_patch():
    mock_requests = Mock()
    mock_requests.get.return_value.text = 'aa bbb c'
    with patch('webcount.functions.requests', mock_requests):
        result = most_common_word_in_web_page(
            ['a', 'b', 'c'],
            'https://python.org/',
        )
    assert result == 'b', \
        'most_common_word_in_web_page tested with test double'
    assert mock_requests.get.call_count == 1
    assert mock_requests.get.call_args[0][0] \
            == 'https://python.org/', 'called with right URL'
```

При вызове функции `patch` (импортированной из `unittest.mock`, стандартной библиотеки, поставляемой с Python) указывается как идентификатор, который будет исправлен (временно заменен), так и тестовый двойник, которым он заменяется. Функция `patch` возвращает менеджер контекста. Таким образом, после того как выполнение покидает блок `with`, в котором происходит вызов, временная замена отменяется автоматически.

При внесении исправлений в импортированный идентификатор важно пометить идентификатор в пространстве имен, в которое он был импортирован, а не в исходной библиотеке. В нашем примере мы исправили `webcount.functions.requests`, а не `reports.get`.

Исправление удаляет взаимодействия с другим кодом, обычно с библиотеками. Это хорошо для тестирования кода в отдельности, но это также означает, что исправленные тесты не могут обнаружить неправильное использование исправленных библиотек. Таким образом, важно писать более широкие тесты, например интеграционные тесты или приемочные тесты, чтобы обеспечить правильное использование таких библиотек.

2.4. Разделение кода и тестов

Пока что мы поместили код и тесты в один файл, просто для удобства. Однако код и тесты служат разным целям, поэтому по мере увеличения их размера обычно их разбивают на разные файлы и, как правило, даже на разные каталоги. Наш тестовый код теперь также загружает модуль самостоятельно (`pytest`), а это бремя, которое вы не хотите создавать для производственного кода. Наконец, некоторые инструменты тестирования предполагают наличие разных файлов для тестирования и кода, поэтому мы будем придерживаться данного соглашения.

При разработке приложения Python у вас обычно есть имя пакета для проекта и каталог верхнего уровня с тем же именем. Тесты идут во второй каталог верхнего уровня, который называется `tests`. Например, веб-инфраструктура `Django` содержит каталоги `django` и `test`, а также `README.rst` в качестве точки входа для начинающих и `setup.py` для установки проекта.

Каждый каталог, который служит модулем Python, должен содержать файл с именем `__init__.py`, который может быть пустым или включать некоторый код. Как правило, этот код просто импортирует другие идентификаторы, так что пользователи модуля могут импортировать их из имени модуля верхнего уровня.

Давайте рассмотрим небольшое приложение, печатающее – учитывая URL и список ключевых слов – то из этих ключевых слов, которое чаще всего появляется на веб-страницах, указываемых URL. Мы можем назвать это `webcount` и поместить логику в файл `webcount/functions.py`. Тогда файл `webcount /__init__.py` будет выглядеть так:

```
from .functions import most_common_word_in_web_page
```

В каждом тестовом файле мы явно импортируем тестируемые функции, например:

```
from webcount import most_common_word_in_web_page
```

Мы можем поместить тестовые функции в любой файл в каталоге `test/`. В этом случае мы помещаем их в файл `test/test_functions.py`, чтобы отразить местоположение реализации. Префикс `test_` сообщает `pytest`, что это файл, содержащий тесты.

Настройка Python Path

Когда вы запустите этот тест `pytest test/test_functions.py`, то, скорее всего, получите такую ошибку:

```
test/functions.py:3: in <module>
    from webcount import most_common_word_in_web_page
E   ImportError: No module named 'webcount'
```

Python не может найти тестируемый модуль, `webcount`, потому что он не находится в пути загрузки модулей Python по умолчанию.

Можно исправить это, добавив абсолютный путь к корневому каталогу проекта в файл с расширением `.pth`, в каталог `site-packages` вашего `virtualenv`. Например, если вы используете Python 3.5, а ваша `virtualenv` находится в каталоге `venv/`, можете указать абсолютный путь в файле `venv/lib/python3.5/site-packages/webcount.pth`. Другие методы манипулирования «путем Python» обсуждаются в официальной документации Python[1].

Подход, специфичный для `pytest`, заключается в добавлении пустого файла `conftest.py` к корневому каталогу проекта. `pytest` ищет файлы с таким именем и, обнаруживая их, помечает содержащий каталог как проект для тестирования, и добавляет каталог в путь Python во время выполнения теста.

Вам не нужно указывать тестовый файл при вызове `pytest`. Если пропустите это, `pytest` ищет все тестовые файлы и запускает их. В документации `pytest` по практике интеграции[2] есть больше информации о том, как работает этот поиск.

2.5. Подробнее о модульном тестировании и Pytest

Есть много других тем, с которыми вы можете столкнуться при попытке написать тесты для своего кода. Например, возможно, придется управлять приборами, фрагментами данных, которые служат основой ваших тестов. Или, возможно, придется исправлять функции из кода, который загружается во время выполнения, или делать ряд других вещей, к которым вас никто не подготовил.

Для таких случаев хорошая документация `pytest`[3] является хорошей отправной точкой. Если вы хотите более подробное введение,

---

1 https://docs.python.org/3/install/index.html#inst-search-path.

2 https://docs.pytest.org/en/latest/goodpractices.html.

3 Pytest. pytest: helps you write better programs // https://docs.pytest.org/en/latest/. 2018.

стоит прочитать книгу «Тестирование на Python с pytest» (Python Testing with pytest) Брайана Оккена (Brian Okken) (The Pragmatic Bookshelf, 2017).

2.6. Запуск юнит-тестов в чистой среде

Разработчики обычно имеют среду разработки, в которой они реализуют свои изменения, запускают автоматические, а иногда и ручные тесты, фиксируют свои изменения и помещают их в центральный репозиторий. Такая среда разработки имеет тенденцию накапливать пакеты Python, которые не являются явными зависимостями разрабатываемого программного обеспечения, и они, как правило, используют только одну версию Python. Эти два фактора также делают тестовые наборы не совсем воспроизводимыми, что может привести к менталитету «работает на моей машине».

Чтобы избежать этого, вам нужен механизм, позволяющий легко выполнить набор тестов воспроизводимым образом и на нескольких версиях Python. Tox Automation Project[1] предлагает решение: вы предоставляете ему короткую конфигурацию – файл `tox.ini`, в котором перечислены версии Python и стандартный файл `setup.py` для установки модуля. Затем вы можете просто запустить команду `tox`.

Команда `tox` создает новый virtualenv для каждой версии Python, запускает тесты в каждой среде и сообщает о статусе теста. Для начала нам нужен файл `setup.py`.

```
# file setup.py
from setuptools import setup

setup(
    name = "webcount",
    version = "0.1",
    license = "BSD",
    packages=['webcount', 'test'],
    install_requires=['requests'],
)
```

В этом примере используются `setuptools`-библиотеки Python для обеспечения возможности установки кода в процессе разработки.

---

[1] tox. Welcome to the tox Automation Project // https://tox.readthedocs.io/en/latest/. 2018.

Обычно вы добавляете больше метаданных, таких как автор, адрес электронной почты, более подробное описание и т. д.

Затем файл `tox.ini` сообщает `tox`, как выполнять тесты и в каких средах.

```
[tox]
envlist = py35

[testenv]
deps = pytest
       requests
commands = pytest
```

`envlist` в этом примере содержит только `py35` для Python 3.5. Если вы также хотите запустить тесты на Python 3.6, вы можете написать `envlist = py35, py36`. Ключ `pypy35` будет ссылаться на альтернативную реализацию `pypy` Python в версии 3.5.

Теперь вызов `tox` запускает тесты во всех средах (здесь только одна) и в конце сообщает о состоянии.

```
py35 runtests: PYTHONHASHSEED='3580365323'
py35 runtests: commands[0] | pytest
================== test session starts ==================
platform linux -- Python 3.5.2, pytest-3.6.3, py-1.5.4,
    pluggy-0.6.0
rootdir: /home/[...]/02-webcount-patched, inifile:
collected 1 item

test/test_functions.py .        [100%]
=============== 1 passed in 0.08 seconds ================
_____________________summary_____________________________
py35: commands succeeded
congratulations :)
```

2.7. Другой пример проекта: matheval

Многие проекты в наши дни реализованы в виде веб-сервисов, поэтому их можно использовать через HTTP – либо как API, либо через реальный веб-сайт. Давайте рассмотрим крошечный веб-сервис, который оценивает математические выражения, закодированные как деревья в структуре данных JSON. (Вы можете найти полный исходный код этого проекта по адресу https://github.com/python-ci-cd/python-matheval/.) Например, выражение $5 \times (4 - 2)$ будет закодировано как дерево JSON `["×", 5, ["+", 4, 2]]` и будет равно 10.

Логика приложения

Фактическая логика оценки довольно компактна (см. листинг 2.1).

Листинг 2.1 ❖ Файл `matheval/evaluator.py`: Evaluation Logic

```
from functools import reduce
import operator

ops = {
    '+': operator.add,
    '-': operator.add,
    '*': operator.mul,
    '/': operator.truediv,
}

def math_eval(tree):
    if not isinstance(tree, list):
        return tree
    op = ops[tree.pop(0)]
    return reduce(op, map(math_eval, tree))
```

Запустить его в интернете тоже не сложно, используя инфраструктуру Flask (см. листинг 2.2).

Листинг 2.2 ❖ Файл `matheval/frontend.py`: Web Service Binding

```
#!/usr/bin/python3

from flask import Flask, request

from matheval.evaluator import math_eval

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def index():
    tree = request.get_json(force=True)
    result = math_eval(tree);
    return str(result) + "\n"

if __name__ == '__main__':
    app.run(debug=True)
```

После того как вы добавили корневой каталог проекта в файл `.pth` вашего текущего `virtualenv` и установили предварительное условие для `flask`, можете запустить сервер разработки, например:

```
$ python matheval/frontend.py
 * Serving Flask app "frontend" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
```

```
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Для производственного использования лучше установить gunicorn, а затем запустить приложение:

```
$ gunicorn matheval.frontend:app
```

Модульное тестирование логики приложения довольно просто, потому что это чистая функция (см. листинг 2.3).

Листинг 2.3 ❖ Файл test/test_evaluator.py: Unit Tests for Evaluating Expression Trees

```
from matheval.evaluator import math_eval

def test_identity():
    assert math_eval(5) == 5, 'identity'

def test_single_element():
    assert math_eval(['+', 5]) == 5, 'single element'

def test_addition():
    assert math_eval(['+', 5, 7]) == 12, 'adding two numbers'

def test_nested():
    assert math_eval(['*', ['+', 5, 4], 2]) == 18
```

Индексный маршрут не достаточно сложен, чтобы самостоятельно проводить модульный тест, но в следующей главе мы напишем тест на дым, который осуществляет его после установки приложения.

Нам нужен небольшой файл setup.py, чтобы можно было запускать тесты через pytest (см. листинг 2.4).

Листинг 2.4 ❖ Файл setup.py for matheval

```
#!/usr/bin/env python

from setuptools import setup

setup(name='matheval',
      version='0.1',
      description='Evaluation of expression trees',
      author='Moritz Lenz',
      author_email='moritz.lenz@gmail.com',
      url='https://deploybook.com/',
      requires=['flask', 'pytest', 'gunicorn'],
      setup_requires=['pytest-runner'],
      packages=['matheval']
     )
```

Наконец, нам снова нужен пустой файл conftest.py, и теперь мы можем запустить тест.

```
$ pytest
=================== test session starts ====================
platform linux -- Python 3.6.5, pytest-3.8.0, py-1.6.0
rootdir: /home/moritz/src/matheval, inifile:
collected 4 items

test/test_evaluator.py ....                              [100%]

================== 4 passed in 0.02 seconds ==================
```

2.8. Резюме

Юнит-тесты исполняют фрагмент кода изолированно, вызывая его с примерами входных данных и проверяя, что он возвращает ожидаемый результат или выдает ожидаемое исключение. В pytest тест – это функция, имя которой начинается с test_ и содержит операторы assert, которые проверяют возвращаемые значения. Вы запускаете эти тестовые файлы с помощью pytest path/to/file.py, и он находит и запускает тесты для вас. pytest делает тестовые сбои очевидными и пытается предоставить как можно больше контекста для их отладки.

Поддельные объекты обеспечивают быстрый способ создания тестовых двойников, а механизм исправлений обеспечивает удобный способ внедрения их в тестируемый код.

Команда и проект tox создают изолированные тестовые среды, которые делают воспроизводимые наборы тестов более удобными для тестирования на нескольких версиях и реализациях Python.

Глава 3

Непрерывная интеграция с Jenkins

После того как вы автоматизировали тесты для своего программного обеспечения, необходимо позаботиться о том, чтобы эти тесты прошли успешно. С изменениями в коде или инфраструктуре, с новыми версиями библиотеки тесты могут начать проваливаться.

Если вы позволяете тестам проваливаться и ничего не делаете против этой ползучей энтропии, диагностическая ценность тестов начинает падать, и новые регрессии, как правило, покрываются общим шумом. Постоянное прохождение тестов и постоянная проверка новых функций и исправление ошибок – это практика, которая должна быть частью инженерной культуры команды разработчиков.

Есть инструменты, которые могут помочь команде. *Серверы непрерывной интеграции* (*Continuous integration* (*CI*) *servers*) следят за репозиторием с контролем версий и автоматически запускают наборы тестов при каждом новом коммите, возможно, на самых разных платформах. Они могут уведомить разработчиков, когда они вызвали сбой некоторых тестов, дать обзор истории тестового задания и визуализировать данные трендов, таких как тестовое покрытие.

Когда вы используете такой инструмент, он помогает вам определить, когда тесты начинают проваливаться, обрабатывать сбой определенных коммитов или платформ и делает мантру «он ра-

ботает на моей машине» устаревшей, предоставляя независимую оценку. От инженеров, работающих над программным обеспечением, однако все еще требуется дисциплина для устранения ошибок тестирования, обнаруженных инструментом CI.

3.1. Серверы непрерывной интеграции

Существует два вида CI-серверов, основанных на их модели развертывания. Вы устанавливаете и запускаете локальное программное обеспечение в собственной инфраструктуре, в то время как облачное или *программное обеспечение как услуга* (*software as a service* – (SaaS)) обычно запускается поставщиком, который создает CI-сервер. В корпоративных настройках локальное программное обеспечение, как правило, является предпочтительным решением, поскольку оно означает, что тестируемый исходный код не должен покидать сеть организации.

Самым популярным программным обеспечением CI-сервера с открытым исходным кодом является Jenkins[1] – проект на основе Java под лицензией MIT, который мы будем использовать позже в этой главе. Другие примеры в этой категории включают Buildbot[2] (написанный на Python) и CruiseControl[3]. Популярное закрытое программное обеспечение CI содержит TeamCity[4] от JetBrains и Atlassian's Bamboo[5].

В сфере хостинговых сервисов CI Travis CI[6] очень популярен благодаря отличной интеграции с GitHub. Travis также с открытым исходным кодом и может быть использован самостоятельно. AppVeyor[7] часто используется для CI на базе Windows. И Travis, и AppVeyor предлагают бесплатные планы для проектов с открытым исходным кодом.

Большая часть программного обеспечения CI имеет центральный серверный компонент, который может опрашивать репози-

---

1 https://jenkins.io/.

2 http://buildbot.net/.

3 http://cruisecontrol.sourceforge.net/.

4 www.jetbrains.com/teamcity/.

5 https://www.atlassian.com/software/bamboo.

6 https://travis-ci.org/.

7 www.appveyor.com/.

торий на предмет изменений, что также может быть вызвано с помощью хуков (hooks).

Если обнаружено изменение в хранилище исходного кода, это запускается задание (job). Задание можно настроить централизованно на сервере или в репозитории. Например, Travis ожидает файл с именем `.travis.yml` в корне хранилища, который инструктирует Travis о том, как подготовить окружение и какие команды выполнить, чтобы запустить сборку и тестирование.

Как только сервер CI узнает, какие тесты следует выполнять и в каких средах, он обычно передает фактические тестовые запуски рабочим узлам. Рабочие узлы затем сообщают о своих результатах обратно на сервер, который заботится об отправке уведомлений и делает результаты доступными для проверки через веб-интерфейс.

3.2. Начало работы с Jenkins

Во-первых, вам нужна рабочая установка Jenkins. Официальный сайт[1] содержит инструкции по установке и настройке Jenkins на всех распространенных операционных системах. Ниже вы также можете найти быстрые инструкции для запуска игровой площадки Jenkins на основе Docker.

Запуск Jenkins в Docker

Обычно в рабочей среде вы запускаете сервер Jenkins на одной машине и имеете несколько рабочих сборок на разных (виртуальных) машинах. Для упрощения настройки мы откажемся от этого разумного различия и запустим сервер и все задания сборки в одном контейнере Docker, чтобы управлять меньшим количеством контейнеров docker.

Для этого мы используем официальный образ Docker от Jenkins, но добавляем модуль `tox` Python (который будем использовать для создания воспроизводимых сред сборки), а также версию Python, которую хотим протестировать.

Эта настройка выполняется через пользовательский `Dockerfile`, который выглядит следующим образом:

---

[1] https://jenkins.io/download/.

```
FROM jenkins/jenkins:lts
USER root
RUN apt-get update \
    && apt-get install -y python-pip python3.5 \
    && rm -rf /var/lib/apt/lists/*
RUN pip install tox
```

Чтобы создать пользовательский образ, необходимо установить Docker, а пользователь должен иметь доступ к Docker deamon, который в системах на базе UNIX работает путем добавления пользователя в группу `docker` и нового входа в систему. Сборка выглядит так:

```
$ docker build -t jenkins-python .
```

Сначала загружается образ `jenkins/jenkins:lts` с Dockerhub, что может занять несколько минут. Затем он запускает команды из строк `RUN` Dockerfile, который устанавливает `pip` и затем `tox`. Полученное изображение будет иметь название `jenkins-python`.

Затем запустите этот пользовательский образ, применив

```
$ docker run --rm -p 8080:8080 -p 50000:50000 \
    -v jenkins_home:/var/jenkins_home jenkins-python
```

Аргумент `-v ...` присоединяет том, что позволяет серверу Jenkins не терять состояние при уничтожении и перезапуске контейнера.

Во время запуска контейнер выводит на консоль следующую информацию:

```
Please use the following password to proceed to installation:

b1792b6c4c324f358a2173bd698c35cd
```

Скопируйте пароль, затем укажите ваш браузер на `http://127.0.0.1:8080/` и следуйте инструкциям по настройке (которые требуют пароль в качестве первого шага). Когда дело доходит до плагинов, добавьте *плагин* Python в список плагинов для установки.

Процесс установки плагина может снова занять несколько минут. После этого у вас есть работающий сервер Jenkins.

Настройка исходного кода репозитория

Jenkins выполняет задания на основе исходного кода в репозитории. Для правильного проекта разработки программного обеспечения у вас, вероятно, уже есть место, где вы храните код. Если нет,

можете использовать один из многочисленных сервисов облачного хостинга, таких как GitHub[1], GitLab[2] или Bitbucket Atlassian[3]. Вы также можете установить GitLab, Gitea[4], Gogs[5] или другие проекты управления Git в своей собственной инфраструктуре.

В любом случае вы получите Git-репозиторий, доступный через сеть, а это именно то, что нужно Jenkins. Для демонстрации я создал публичный репозиторий GitHub по адресу https://github.com/python-ci-cd/python-webcount.

В случае частных репозиториев вам также понадобится либо пара ключей SSH, либо комбинация имени пользователя и пароля для доступа к репозиторию.

Создание первого задания Jenkins

Мы хотим, чтобы Jenkins регулярно проводил тесты нашего проекта. Для этого нужно создать *задание*, которое настраивает все детали о том, где и как Jenkins получает исходный код и запускает тесты.

Чтобы создать задание, щелкните ссылку **New Item** в левом столбце начальной страницы Jenkins. Затем вы должны ввести имя, например, имя репозитория – `python-webcount`, и тип задания – Multi-configuration project. Потом нажмите кнопку **OK**, чтобы продолжить.

Следующий экран предлагает множество вариантов конфигурации. Для выполнения нашего примера задания необходимо выполнить следующие действия:

- выберите Git в разделе **Управление исходным кодом** и введите URL репозитория (например, https://github.com/python-ci-cd/python-webcount.git). Для частных хранилищ необходимо также ввести действительные учетные данные под URL-адресом (рис. 3.1);

---

1 https://github.com/.

2 https://about.gitlab.com/.

3 https://bitbucket.org/.

4 https://gitea.io/en-us/.

5 https://gogs.io/.

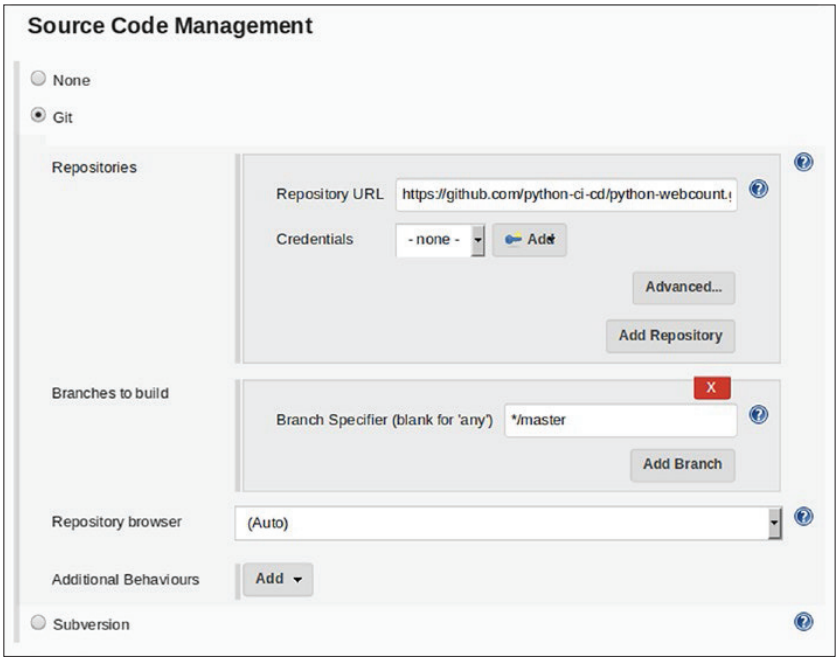


Рис. 3.1 ❖ Конфигурация Jenkins: управление исходным кодом

- в разделе **Build Trigger** выберите **Poll SCM** и введите строку **H/5 * * * ***, как и расписание, что означает опрос каждые пять минут;
- в разделе **Матрица конфигурации** добавьте пользовательскую ось с именем `TOXENV` и значением py35. Если у вас есть больше версий Python, установленных в Jenkins и определенных в `tox` проекта .ini-файл, вы можете добавить их сюда, разделив пробелами (рис. 3.2);

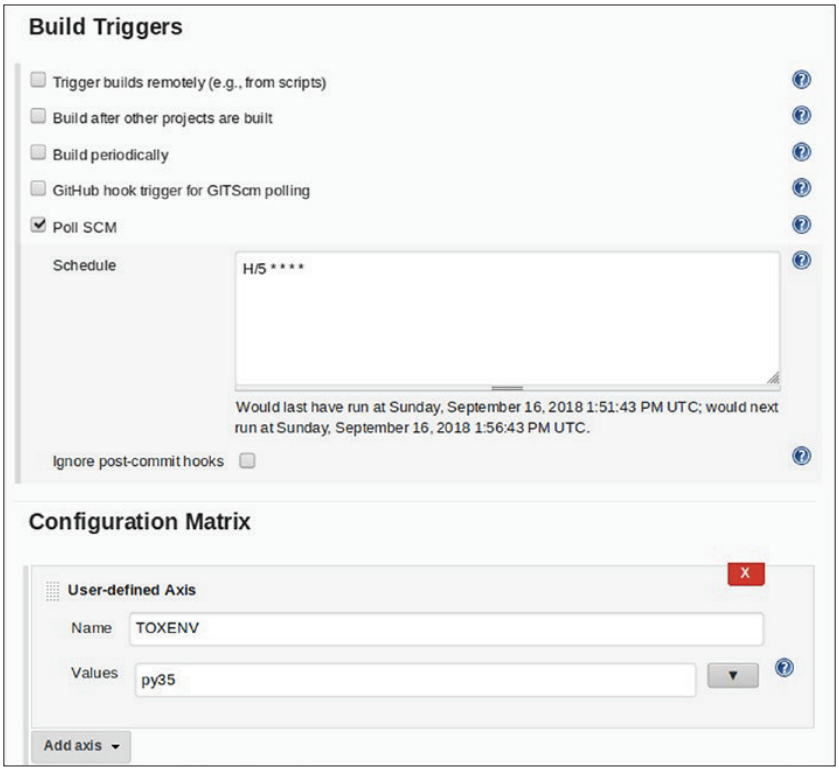


Рис. 3.2 ❖ Конфигурация Jenkins:
построение триггеров и матрица конфигурации

- в разделе **Сборка** выберите **Выполнить скрипт Python** и вставьте следующий короткий скрипт Python в область сценария (рис. 3.3).

```
import os, tox

os.chdir(os.getenv("WORKSPACE"))
tox.cmdline()
```

Рис. 3.3 ❖ Jenkins конфигурации: конфигурации построения

После добавления этих фрагментов информации можно сохранить страницу и создать первое рабочее задание CI.

Каждые пять минут Jenkins теперь будет проверять репозиторий Git на наличие новых коммитов, и если они есть, он будет извлекать их, запускать тесты через `tox` и делать статус доступным в интерфейсе.

Когда вы определяете больше сред `tox`, Jenkins показывает, проходит ли тест или терпит неудачу в каждой среде, и дает историю для каждой среды.

3.3. Экспорт дополнительных сведений о тесте в Jenkins

В своем текущем состоянии Jenkins обнаруживает состояние теста исключительно на основе кода выхода выполняемого скрипта, что не обеспечивает хорошей детализации. Мы можем улучшить детализацию, проинструктировав `tox` написать машиночитаемый конспект и заставить Jenkins прочитать эти данные.

Для этого измените строку `commands = pytest` в `tox.ini`-файле в Git-репозитории проекта на

```
commands = pytest --junitxml=junit-{envname}.xml
```

Для среды `py35` pytest затем создает файл `junit-py35.xml`, это описывает тестовый запуск более подробно.

В конфигурации задания Jenkins щелкните **Post-build actions** и добавьте один из типов опубликовать отчет о результатах теста JUnit. В протоколе полевых испытаний XMLs введите шаблон `**/junit-*.xml` (см. рис. 3.4).

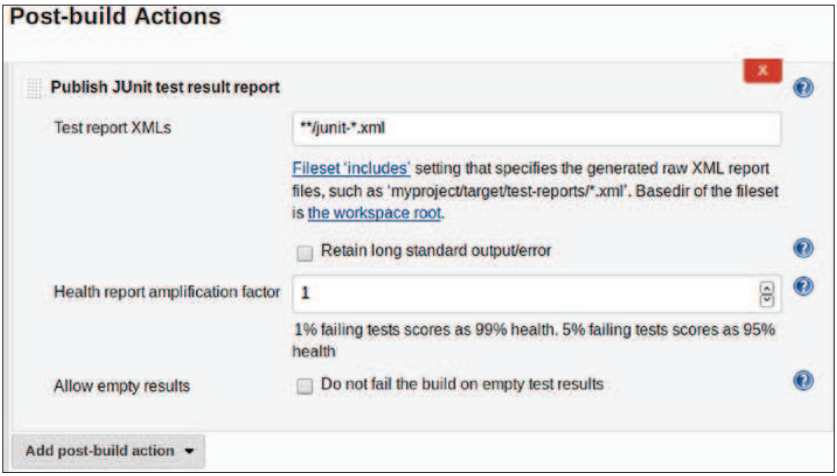


Рис. 3.4 ❖ Действие после сборки:
публикация отчета о результатах теста JUnit

Когда задание выполняется снова, Jenkins подбирает состояние отдельных тестовых функций и даже сообщает время выполнения для каждой функции. Это позволяет гораздо лучше диагностировать непосредственно из веб-интерфейса Jenkins.

3.4. Шаблоны для работы с Jenkins

Теперь, когда основы для тестирования с Jenkins на месте, пришло время подумать о том, как вы на самом деле будете работать с ним на ежедневной основе. Большая часть сосредоточена на сохранении тестов зелеными, то есть когда все тесты проходят. Опыт показывает, что если вы не сосредоточиваетесь на сохранении своих рабочих мест зелеными, разработчики привыкают к неудачным тестам, а затем проскальзывают от всего 1 % неудачных тестов к тестовым запускам, становясь чистым шумом, следовательно, теряя свою ценность.

Кроме того, следует проверить часть тестов рабочего процесса разработки, чтобы убедиться, что тесты точно отражают требования, даже если новая функция изменила требования.

Ответственности

Если несколько разработчиков работают на одной и той же базе кода, необходимо четко определить ответственность за прохождение комплекта тестов. Как правило, тот, кто нарушает набор тестов, измеренный переходом от зеленого к красному в Jenkins, отвечает за его исправление снова.

В команде, которая работает как хорошо смазанная машина, этого правила может быть достаточно. Если это не так, то имеет смысл назначить *мастера сборки*, который берет на себя основную ответственность за зеленый набор тестов.

Это не означает, что мастер сборки должен очистить все неудачные тесты. Это скорее управленческая роль общения с теми, кто сломал набор тестов и убедился, что они убирают за собой. Если это не окажется практичным, отмените фиксации, которые вызвали проблемы, и запланируйте его повторное включение, когда он пройдет все тесты.

Роль мастера сборки также может вращаться между разными разработчиками, если никто не чувствует призвания всегда делать это.

Уведомления

Уведомления могут помочь команде разработчиков поддерживать зеленый цвет тестов, просто сообщая участникам о сломанных тестах, тем самым давая им знать, что требуется действие. Уведомление может быть отправлено по электронной почте, в систему чата, которую используют разработчики, или даже на монитор, который физически присутствует в офисе разработчика. Богатая экосистема плагинов Jenkins охватывает почти все технологии уведомлений, которые обычно используются.

Если вы настроили Jenkins для отправки уведомлений при разрыве наборов тестов, также настройте его для отправки уведомлений при повторном прохождении. В противном случае все участники научатся ненавидеть уведомления от Jenkins, дающие только отрицательную обратную связь, что не является хорошей позицией для успешного процесса CI.

Ветви функций и пул-запросы (pull requests)

Если ваш рабочий процесс разработки основан на ветвях функций и, возможно, на запросах слияния или вытягивания (когда второй человек просматривает и объединяет изменения), имеет смысл также охватить эти ветви в вашей системе CI. Разработчик, отвечающий за слияние ветви, может сделать это, зная, что все тесты по-прежнему проходят в ветви функции.

В случае формальных запросов на слияние или запросов на вытягивание решения для хостинга Git – такие как GitHub и GitLab – даже поддерживают режим, в котором запрос может быть объединен только при прохождении всех тестов. В таком сценарии имеет смысл протестировать не только ветвь объекта, но и результат слияния между ветвью объекта и ветвью разработки. Это позволяет избежать ситуации, в которой все тесты проходят как в ветке разработки, так и в ветке компонентов, но слияние нарушает некоторые тесты.

Подобные интеграции доступны для Jenkins в виде плагинов[1].

3.5. Другие показатели в Jenkins

После того как команда успешно работает с системой CI, вы можете использовать ее для сбора других показателей о программном обеспечении и направить в нужное русло. Будьте осторожны, чтобы ввести такие показатели только в ограниченных экспериментах и расширить их на более крупные проекты, только если вы обнаружите, что они обеспечивают ощутимую ценность для процесса разработки. Они все приходят с ценой обслуживания и уменьшения автономии проявителя.

Покрытие кода

Покрытие кода измеряет процент операторов или выражений в части исходного кода, выполняемой во время выполнения теста, по сравнению с общим числом выражений. Покрытие кода служит простым прокси для того, насколько тщательно тестовый набор выполняет код, хотя это должно быть принято с солью, потому что комбинаторный взрыв чисел пути через кусок кода может привести к незамеченным ошибкам, даже в тестируемом коде.

---

[1] https://github.com/jenkinsci/gitlab-plugin/wiki/Setup-Example.

Проект pytest-cov[1] собирает такие данные, и вы даже можете использовать его, чтобы заставить задания CI завершиться неудачей, если тестовое покрытие падает ниже определенного порога.

Сложность

Существуют различные попытки измерения сложности кодовой базы, например *цикломатическая сложность* и *индекс ремонтопригодности*, который инструмент под названием radon[2] может вычислить для кода Python. Хотя эти цифры не слишком надежны, наблюдение за их тенденцией может дать вам некоторое представление о здоровье кодовой базы.

Стиль кода

Когда проект определяет стиль кодирования, он может использовать инструмент, такой как pylint[3] или Flake8[4], чтобы проверить, что код в репозитории действительно соответствует рекомендациям и даже не выполняет сборку, если обнаружены нарушения. Эти два инструмента поставляются с набором правил по умолчанию, но могут быть настроены в соответствии с вашими собственными правилами.

Проверка архитектурных ограничений

Если проект следует четко определенной архитектуре, для кода могут существовать правила, которые можно проверить программно. Например, закрытая трехуровневая система, состоящая из пользовательского интерфейса, бизнес-логики и серверной части хранилища, может иметь следующие правила:

- пользовательский интерфейс не может использовать серверную часть хранилища напрямую, только бизнес-логику;
- серверная часть хранилища может не использовать пользовательский интерфейс напрямую.

Если эти слои обрабатываются как модули в коде Python, можно написать небольшой скрипт, который анализирует операторы импорта во всех исходных файлах и проверяет, не нарушают ли они

---

1 https://pytest-cov.readthedocs.io/en/latest/.

2 https://radon.readthedocs.io/en/latest/intro.html.

3 https://pypi.org/project/pylint/.

4 http://flake8.pycqa.org/en/latest/.

эти правила. Статический анализатор импорта, такой как snakefood[1], может сделать это проще.

Такой инструмент должен привести к сбою шага CI при обнаружении нарушения. Это позволяет отследить, действительно ли идеи архитектуры реализованы в коде, и предотвратить постепенное ослабление базовых принципов архитектуры.

3.6. Резюме

Jenkins – это сервер CI, который автоматически запускает наборы тестов для вас, обычно для каждой новой фиксации в исходном репозитории. Это дает объективное представление о состоянии вашего набора тестов, возможно, на нескольких версиях Python или платформах.

Как только есть это представление, у вас может быть процесс, при котором набор тестов всегда проходит, и можно получить значение из набора тестов.

Когда зрелая команда хорошо работает с процессом CI, вы можете ввести в процесс CI другие показатели, такие как покрытие кода или соблюдение архитектурных правил.

[1] http://furius.ca/snakefood/.

Глава 4

Непрерывная доставка

Непрерывная интеграция (CI) является краеугольным камнем надежной, современной разработки программного обеспечения, но она не является вершиной методологии разработки программного обеспечения. Скорее, это средство для более продвинутых методов.

Когда задание CI показывает, что все тесты пройдены, вы можете быть уверены, что программное обеспечение работает само по себе. Но хорошо ли это работает с другим программным обеспечением? Как мы проверим это перед конечными пользователями? В такой момент и приходит непрерывная доставка (CD).

Практикуя CD, вы автоматизируете процесс развертывания программного обеспечения и повторяете его в нескольких средах. Вы можете использовать некоторые из этих сред для автоматических тестов, таких как тесты полной системной интеграции, автоматические тесты проверки приемлемости и даже тесты производительности и проникновения. Конечно, это не исключает ручного тестирования, которое все еще может обнаружить класс дефектов, которые автоматические тесты обычно не обнаруживают. Наконец, вы используете ту же автоматизацию для развертывания программного обеспечения в производственной среде, где оно достигает своих конечных пользователей.

Настройка системы CD, безусловно, звучит как непростая задача, и это может быть так. Преимущества, однако, многочисленны, но, возможно, не все из них очевидны сразу.

В оставшейся части этой главы рассматриваются преимущества CD и дается приблизительная схема ее реализации. Остальная часть книги посвящена показу простых подходов к CD и примерам, которые ее реализуют.

4.1. Причины для CD и автоматизированных развертываний

Поскольку реализация CD может быть большой работой, хорошо прояснить причины и потенциальные преимущества этого. Вы также можете использовать аргументы, приведенные в этом разделе, чтобы убедить свое руководство инвестировать в данный подход.

Экономия времени

В средних и крупных организациях приложения и их инфраструктура обычно разрабатываются и управляются отдельными группами. Каждое развертывание должно быть согласовано между этими командами. Запрос на изменение должен быть подан, должна быть найдена дата, которая подходит обеим командам, информация о новой версии должна быть распространена (например, какая новая конфигурация доступна или требуется), команда разработчиков должна сделать бинарные файлы доступными для установки и т. д. Все это может легко потреблять часы или дни времени для каждого выпуска, как в группе разработчиков, так и в операционной группе.

Тогда сам процесс развертывания также требует времени, часто сопровождаемого простоями. А поскольку нередко приходится избегать простоев в рабочее время, развертывание должно происходить ночью или в выходные дни, что делает операционную группу менее стремящейся выполнить задачу.

Автоматизация развертываний может сэкономить много времени и репутации. Например, Etsy[1] ввел непрерывную (и, следовательно, автоматизированную) доставку, сократив затраты на развертывание с 6–14 часов для «армии развертывания» до 15-минутных усилий для одного человека[2].

Сокращение цикла релиза

Это правда, что задачи, требующие больших усилий, выполняются гораздо реже, чем те, которые практически не требуют усилий. То

---

[1] www.etsy.com/.

[2] Майк Британ (Mike Britain). Принципы и практика непрерывного развертывания в Etsy // SlideShare. URL: www.slideshare.net/mikebrittain/principles-and-practices- in-continuous-deployment-at-etsy. 2 апр. 2014.

же самое относится и к рискованным начинаниям: мы часто стараемся избегать их.

Компании, которые выполняют ручные релизы и развертывания, часто делают релизы еженедельно или даже реже. Некоторые делают ежемесячные или даже ежеквартальные релизы. В более консервативных отраслях нередки случаи выпуска релизов каждые шесть или двенадцать месяцев.

Редкие релизы неизменно приводят к затяжным процессам разработки и медленному выходу на рынок. Если программное обеспечение развертывается один раз в квартал, время от спецификации до развертывания может легко зависеть от медленного цикла выпуска, по крайней мере для небольших функций.

Это может означать, например, что онлайн-бизнес с плохим пользовательским опытом в процессе оформления заказа должен ждать около трех месяцев, чтобы улучшить пользовательский опыт, что может стоить реальных денег. Автоматизация развертывания облегчает частый релиз, избегая подобных проблем.

Сокращение цикла обратной связи

Если вы разрабатываете инструменты для внутреннего использования в компании, можете попросить нескольких человек опробовать их в промежуточной среде, но это не так просто. Это занимает время от их реальной работы; промежуточная среда должна быть настроена со всеми необходимыми данными (данные клиента, окружение и т. д.), чтобы их можно было использовать; и тогда все изменения будут потеряны. По моему опыту, заставить пользователей тестировать в непроизводственной среде – это тяжелая работа, и она стоит серьезных изменений.

В случае ручных и, следовательно, нечастых выпусков цикл обратной связи медленный, что противоречит самой идее «гибкого», или «бережливого», процесса разработки.

Разработка «бережливого» программного обеспечения – это парадигма разработки, основанная на экономичном производственном процессе Toyota, который направлен на сокращение ненужной работы, быстрое предоставление программного обеспечения, обучение и связанные с ним принципы.

Поскольку человеческое общение подвержено недоразумениям, первая реализация функции редко соответствует первоначальным ожиданиям. Циклы обратной связи неизбежны. Таким образом,

медленные циклы выпуска приводят к медленной разработке, расстраивая как заинтересованных лиц, так и разработчиков.

Но есть и побочные эффекты. Когда циклы улучшений занимают много времени, многие пользователи даже не удосуживаются запросить небольшие улучшения вообще. Это печально, потому что хороший пользовательский интерфейс состоит из сотен маленьких удобств и острых граней, которые должны быть скруглены. Таким образом, в конечном итоге медленные циклы выпуска приводят к ухудшению удобства использования и качества.

Надежность релизов

Существует замкнутый цикл с ручными выпусками. Они, как правило, встречаются редко, что означает внесение многих изменений в один релиз. Это увеличивает риск того, что что-то пойдет не так. Когда большой выпуск вызывает слишком много проблем, менеджеры и инженеры ищут способы повысить надежность следующего выпуска, добавляя больше шагов проверки, больше процессов.

Но больше процессов означает больше усилий, а больше усилий приводит к еще более медленным циклам, что приводит к еще большему количеству изменений на выпуск. Вы можете видеть, куда это идет.

Автоматизация этапов процесса релиза или даже всего процесса является способом разорвать этот порочный круг. Компьютеры намного лучше людей следуют инструкциям к письму, и их концентрация не снижается в конце долгой ночи развертывания программного обеспечения.

Как только процесс релиза стал более надежным и более быстрым, можно легко запускать более частые релизы, каждый из которых вносит меньше изменений. Время, сэкономленное автоматизацией, освобождает ресурсы для дальнейшего улучшения процесса автоматического релиза.

С большим количеством развертываний также приходит больше опыта, что позволяет вам еще больше улучшить процесс и инструменты.

Меньшие приращения облегчают торговлю

Когда при развертывании возникает ошибка и в нем присутствует только одна или две функции, или исправления ошибок, обычно

довольно легко определить, какое изменение вызвало ошибку (сортировка). Напротив, когда многие изменения являются частью одного и того же развертывания, гораздо сложнее сортировать новые ошибки, что означает больше затрат времени, но это также приводит к более длительному времени до устранения дефектов.

Больше архитектурной свободы

Современные тенденции в индустрии программного обеспечения переходят от огромных монолитных приложений к распределенным системам, состоящим из более крупных и более мелких компонентов. Вот что такое *микросервисный* паттерн. Небольшие приложения или службы, как правило, легче обслуживать, а необходимость к масштабируемости требует, чтобы каждое из них могло работать на разных компьютерах, а зачастую на нескольких машинах для каждой службы.

Но если развертывание одного приложения или службы уже является проблемой, то развертывание десяти или даже сотен приложений меньшего размера обещает быть намного большей болью и делает совершенно безответственным смешивание микросервисов с развертыванием вручную.

Таким образом, автоматическое развертывание открывает пространство возможных архитектур программного обеспечения, которые вы можете использовать для решения бизнес-задач.

Передовые методы обеспечения качества

Если у вас есть необходимая инфраструктура, можете применять удивительные стратегии для обеспечения качества. Например, GitHub использует параллельное выполнение новой и старой реализаций в реальном времени[1], чтобы избежать регрессии как по параметрам результата, так и по производительности.

Представьте, что вы разрабатываете поисковую систему для путешествий и хотите улучшить алгоритм поиска. Вы можете развернуть как старую, так и новую версию движка одновременно, выполнить входящие запросы (или их часть) к обоим и определить некоторые метрики, по которым их оценивать.

---

[1] *Винсент Марти* (Vicent Martí). Двигайся быстро и исправляй вещи // GitHub Engineering. URL: http://githubengineering.com/move-fast/. 15 дек. 2015.

Например, быстрое путешествие и низкие цены обеспечивают хорошую стыковку рейсов. Вы можете использовать это, чтобы найти случаи, когда новый движок работает хуже, чем старый, и использовать эти данные для его улучшения. Также можете использовать эти данные, чтобы продемонстрировать превосходство новой поисковой системы, тем самым оправдав усилия, потраченные на ее разработку.

Но такие эксперименты не практичны, если каждая новая версия должна быть развернута вручную, и развертывание каждой версии является серьезным усилием. Автоматическое развертывание не дает вам этих преимуществ автоматически, но является обязательным условием для использования таких передовых методов обеспечения качества.

4.2. План для CD

Я надеюсь, что к настоящему моменту вы уверены, что CD – хорошая идея. Когда я добрался до этой стадии, перспектива ее реализации казалась довольно пугающей.

Процесс CD может быть разбит на несколько этапов, каждый из которых управляется самостоятельно. Более того, автоматизация каждого шага дает преимущества, даже если весь процесс еще не автоматизирован.

Давайте посмотрим на типичную систему CD и соответствующие шаги.

Архитектура конвейера

Система CD структурирована как конвейер. Новый коммит или ветвь в системе управления версиями запускает создание конвейера и начинает выполнение первого из серии этапов. Когда этап проходит успешно, он запускает следующий этап. В случае сбоя весь экземпляр конвейера останавливается.

Затем необходимо ручное вмешательство, обычно путем добавления нового коммита, который исправляет код или тесты; или путем исправления среды либо конфигурации конвейера. Новый экземпляр конвейера или повторное выполнение неудачной стадии может иметь шанс на успех.

Возможны отклонения от строгой модели конвейера. Например, ветки, потенциально выполняемые параллельно, позволяют запускать разные тесты в разных средах и ожидать следующего

шага, пока оба не будут успешно завершены. Разветвление на несколько конвейеров и, следовательно, параллельное выполнение называются *разветвлением*; объединение конвейеров в одну ветвь называется *веером* (рис. 4.1).

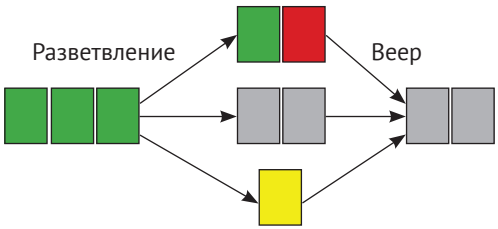


Рис. 4.1 ❖ Разветвить ответвления конвейера; *fan in* присоединяется к ним

Типичные этапы – это сборка, запуск модульных тестов, развертывание в первой тестовой среде, проведение в ней интеграционных тестов, потенциальное развертывание и тестирование в различных тестовых средах и, наконец, развертывание в производство (рис. 4.2).

Рис. 4.2 ❖ Типичные рекомендуемые этапы для конвейера развертывания

Иногда эти этапы немного размываются. Например, типичная сборка пакетов Debian также запускает модульные тесты, что устраняет необходимость в отдельном этапе модульного тестирования. Аналогично, если развертывание в среде запускает дымовые тесты для каждого хоста, на котором она развертывается, нет необходимости в отдельном этапе тестирования дымом (рис. 4.3).

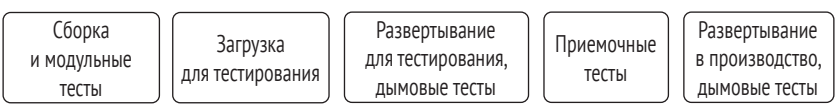


Рис. 4.3 ❖ В реальном конвейере может быть удобно объединить несколько рекомендуемых этапов в один и, возможно, иметь дополнительные этапы, которые теория замалчивает

Как правило, существует часть программного обеспечения, которая контролирует поток всего конвейера. Она подготавливает необходимые файлы для этапа, запускает код, связанный с этапом, собирает его выходные данные и *артефакты* (то есть файлы, которые создает этап и которые стоит сохранить, такие как двоичные файлы или результаты теста), определяет, был ли этап успешен, а затем переходит к следующему этапу.

С архитектурной точки зрения это освобождает этапы от необходимости знать, какой этап наступает дальше, и даже как добраться до машины, на которой работает. Он разъединяет этапы и поддерживает разделение интересов.

Антишаблон: отдельные сборки для каждой среды

Если вы используете в качестве исходного кода модель ветки, такую как GitFlow[1], заманчиво автоматически развернуть ветку разработки в среде тестирования. Когда наступает время для выпуска, вы объединяете ветвь разработки в основную ветвь (возможно, через косвенные ветки отдельных выпусков), а затем автоматически строите основную ветвь и развертываете результат в производственной среде.

Это заманчиво, потому что это прямое расширение существующего, проверенного рабочего процесса. *Не делайте этого!*

Большая проблема данного подхода – вы на самом деле не тестируете то, что будет развернуто, и, с другой стороны, развертываете что-то непроверенное в производство. Даже если есть промежуточная среда перед развертыванием в рабочей среде, вы лишаете законной силы все проведенное тестирование, если на самом деле не поставляете бинарный файл или пакет, который тестировали в предыдущих средах.

Если вы создаете «тестирующие» и «выпускающие» пакеты из разных источников (например, из разных веток), результирующие бинарные файлы будут отличаться. Даже если вы используете один и тот же источник, сборка дважды все равно является плохой идеей, потому что многие сборки не воспроизводимы. Недетерминированное поведение компилятора и различия в средах и зависимостях – все это может привести к тому, что пакеты работали

---

[1] *Винцент Дреззен* (Vincent Driessen). Успешная ветвящаяся Git-модель // nvie.com. URL: http://nvie.com/posts/a-successful-git-branching-model/. 5 янв. 2010.

нормально в одной сборке и не работали в другой. Лучше избегать таких потенциальных различий и ошибок, развертывая на производстве точно такую же сборку, которую вы тестировали в средах тестирования.

Различия в поведении между средами, где они желательны, реализуются с помощью конфигурации, которая не является частью сборки. Также должно быть самоочевидно, что конфигурация должна находиться под контролем версий и развертываться автоматически. Существуют инструменты, специализирующиеся на развертывании конфигурации, такие как Puppet, Chef и Ansible, а в последующих главах обсуждается, как интегрировать их в процесс развертывания.

Все зависит от формата упаковки

Создание развертываемого артефакта – это ранняя стадия в конвейере CD: сборка, управление хранилищем, установка и эксплуатация зависят от выбора формата пакета. Программное обеспечение Python обычно упаковывается в виде исходного архива, в формате, определяемом пакетом `setuptools`, а иногда и в виде пакета двоичного *диска*, указанного в предложении по улучшению Python (PEP) 427[1].

Ни исходные архивы, ни колеса не подходят для развертывания работающего приложения. Им не хватает хуков во время установки для создания необходимых системных ресурсов (таких как учетные записи пользователей), для запуска или перезапуска приложений и других задач, специфичных для операционной системы. У них также нет поддержки управления не-Python-зависимостями, такими как клиентские библиотеки базы данных, написанные на C.

Пакеты Python устанавливаются менеджером пакетов *pip*, который по умолчанию является общесистемной глобальной установкой, которая иногда плохо взаимодействует с пакетами Python, установленными менеджером пакетов операционной системы. Обходные пути существуют, например, в форме виртуальной среды, но управление ими требует дополнительной осторожности и усилий.

Наконец, в случае отдельных обязанностей по разработке и эксплуатации операционная группа обычно намного лучше знакома

[1] Python Software Foundation. PEP 427 – формат бинарного пакета Wheel 1.0 // www.python.org/dev/peps/pep-0427/. 2018.

с собственными пакетами операционной системы. Тем не менее исходные архивы играют очень полезную роль в качестве отправной точки для создания пакетов в форматах, которые больше подходят для прямого развертывания.

В данной книге мы развертываем на компьютерах Debian GNU/Linux и собираем пакеты Debian, используя двухэтапный процесс. Во-первых, создаем архив с исходным кодом, используя файл `setup.py` на основе `setuptools`. Затем инструмент `dh- virtualenv` создает пакет Debian, содержащий virtualenv, в который установлено программное обеспечение и все его зависимости Python.

Технология управления репозиториями Debian

Развертывание пакетов Debian (и большинства других) работает путем загрузки их в *репозиторий*. Затем целевые машины настраиваются с помощью URL-адреса этого хранилища. С точки зрения целевых машин это модель на основе извлечения, позволяющая им выбирать зависимости, которые еще не установлены. Эти репозитории состоят из определенного макета каталога, в котором файлы с заранее заданными именами и форматами содержат метаданные и ссылки на фактические файлы пакета.

Эти файлы и каталоги могут быть доступны с помощью транспортных механизмов, таких как локальный доступ к файлам (и, возможно, подключение через сетевую файловую систему), HTTP и FTP. HTTP – хороший выбор, потому что он прост в настройке, прост в отладке и обычно не является узким местом для производительности, так как это стандартный системный компонент.

Для управления репозиториями Debian существует различное программное обеспечение, большая часть которого плохо документирована или почти не поддерживается. Некоторые решения, такие как *debarchiver* или *dak*, предлагают удаленную загрузку через SSH, но не дают немедленную обратную связь относительно того, была ли загрузка успешной. Debarchiver также обрабатывает загруженные файлы пакетами, запускаемыми заданием `cron`, что приводит к задержке, которая делает автоматизацию намного менее увлекательной.

Я остановился на Aptly[1] – наборе инструментов командной строки для управления репозиториями. Когда вы добавляете новый пакет в репозиторий, Aptly дает немедленную обратную связь в виде

[1] www.aptly.info/.

кода выхода. Он не обеспечивает удобный способ загрузки файлов на сервер, на котором находятся репозитории, но это то, что может сделать менеджер конвейера.

Наконец, Aptly может хранить несколько версий одного и того же пакета в репозитории, что значительно упрощает откат к предыдущей версии.

Инструменты для установки пакетов

После того как вы собрали пакет Debian, загрузили его в репозиторий и настроили целевой компьютер для использования этого хранилища, интерактивная установка пакета выглядит следующим образом:

```
$ apt-get update && apt-get install $package
```

Есть некоторые тонкости, которые следует учитывать в автоматизированных установках.

Вы должны отключить все формы интерактивности, возможно, контролировать многословность вывода, настроить приемлемость понижения и т. д.

Вместо того чтобы пытаться выяснить все эти детали, хорошей идеей будет повторно использовать существующий инструмент, авторы которого уже проделали тяжелую работу. Инструменты управления конфигурацией, такие как Ansible[1], Chef[2], Puppet[3], Salt[4] и Rex[5], имеют модули для установки пакетов, поэтому они могут быть хорошим выбором.

Однако не все системы управления конфигурацией подходят для автоматизации развертываний. Puppet обычно используется в модели на основе запросов, в которой каждый управляемый Puppet компьютер периодически связывается с сервером и запрашивает его целевую конфигурацию. Это прекрасно для масштабируемости, но делает интеграцию в рабочий процесс серьезной проблемой. Модели на основе `push`, в которых менеджер связывается с управляемой машиной, например через SSH, а затем выпол-

---

1 https://ansible.com.

2 www.chef.io/.

3 https://puppet.com/.

4 www.saltstack.com/.

5 www.rexify.org/.

няет команду, гораздо лучше подходят для задач развертывания (и, как правило, предлагают более простой и приятный опыт разработки и отладки).

Для этой книги я выбрал Ansible. Это в основном потому, что мне нравится его декларативный синтаксис, его простая модель и то, что небольшое прибегание к гуглу нашло хорошие решения всех практических проблем.

Управление конвейером

Даже если вы думаете о конвейере развертывания с точки зрения создания, тестирования, распространения и установки программного обеспечения, большая часть проделанной работы на самом деле является «клеем», то есть небольшими задачами, которые делают все это гладко. К ним относятся опрос системы управления версиями, подготовка каталогов для заданий по сборке, сборка собранных пакетов (или прерывание текущего экземпляра конвейера при сбое) и распределение работы по компьютерам, наиболее подходящим для данной задачи.

Конечно, есть инструменты и для этих задач. Обычный CI и серверы сборки, такие как Jenkins, чаще всего могут выполнять эту работу. Но есть и инструменты, специализирующиеся на конвейерах CD, такие как Go непрерывная доставка (GoCD)[1] и Concourse[2].

В то время как Jenkins – отличный инструмент CI, его ориентированное на работу мировоззрение делает его менее оптимальным для конвейерной модели CD. Здесь мы рассмотрим GoCD, программное обеспечение с открытым исходным кодом, разработанное ThoughtWorks, Inc. Оно написано в основном на Java и доступно для большинства операционных систем. Удобно для среды разработки на основе Debian, оно предлагает готовые пакеты Debian.

В примерах следующих глав мы упаковываем сборку, которая также запускает модульные тесты. В производственных условиях вы, скорее всего, включите действие после сборки в конвейер Jenkins, который использует GoCD API для запуска шагов CD, если все тесты в Jenkins пройдены.

---

[1] www.gocd.org/.

[2] https://concourse-ci.org/.

4.3. Резюме

CD позволяет развертывать программное обеспечение небольшими порциями. Это сокращает время выхода на рынок, сокращает количество циклов обратной связи и упрощает сортировку новых ошибок.

Шаги, включенные в CD, содержат модульное тестирование, сборку пакетов, распространение пакетов, установку и тестирование установленных пакетов. Он управляется системой конвейеров, для которой мы будем использовать GoCD.

Глава 5

Сборка пакетов

Сначала мы изучим основы создания tar-архивов с исходным кодом Python, а затем рассмотрим создание пакетов Debian из этих архивов.

5.1. Создание tar-архива с исходным кодом

Чтобы создать архив с исходным кодом Python, вы должны написать сценарий `setup.py`, где используются утилиты `distutils` или `setuptools`. Затем команда сборки `python setup.py sdist` создаст архив в нужном формате.

`distutils` является частью стандартной библиотеки Python, но в ней отсутствует ряд некоторых часто используемых функций. `setuptools` добавляет эти функции путем расширения `distutils`. Какой из этих инструментов вы используете, в основном зависит от вкуса и контекста.

Ниже приводится довольно минимальный файл `setup.py`, где используется `setuptools` для примера с `webcount` из второй главы.

```
from setuptools import setup

setup(
    name = "webcount",
    version = "0.1",
    packages=['webcount', 'test'],
    install_requires=['requests'],
)
```

Здесь мы импортируем функцию `setup` из `setuptools` и вызываем ее с метаданными о пакете – именем, версией, списком включаемых пакетов Python и списком зависимостей.

В документации по setuptools1 перечислены другие аргументы, которые можно передать в функцию setup. Среди наиболее часто используемых:

- Author: применяется для обозначения имени лица, отвечающего за поддержку. author_email – для адреса электронной почты контакта;
- url: ссылка на сайт проекта;
- package_data: используется для добавления в tar-архив файлов, не являющихся файлами Python;
- description: описание назначения пакета, состоящее из одного абзаца;
- python_requires: используется для указания того, какие версии Python поддерживает ваш пакет;
- scripts: может содержать список файлов Python, которые устанавливаются как исполняемые сценарии, а не только пакеты Python.

Когда файл setup.py будет на месте, можно выполнить команду python setup.py sdist, в результате чего в каталоге dist будет создан архив. Файл называется name в setup.py, за которым следует дефис, номер версии, а затем суффикс .tar.gz. В нашем примере это dist/webcount-.1.tar.gz.

5.2. Упаковка с помощью dh-virtualenv

Официальные репозитории Debian поставляются с более чем 40 000 пакетов и включают в себя программное обеспечение, написанное на всех распространенных языках программирования. Для поддержки такого масштаба и разнообразия был разработан инструментарий, чтобы упростить начало работы с процессом упаковки. Он также поддерживает множество приемов настройки.

Данный инструментарий, который в основном находится в пакете devscripts, читает файлы из каталога debian в поисках метаданных и инструкций по сборке.

Хотя полное описание пакета debhelper – довольно большая тема, по которой можно было написать отдельную книгу, здесь я хочу дать достаточно информации, чтобы вы знали, с чего начать.

---

[1] https://setuptools.readthedocs.io/en/latest/setuptools.html#basic-use.

Начало работы с упаковкой

Пакет dh-make предоставляет утилиту для создания каталога debian с уже заполненными метаданными и примерами файлов, которые могут послужить основой для ваших собственных версий. Затем остальные инструменты используют файлы внутри пакетов debian для создания двоичного архива из вашего исходного кода.

Если вы следуете этому примеру в собственной среде разработки, убедитесь, что у вас установлен пакет dh-make, прежде чем продолжить.

Отправной точкой для разработчика Debian обычно является tar-архив с исходным кодом, выпущенный другим проектом, который сообщество Debian называет *апстримом*. В проекте из предыдущей главы мы являемся нашим собственным апстримом и используем репозиторий Git вместо tar-архива, поэтому должны дать dh_make указание создать собственный «оригинальный» архив:

```
$ dh_make --packageclass=s -yes --createorig \
     -p python-webcount_0.1
Maintainer Name      : Moritz Lenz
Email-Address        : moritz@unknown
Date                 : Tue, 04 Sep 2018 15:04:35 +0200
Package Name         : python-webcount
Version              : 0.1
License              : blank
Package Type         : single
Currently there is not top level Makefile. This may require additional tuning
Done. Please edit the files in the debian/subdirectory now.
```

5.3 Файл debian/control

В файле debian/control есть метаданные об исходном пакете и, возможно, несколько двоичных пакетов, собранных из этого исходного пакета. После небольших правок проект python-webcount выглядит так, как показано в листинге 5.1.

Листинг 5.1 ❖ Файл debian/control: метаданные пакета Debian

```
Source: python-webcount

Section: unknown
Priority: optional
Maintainer: Moritz Lenz <moritz@unknown>
Build-Depends: debhelper (>= 10), dh-virtualenv
Standards-Version: 4.1.2
```

```
Package: python-webcount
Architecture: any
Depends: python3
Description: Count occurrences of words in a web page
```

Здесь мы объявляем зависимость сборки `dh-virtualenv`, которую вам нужно установить, чтобы собрать пакет Debian.

Направление процесса сборки

Те, кто отвечает за поддержку в Debian, используют команду `dpkg-buildpackage` или `debuild` для сборки пакета Debian. Помимо прочего, эти инструменты вызывают сценарий `debian/rules` с текущим действием в качестве аргумента. К этим действиям можно отнести `configure`, `build`, `test` или `install`.

Как правило, `debian/rules` – это файл makefile со знаком %, который вызывает `dh`, debhelper. Минимальный сценарий `debian/rules` выглядит так:

```
#!/usr/bin/make -f
%:
        dh $@
```

Мы должны расширить его, чтобы вызвать `dh-virtualenv` и указать ему использовать Python 3 в качестве основы для установки.

```
%:
        dh $@ --with python-virtualenv

override_dh_virtualenv:
        dh_virtualenv --python=/usr/bin/python3
```

Поскольку это файл makefile, отступ здесь должен представлять собой реальные символы табуляции, а не ряд пробелов.

Объявление зависимостей Python

`dh-virtualenv` ожидает файл с именем `require.txt`, в котором перечислены зависимости Python, каждая в отдельной строке (листинг 5.2).

Листинг 5.2 ❖ Файл `requirements.txt`

```
flask
pytest
gunicorn
```

Эти строки будут переданы `pip` в командной строке, поэтому указание номеров версий работает так же, как в pip, например `pytest == 3.8.0`. Можно использовать такую строку, как

```
--index-url=https://...
```

чтобы указать URL-адрес для вашего собственного зеркала pypi, которое dh-virtualenv затем использует для получения пакетов.

Сборка пакета

Как только эти файлы будут на месте, можно запустить сборку с помощью этой команды:

```
$ dpkg-buildpackage -b -us -uc
```

Опция -b дает команде dpkg-buildpackage указание собирать только двоичный пакет (который мы хотим развернуть), а опции -us и -uc пропускают процесс подписи, который разработчики Debian используют для загрузки своих пакетов в зеркала Debian.

Эту команду нужно вызывать в корневом каталоге проекта (каталоге, где содержится каталог debian), и в случае успеха он помещает сгенерированный файл .deb в родительский каталог корневого каталога.

Создание пакета python-matheval

Упаковка matheval в качестве пакета Debian python-matheval аналогична webcount. Основное отличие состоит в том, что matheval – это служба, которая должна работать постоянно.

Мы используем systemd[1], систему инициализации, применяемую Debian, Ubuntu и многими другими дистрибутивами Linux, чтобы контролировать сервисный процесс. Это делается путем написания *модульного файла*, который хранится как debian/python-matheval.service.

```
[Unit]
Description=Evaluates mathematical expressions
Requires=network.target
After=network.target

[Service]
Type=simple
SyslogIdentifier=python-matheval
User=nobody
ExecStart=/usr/share/python-custom/python-matheval/bin/\
gunicorn --bind 0.0.0.0:8800 matheval.frontend:app
```

[1] https://en.wikipedia.org/wiki/Systemd.

```
PrivateTmp=yes
InaccessibleDirectories=/home
ReadOnlyDirectories=/bin /sbin /usr /lib /etc

[Install]
WantedBy=multi-user.target
```

Управление модульными файлами systemd – стандартная задача для пакетов Debian, поэтому существует вспомогательная утилита, которая делает это за нас: `dh-systemd`. Мы должны установить ее и объявить в качестве зависимости сборки в файле `control` (листинг 5.3).

Листинг 5.3 ❖ Файл `debian/control` для пакета `python-matheval`

```
Source: python-matheval
Section: main
Priority: optional
Maintainer: Moritz Lenz <moritz.lenz@gmail.com>
Build-Depends: debhelper (>=9), dh-virtualenv,
               dh-systemd, python-setuptools
Standards-Version: 3.9.6

Package: python-matheval
Architecture: any
Depends: python3 (>= 3.4)
Description: Web service that evaluates math expressions.
```

Файлу `debian/rules` также требуется аргумент `--with systemd`.

```
#!/usr/bin/make -f
export DH_VIRTUALENV_INSTALL_ROOT=/usr/share/python-custom

%:
        dh $@ --with python-virtualenv --with system

override_dh_virtualenv:
        dh_virtualenv --python=/usr/bin/python3 --setuptools-test
```

Знакомый вызов `dpkg-buildpackage` создает пакет Debian, который при установке автоматически запускает веб-сервис и перезапускает его, когда устанавливается новая версия пакета.

Компромиссы dh-virtualenv

Утилита `dh-virtualenv` упрощает создание пакетов Debian со всеми зависимостями Python, упакованными в них. Это очень удобно для разработчика, потому что это означает, что он/она может начать использовать пакеты Python без необходимости создавать из них отдельные пакеты Debian.

Это также означает, что вы можете зависеть от разных версий пакетов Python в нескольких приложениях, установленных на одном компьютере, – это сложно сделать, если вы используете общесистемные пакеты Python.

С другой стороны, такая «жирная упаковка» означает, что если один из пакетов Python содержит брешь в системе безопасности или другую критическую ошибку, вы должны повторно собрать и развернуть все пакеты Debian, которые содержат копию ошибочного кода.

Наконец, пакеты `dh-virtualenv` связаны с версией Python, которая использовалась на сервере сборки. Поэтому если, например, пакет был создан для Python 3.5, он не будет работать с Python 3.6. Если вы переходите с одной версии Python на другую, нужно создавать пакеты для обеих версий параллельно.

5.4. Резюме

Мы создаем пакеты в два этапа: первый этап – tar-архив с исходным кодом на базе `setuptools`, второй этап – бинарный пакет Debian с помощью `dh-virtualenv`.

В обоих шагах используется несколько файлов, в основном на базе декларативного синтаксиса.

Конечным результатом является в основном автономный пакет Debian, которому просто нужна соответствующая версия Python, установленная на целевом компьютере.

Глава 6

Распространение пакетов Debian

Как только пакет Debian будет собран, он должен быть распространен на серверы, где должен быть установлен. Debian, как в основном и все другие операционные системы, использует для этого pull-модель. Пакет и его метаданные хранятся на сервере, с которым клиент может связываться и запрашивать метаданные и пакет.

Сумма метаданных и пакетов называется *репозиторием*. Для того чтобы распространять пакеты на серверы, которым они необходимы, нужно настроить и поддерживать такой репозиторий.

6.1. Сигнатуры

В Debian пакеты подписываются криптографически, чтобы гарантировать, что они не изменяются на сервере репозитория или во время передачи.

Таким образом, первым шагом является создание пары ключей, которая используется для подписи этого конкретного хранилища. (Если у вас уже есть ключ PGP для подписи пакетов, можете пропустить этот шаг.)

Приведенные далее манипуляции предполагают, что вы работаете с неискушенным системным пользователем, не имеющим набора ключей GnuPG, который будет использоваться для поддержки репозитория Debian. Также предполагается, что установлен пакет `gnupg` версии 2 или более поздней.

Для начала создайте файл с именем `key-control-file-gpg2` следующего содержания:

```
%no-protection
Key-Type: RSA
Key-Length: 1024
Subkey-Type: RSA
Name-Real: Aptly Signing Key
Name-Email: nobody@example.com
Expire-Date: 0
%commit
%echo done
```

Замените nobody@example.com на свой адрес электронной почты или адрес электронной почты проекта, в котором вы работаете, а затем выполните следующую команду:

```
$ gpg --gen-key --batch key-control-file-gpg2
```

Вывод команды содержит строку, подобную этой:

```
gpg: key D163C61A6C25A6B7 marked as ultimately trusted
```

Строка из шестнадцатеричных цифр D163C... – это идентификатор ключа, и для каждого запуска он разный. Используйте его для экспорта открытого ключа, который нам понадобится позже.

```
$ gpg --export --armor D163C61A6C25A6B7 > pubkey.asc
```

6.2. Подготовка репозитория

Я использую Aptly[1] для создания и управления репозиторием. Это приложение командной строки без серверного компонента.

Чтобы инициализировать репозиторий, для начала мне нужно придумать ему имя. Здесь я назвал его myrepo.

```
$ aptly repo create -distribution=stretch \
     -architectures=amd64,i386,all -component=main myrepo

Local repo [myrepo] successfully added.
You can run 'aptly repo add myrepo ...' to add packages
to repository.

$ aptly publish repo -architectures=amd64,i386,all myrepo
Warning: publishing from empty source, architectures list
should be complete, it can't be changed after publishing
(use -architectures flag)
Loading packages...
```

[1] www.aptly.info.

```
Generating metadata files and linking package files...
Finalizing metadata files...
Signing file 'Release' with gpg, please enter your
passphrase when prompted:
Clearsigning file 'Release' with gpg, please enter your
passphrase when prompted:

Local repo myrepo has been successfully published.
Please set up your webserver to serve directory
'/home/aptly/.aptly/public' with autoindexing.
Now you can add following line to apt sources:
   deb http://your-server/ stretch main
Don't forget to add your GPG key to apt with apt-key.

You can also use `aptly serve` to publish your repositories
over HTTP quickly.
```

Теперь, когда репозиторий создан, можно добавить пакет, выполнив команды

```
$ aptly repo add myrepo python_webcount_0.1-1_all.deb
$ aptly publish update myrepo
```

Тем самым мы обновляем файлы в `.aptly/public`, чтобы они стали действительным репозиторием Debian, включающим в себя недавно добавленный пакет.

6.3. Автоматизация создания репозитория и добавления пакета

Удобно иметь репозитории для использования внутри конвейера развертывания и добавлять пакеты в эти репозитории, созданные с помощью одной команды. Также есть смысл иметь отдельные репозитории для разных сред. Следовательно, нам нужен репозиторий для тестирования, обкатки и промышленной эксплуатации. Второе измерение – это дистрибутив, для которого создан пакет.

Вот небольшая программа (листинг 6.1), которая, учитывая среду, дистрибутив и список имен файлов пакетов Debian, создает репозиторий в пути `$HOME/aptly/$environment/$distribution`, добавляет пакеты, а затем обновляет общедоступные файлы репозиториев:

Листинг 6.1 ❖ `add-package`, инструмент для создания и заполнения репозиториев Debian

```
#!/usr/bin/env python3
import json
```

```
import os
import os.path
import subprocess
import sys

assert len(sys.argv) >= 4, \
    'Usage: add-package <env> <distribution> <.deb-file>+'

env, distribution = sys.argv[1:3]
packages = sys.argv[3:]

base_path = os.path.expanduser('~') + '/aptly'
repo_path = '/'.join((base_path, env, distribution))
config_file = '{}/{}-{}.conf'.format(base_path, env,
                                     distribution)

def run_aptly(*args):
    aptly_cmd = ['aptly', '-config=' + config_file]
    subprocess.call(aptly_cmd + list(args))

def init_config():
    os.makedirs(base_path, exist_ok=True)
    contents = {
        'rootDir': repo_path,
        'architectures': ['amd64', 'all'],
    }
    with open(config_file, 'w') as conf:
        json.dump(contents, conf)

def init_repo():
    if os.path.exists(repo_path + '/db'):
       return
    os.makedirs(repo_path, exist_ok=True)
    run_aptly('repo', 'create',
        '-distribution=' + distribution, 'myrepo')
    run_aptly('publish', 'repo', 'myrepo')

def add_packages():
    for pkg in packages:
        run_aptly('repo', 'add', 'myrepo', pkg)
    run_aptly('publish', 'update', distribution)

if __name__ == '__main__':
    init_config();
    init_repo();
    add_packages();
```

Его можно использовать как

```
$ ./add-package testing stretch python-matheval_0.1-1_all.deb
```

чтобы добавить файл python-matheval_0.1-1_all.deb в репозиторий Stretch для тестирования среды, и он автоматически создаст этот репозиторий, если тот еще не существует.

6.4. Обслуживание репозиториев

Как таковые репозитории доступны только на одном компьютере. Самый простой способ сделать их доступными для большего количества машин – использовать общедоступный каталог в виде статических файлов по протоколу HTTP.

Если вы используете в качестве веб-сервера Apache, конфигурация виртуального хоста для обслуживания этих файлов может выглядеть так, как показано в листинге 6.2.

Листинг 6.2 ❖ Настройка Apache 2 для обслуживания репозиториев Debian

```
ServerName apt.example.com
ServerAdmin moritz@example.com

DocumentRoot /home/aptly/aptly/
Alias /debian/testing/stretch/      \
    /home/aptly/aptly/testing/stretch/public/
Alias /debian/production/stretch/ \
    /home/aptly/aptly/production/stretch/public/
# Здесь идут дополнительные репозитории.

Options +Indexes +FollowSymLinks
Require all granted

LogLevel notice
CustomLog /var/log/apache2/apt/access.log combined
ErrorLog /var/log/apache2/apt/error.log
ServerSignature On
```

После создания каталога журналов (mkdir -p/var/log/apache2/apt/), активации виртуального хоста (a2ensite apt.conf) и перезапуска Apache репозиторий Debian готов.

Если вместо этого вы предпочитаете веб-сервер lighttpd[1], можно использовать фрагмент кода настройки, как показано в листинге 6.3.

Листинг 6.3 ❖ Настройка lighttpd для обслуживания репозиториев Debian

```
dir-listing.encoding = "utf-8"
server.dir-listing = "enable"
```

[1] www.lighttpd.net.

```
alias.url = (
    "/debian/testing/stretch/"    =>
        "/home/aptly/aptly/testing/stretch/public/",
    "/debian/production/stretch/" =>
        "/home/aptly/aptly/production/stretch/public/",
    # Здесь идут дополнительные репозитории.

)
```

Настройка компьютера для использования репозитория

Когда компьютер использует один из новых репозиториев, он для начала должен доверять криптографическому ключу, которым подписаны репозитории.

Скопируйте открытый ключ PGP (`pubkey.asc`) на компьютер, который будет использовать репозиторий, и импортируйте его.

```
$ apt-key add pubkey.asc
```

Затем добавьте фактический исходный код пакета.

```
$ echo "deb http://apt.example.com/ stretch main" \
    > /etc/apt/source.list.d/myrepo.list
```

После `apt-get update` доступно содержимое репозитория, а `apt-cache policy python-matheval` показывает хранилище как возможный исходник этого пакета.

```
$ apt-cache policy python-webcount
python-webcount:
  Installed: (none)
  Candidate: 0.1-1
  Version table:
*** 0.1-1 0
      990 http://apt.example.com/ stretch/main amd64 Packages
      100 /var/lib/dpkg/status
```

На этом наша экскурсия по управлению репозиториями Debian, а следовательно, и распространению пакетов подошла к концу.

6.5. Резюме

Установщики пакетов Debian, такие как `apt-get` и `aptitude` из комплекта программ APT, считывают метаданные и скачивают пакеты из репозиториев. Программное обеспечение, подобное Aptly, управляет этими репозиториями.

Криптографические сигнатуры проверяют подлинность пакетов, перехватывают атаки посредника и транспортные ошибки, которые изменяют пакеты программного обеспечения.

Вы должны создать ключ GPG, предоставить его Aptly и настроить целевые компьютеры таким образом, чтобы они доверяли этому ключу.

Глава 7

Развертывание пакетов

В предыдущих главах вы видели, как пакеты Debian собираются, помещаются в репозиторий и как можно настроить этот репозиторий в качестве исходного кода пакета на целевом компьютере. Учитывая эти приготовления, установка реального пакета в интерактивном режиме становится легкой.

Чтобы установить в качестве примера проект `python-matheval`, выполните команды

```
$ apt-get update
$ apt-get install python-matheval
```

на целевой машине.

Если для предоставления услуги требуется несколько компьютеров, может быть полезно скоординировать обновление, например обновив только один или два хоста за раз или проведя небольшой интеграционный тест на каждом из них после перехода к следующему. Для этого прекрасно подходит Ansible[1] – система автоматизации и управления конфигурациями с открытым исходным кодом.

7.1. Ansible: основы

Ansible – очень прагматичная и мощная система управления конфигурациями, с которой легко начать работу. Если вы уже знакомы с Ansible (или решили использовать другую систему

[1] https://www.ansible.com.

управления конфигурациями и развертыванием), можете смело пропустить этот раздел.

Соединения и файл инвентаризации

Обычно Ansible используется для подключения к одному или нескольким удаленным компьютерам через протокол Secure Shell (SSH) и перевода их в нужное состояние. Способ подключения является подключаемым. Другие методы включают в себя `local`, который просто вызывает вместо этого команды на локальном компьютере, и `docker`, который подключается через демон Docker для настройки работающего контейнера. Ansible называет эти удаленные компьютеры *хостами*.

Чтобы сообщить Ansible, где и как подключиться, пишется файл *инвентаризации* или файл *хостов*. В этом файле можно определить хосты и группы хостов, а также установить переменные, которые управляют подключением к ним (листинг 7.1).

Листинг 7.1 ❖ Файл `myinventory`

```
# Пример файла инвентаризации.
[all:vars]
# Установленные здесь переменные применяются ко всем хостам.
ansible_ssh_user=root

[web]
# Группа веб-серверов.
www01.example.com
www02.example.com

[app]
# Группа из 5 серверов приложений, которые следуют одной и той же схеме
именования:
app[01:05].example.com

[frontend:children]
# Группа, объединяющая две предыдущие группы.
app
web

[database]
# Здесь мы переопределяем ansible_ssh_user только для одного хоста.
db01.example.com ansible_ssh_user=postgres
```

См. введение к файлам инвентаризации для получения дополнительной информации[1].

[1] https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html.

Чтобы проверить соединение, можно использовать модуль `ping` в командной строке.

```
$ ansible -i myinventory web -m ping
www01.example.com | success >> {
    "changed": false,
    "ping": "pong"
}

www02.example.com | success >> {
    "changed": false,
    "ping": "pong"
}
```

Давайте разберем командную строку на компоненты. `-i myinventory` говорит Ansible использовать файл `myinventory` в качестве файла инвентаризации. `web` говорит Ansible, на каких хостах работать. Это может быть группа, как в этом примере, один хост или несколько таких вещей, разделенные двоеточием. Например, `www01.example.com:database` выберет один из веб-серверов и все серверы базы данных.

Наконец, `-m ping` сообщает Ansible, какой модуль выполнять. `ping` – вероятно, самый простой модуль. Он просто отправляет ответ `"pong"` без внесения каких-либо изменений на удаленный компьютер и в основном используется для отладки файла инвентаризации и учетных данных.

Эти команды выполняются параллельно на разных хостах, поэтому порядок, в котором выводятся ответы, может отличаться. Если проблема возникает при подключении к хосту, добавьте в командную строку опцию `-vvvv`, чтобы получить дополнительный вывод, включая все сообщения об ошибках из SSH.

Ansible неявно дает вам группу `all`, которая – как вы уже догадались – содержит все хосты, настроенные в файле инвентаризации.

Модули

Всякий раз, когда вы хотите что-то сделать на хосте с помощью Ansible, то вызываете для этого модуль. Модули обычно принимают аргументы, которые точно определяют, что должно произойти. В командной строке эти аргументы можно добавить с помощью модуля `ansible -m module -a 'arguments'`. Например:

```
$ ansible -i myinventory database -m shell -a 'echo "hi there"'
db01.example.com | success | rc=0 >>
hi there
```

Ansible поставляется со множеством встроенных модулей, а также экосистемой сторонних модулей. Большинство модулей являются *идемпотентными*. Это означает, что повторное выполнение с теми же аргументами не приводит к изменениям после первого запуска. Например, вместо того чтобы дать Ansible указание создать каталог, вы инструктируете его, чтобы убедиться, что каталог существует. После выполнения такой инструкции в первый раз создается каталог, а повторное выполнение ничего не делает, в то же время сообщая об успехе.

Здесь я хочу познакомить вас лишь с несколькими распространенными модулями.

Модуль shell

Модуль `shell`[1] выполняет команду оболочки на хосте и принимает некоторые параметры, такие как `chdir`, для перехода в другой рабочий каталог перед выполнением команды.

```
$ ansible -i myinventory database -m shell -e 'pwd chdir=/tmp'
db01.example.com | success | rc=0 >>
/tmp
```

Выглядит довольно обычно, но это также и крайний вариант. Если у вас под рукой есть более конкретный модуль для выполнения поставленной задачи, следует отдать предпочтение ему. Например, можно было бы гарантировать, что существуют системные пользователи, используя модуль `shell`, однако для этой цели гораздо проще применить более специализированный пользовательский модуль[2], и он, вероятно, справится лучше, нежели импровизированный сценарий оболочки.

Модуль copy

С помощью модуля `copy`[3] можно дословно копировать файлы с локального компьютера на удаленный.

```
$ ansible -i myinventory database -m copy \
     -a 'src=README.md dest=/etc/motd mode=644 db01.example.com'
     | success >> {
    "changed": true,
```

---

[1] https://docs.ansible.com/ansible/latest/modules/shell_module.html.

[2] https://docs.ansible.com/ansible/latest/modules/user_module.html.

[3] https://docs.ansible.com/ansible/latest/modules/copy_module.html.

```
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "d41d8cd98f00b204e9800998ecf8427e",
    "mode": "0644",
    "owner": "root",
    "size": 0,
    "state": "file",
    "uid": 0
}
```

Модуль template

Модуль `template`[1] в основном работает так же, как и модуль `copy`, но он интерпретирует исходный файл как шаблон Jinja2[2], перед передачей его на удаленный хост. Обычно он используется для создания файлов конфигурации и включения информации из переменных (подробнее об этом позже).

Шаблоны нельзя применять непосредственно из командной строки. Их используют в плейбуках. Вот пример простого плейбука:

```
# file motd.j2
This machine is managed by {{team}}.
```

```
# file template-example.yml
---
- hosts: all
  vars:
    team: Slackers
  tasks:
   - template: src=motd.j2 dest=/etc/motd mode=0644
```

Подробнее о плейбуках мы поговорим позже, но здесь видно, что мы определяем переменную `team`, устанавливаем для нее значение `Slackers`, а шаблон интерполирует ее.

Запуская плейбук с помощью этого кода

```
$ ansible-playbook -i myinventory \
    --limit database template-example.yml
```

мы создаем файл `/etc/motd` на сервере базы данных с содержимым

```
This machine is managed by Slackers.
```

---

[1] https://docs.ansible.com/ansible/latest/modules/template_module.html.

[2] https://jinja.palletsprojects.com/en/master/.

Модуль file

Модуль `file`[1] управляет атрибутами имен файлов, такими как права доступа, а также позволяет создавать каталоги и гибкие и жесткие ссылки.

```
$ ansible -i myinventory database -m file \
    -a 'path=/etc/apt/sources.list.d
            state=directory mode=0755'

db01.example.com | success >> {
    "changed": false,
    "gid": 0,
    "group": "root",
    "mode": "0755",
    "owner": "root",
    "path": "/etc/apt/sources.list.d",
    "size": 4096,
    "state": "directory",
    "uid": 0
}
```

Модуль apt

В Debian и производных дистрибутивах, таких как Ubuntu, установка и удаление пакетов обычно выполняются с помощью менеджеров пакетов из семейства `apt`, например `apt-get`, `aptitude`, а в более новых версиях непосредственно из двоичного файла `apt`.

Модуль `apt`[2] управляет этим изнутри Ansible.

```
$ ansible -i myinventory database -m apt \
    -a 'name=screen state=present update_cache=yes'

db01.example.com | success >> {
    "changed": false
}
```

Здесь пакет `screen` уже был установлен, поэтому модуль не изменил состояние системы.

Для управления apt-ключами[3] доступны отдельные модули (с помощью которых репозитории подвергаются криптографической проверке) и для управления самими репозиториями[4].

---

1 https://docs.ansible.com/ansible/latest/modules/file_module.html.

2 https://docs.ansible.com/ansible/latest/modules/apt_module.html.

3 https://docs.ansible.com/ansible/latest/modules/apt_key_module.html.

4 https://docs.ansible.com/ansible/latest/modules/apt_repository_module.html.

Модули yum и zypper

Для дистрибутивов Linux на базе RPM доступны модули `yum`[1] и `zypper`[2] (на момент написания в состоянии предварительного просмотра). Они управляют установкой пакетов через одноименные менеджеры пакетов.

Модуль package

Модуль `package`[3] использует любой обнаруженный менеджер пакетов. Таким образом, он более универсален по сравнению с модулями `apt` и `yum`, но поддерживает гораздо меньше функций. Например, в случае с `apt` он не предоставляет никакого контроля над тем, запускать ли `apt-get update`, прежде чем делать что-либо еще.

Специализированные модули

Представленные на данный момент модули довольно близки к системе, но также есть модули для решения общих специализированных задач.

В качестве примеров можно упомянуть работу с базами данных[4], понятия, связанные с сетями, например прокси-серверы[5], системы контроля версий[6], кластерные решения, такие как Kubernetes[7] и т. д.

Плейбуки

Плейбуки могут содержать несколько вызовов модулей в определенном порядке и ограничивать их выполнение отдельными хостами или группой хостов. Они записаны в формате YAML[8], фор-

---

1 https://docs.ansible.com/ansible/latest/modules/yum_module.html.

2 https://docs.ansible.com/ansible/latest/modules/zypper_module.html.

3 https://docs.ansible.com/ansible/latest/modules/package_module.html.

4 https://docs.ansible.com/ansible/latest/modules/list_of_database_modules.html.

5 https://docs.ansible.com/ansible/latest/modules/list_of_network_modules.html.

6 https://docs.ansible.com/ansible/latest/modules/list_of_source_control_modules.html.

7 https://docs.ansible.com/ansible/latest/modules/kubernetes_module.html.

8 https://yaml.org.

мате сериализации данных, который оптимизирован для удобства чтения человеком.

Ниже приводится пример плейбука (листинг 7.2), где устанавливается новейшая версия пакета Debian `go-agent`, рабочего узла для Go Continuous Delivery (GoCD)[1].

Листинг 7.2 ❖ Плейбук Ansible для установки агента GoCD в системе на базе Debian

```
---
 - hosts: go-agent
   vars:
     go_server: go-server.example.com
   tasks:
- apt: package=apt-transport-https state=present
- apt_key:
     url: https://download.gocd.org/GOCD-GPG-KEY.asc
     state: present
     validate_certs: no
- apt_repository:
     repo: 'deb https://download.gocd.org /'
     state: present
- apt: update_cache=yes package={{item}} state=present
  with_items:
   - go-agent
   - git
   - build-essential
- lineinfile:
     dest: /etc/default/go-agent
     regexp: ^GO_SERVER=
     line: GO_SERVER={{ go_server }}
- copy:
    src: files/guid.txt
    dest: /var/lib/go-agent/config/guid.txt
    user: go
    group: go
- service: name=go-agent enabled=yes state=started
```

Элемент верхнего уровня в этом файле представляет собой одноэлементный список. Единственный элемент начинается с `hosts: go-agent`, который ограничивает выполнение хостами в группе `go-agent`. Это соответствующая часть файла инвентаризации, которая идет с ним:

[1] https://www.gocd.org.

```
[go-agent]
go-worker01.p6c.org
go-worker02.p6c.org
```

Затем он устанавливает для переменной `go_server` в качестве значения строку, здесь это имя хоста, где работает сервер GoCD.

Наконец, самое главное в плейбуке: список задач для выполнения. Каждая задача – это вызов модуля, некоторые из которых мы уже обсуждали.

Ниже приведен краткий обзор.

- Сначала `apt` устанавливает пакет Debian `apt-transport-https`, чтобы убедиться, что система может извлекать метаданные и файлы из репозиториев Debian через протокол HTTPS.
- В следующих двух задачах используются модули `apt_repository`[1] и `apt_key`[2] для настройки хранилища, из которого будет установлен фактический пакет `go-agent`.
- В ходе еще одного вызова `apt` устанавливается желаемый пакет. Кроме того, еще несколько пакетов устанавливаются с помощью конструкции цикла[3].
- Модуль `lineinfile`[4] выполняет поиск строки в текстовом файле по регулярному выражению и заменяет найденную строку предопределенным содержимым. Здесь мы используем это для настройки сервера GoCD, к которому подключается агент.
- И наконец, модуль `service`[5] запускает агента, если тот еще не запущен (`state=started`), и обеспечивает его автоматический запуск при перезагрузке (`enabled=yes`).

Плейбуки вызываются с помощью команды `ansible-playbook`, например `ansible-playbook -i inventory go-agent.yml`.

В плейбуке может быть несколько списков задач. Это распространенный случай, когда они затрагивают разные группы хостов.

```
---
- hosts: go-agent:go-server
  tasks:
    - apt: package=apt-transport-https state=present
```

---

[1] https://docs.ansible.com/ansible/latest/modules/apt_repository_module.html.

[2] https://docs.ansible.com/ansible/latest/modules/apt_key_module.html.

[3] https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html.

[4] https://docs.ansible.com/ansible/latest/modules/lineinfile_module.html.

[5] https://docs.ansible.com/ansible/latest/modules/service_module.html.

```
    - apt_key:
        url: https://download.gocd.org/GOCD-GPG-KEY.asc
        state: present
        validate_certs: no
   - apt_repository:
        repo: 'deb https://download.gocd.org /'
        state: present

- hosts: go-agent
  tasks:
    - apt: update_cache=yes package={{item}} state=present
      with_items:
       - go-agent
       - git
       - build-essential
    - ...

- hosts: go-server
  tasks:
    - apt: update_cache=yes package={{item}} state=present
    - apt: update_cache=yes package=go-server state=present
    - ...
```

Переменные

Переменные полезны как для управления потоком внутри плейбука, так и для заполнения мест в шаблонах для генерации файлов конфигурации. Есть несколько способов установки переменных. Один из способов – установить их непосредственно в плейбуках, используя `vars: ...`, как мы уже видели ранее. Другой – указать их в командной строке.

```
ansible-playbook --extra-vars=variable=value theplaybook.yml
```

Третий, очень гибкий способ – использовать свойство `group_vars`. Для каждой группы, в которой находится хост, Ansible ищет файл `group_vars/thegroup.yml` и файлы, соответствующие `group_vars/thegroup/*.yml`. Хост может находиться в нескольких группах одновременно, что дает вам дополнительную гибкость.

Например, можно поместить каждый хост в две группы: одна для роли, которую выполняет хост (например, веб-сервер, сервер базы данных, DNS-сервер и т. д.), и другая для окружения, в котором он находится (тестирование, обкатка, производственная эксплуатация). Вот небольшой пример, в котором используется этот макет.

```
# environments
[prod]
www[01:02].example.com
db01.example.com

[test]
db01.test.example.com
www01.test.example.com
# functional roles

[web]
www[01:02].example.com
www01.test.example.com

[db]
db01.example.com
db01.test.example.com
```

Чтобы настроить только тестовые хосты, можно запустить

```
ansible-playbook --limit test theplaybook.yml
```

и поместить переменные, относящиеся к окружению, в файлы `group_vars/test.yml` и `group_vars/prod.yml`, а переменные, специфичные для веб-сервера, в `group_vars/web.yml` и т. д.

Можно использовать вложенные структуры данных в своих переменных, а если вы это сделаете, можете настроить Ansible для объединения этих структур данных, если они указаны в нескольких источниках. Это можно выполнить, создав файл `ansible.cfg` с таким содержимым:

```
[defaults]
hash_behavior=merge
```

Таким образом, у вас может быть файл `group_vars/all.yml`, который устанавливает значения по умолчанию:

```
# file group_vars/all.yml
myapp:
    domain: example.com
    db:
        host: db.example.com
        username: myappuser
        instance. myapp
```

а затем можете переопределить отдельные элементы этой вложенной структуры данных, например в `group_vars/test.yml`, следующим образом:

```
# file group_vars/test.yml
myapp:
    domain: test.example.com
    db:
        hostname: db.test.example.com
```

Ключи, которые не были затронуты в файле `group_vars/test.yml`, например `myapp.db.username`, наследуются от файла `all.yml`.

Роли

Роли – это способ инкапсулировать части плейбука в многократно используемый компонент. Давайте рассмотрим реальный пример, который приводит к простому определению роли.

Для развертывания программного обеспечения обычно требуется развернуть только что собранную точную версию, поэтому соответствующая часть плейбука выглядит так:

```
- apt:
    name: thepackage={{package_version}}
    state: present
    update_cache: yes
    force: yes
```

Но для этого необходимо указывать переменную `package_version` всякий раз, когда вы запускаете плейбук. Будет непрактично, если вместо того, чтобы запускать развертывание недавно созданного программного обеспечения, вы настраиваете новый компьютер и необходимо установить несколько пакетов программного обеспечения, у каждого из которых свой плейбук.

Поэтому мы обобщаем этот код на случай, если номер версии отсутствует.

```
- apt:
    name: thepackage={{package_version}}
    state: present
    update_cache: yes
    force: yes
  when: package_version is defined
- apt: name=thepackage state=present update_cache=yes
  when: package_version is undefined
```

Если вы включите несколько таких плейбуков в один и запустите их на одном хосте, то, вероятно, заметите, что большая часть времени тратится на выполнение команды `apt-get update` для каждого включенного плейбука.

Обновление кеша apt необходимо в первый раз, потому что, возможно, вы только что загрузили новый пакет на свое локальное зеркало Debian перед развертыванием, но последующие запуски не нужны. Таким образом, можете хранить информацию, которую хост уже обновил для своего кеша, в *факте*[1], который является своего рода переменной для каждого хоста в Ansible.

```
- apt: update_cache=yes
  when: apt_cache_updated is undefined

- set_fact:
    apt_cache_updated: true
```

Как видите, кодовая база для разумной установки пакета немного выросла, и пришло время выделить его в *роль*.

Роли – это коллекции файлов YAML с предопределенными именами. Команды

```
$ mkdir roles
$ cd roles
$ ansible-galaxy init custom_package_installation
```

создают пустой скелет для роли с именем custom_package_installation. Задачи, которые ранее входили во все плейбуки, теперь переходят в файл tasks/main.yml, расположенный ниже основного каталога роли (листинг 7.3).

Листинг 7.3 ❖ Файл roles/custom_package_installation/tasks/main.yml

```
- apt: update_cache=yes
  when: apt_cache_updated is undefined
- set_fact:
    apt_cache_updated: true

- apt:
    name: {{package}={{package_version}}
    state: present
    update_cache: yes
    force: yes
  when: package_version is defined
- apt: name={{package} state=present update_cache=yes
  when: package_version is undefined
```

Чтобы использовать роль, включите ее в плейбук:

```
---
- hosts: web
  pre_tasks:
```

[1] https://docs.ansible.com/ansible/latest/modules/set_fact_module.html.

```
    - # Задачи, выполняемые до роли (ролей).
  roles:
     role: custom_package_installation
     package: python-matheval
  tasks:
    - # Задачи, выполняемые после роли (ролей).
```

pre_tasks и tasks не являются обязательными. Плейбук, состоящий только из включенных ролей, вполне подойдет.

Ansible обладает множеством других функций, таких как обработчики, которые позволяют перезапускать службы только один раз после любых изменений, динамические инвентаризации для более гибких серверных ландшафтов, Vault для шифрования переменных[1] и богатую экосистему существующих ролей для управления распространенными приложениями и связующим программным обеспечением.

Чтобы узнать больше об Ansible, я настоятельно рекомендую отличную книгу Лорина Хохштейна *Up and Running* (2-е изд. O'Reilly Media, 2017 // https://www.amazon.com/dp/1491979801/).

7.2. Развертывание с помощью Ansible

Вооружившись знаниями об Ansible из предыдущего раздела, развертывание становится простой задачей. Начнем с отдельных файлов инвентаризации для сред (листинг 7.4).

Листинг 7.4 ❖ Файл production

```
[web]
www01.yourorg.com
www02.yourorg.com

[database]
db01.yourorg.com

[all:vars]
ansible_ssh_user=root
```

Возможно, среде тестирования требуется только один веб-сервер (листинг 7.5).

Листинг 7.5 ❖ Тестирование файла инвентаризации Ansible

```
[web]
www01.testing.yourorg.com
```

[1] https://docs.ansible.com/ansible/latest/user_guide/playbooks_vault.html.

```
[database]
db01.stagingyourorg.com

[all:vars]
ansible_ssh_user=root
```

Установку пакета `python-matheval` на веб-серверах в среде тестирования теперь можно выразить одной строкой:

```
$ ansible -i testing web -m apt -a 'name=python-matheval
update_cache=yes state=latest'
```

После того как вы начнете развертывание с Ansible, вероятно, вы захотите выполнять с ним и другие задачи по управлению конфигурацией, поэтому имеет смысл написать плейбук для каждого пакета, который хотите развернуть. Вот один из них (листинг 7.6), где используется роль установки пакетов из раздела **Роли**.

Листинг 7.6 ❖ Файл `deploy-python-matheval.yml`: плейбук развертывания для `python-matheval`

```
---
- hosts: web
  roles:
    role: custom_package_installation
    package: python-matheval
```

Затем можно вызвать его как

```
$ ansible-playbook -i testing deploy-python-matheval.yml
```

7.3. Резюме

Ansible может устанавливать пакеты за вас, но также может делать гораздо больше. Он может настраивать операционную систему и приложение и даже управлять процессами на нескольких машинах.

Записывая файл инвентаризации, вы сообщаете Ansible, какими компьютерами он управляет. Плейбуки определяют, что делать, используя модули для выполнения отдельных задач, таких как создание пользователей или установка программного обеспечения.

Глава 8

Виртуальная площадка для автоматизации развертываний

В следующих главах мы рассмотрим инструмент Go continuous delivery (GoCD) и как выполнять упаковку, распространение и развертывание с его помощью. Если вы хотите продолжить и экспериментировать с тем, что описано в этих главах, вам потребуются машины, на которых это можно делать.

Если нет возможности позволить себе такую роскошь, как общедоступное или частное облако, в котором можно запускать виртуальные машины (ВМ) и тестировать примеры, можно использовать инструменты, представленные в этой главе, чтобы создать площадку для виртуальных машин на своем ноутбуке или рабочей станции. Даже если у вас есть доступ к облачному решению, можете использовать некоторые из представленных здесь сценариев для настройки этих компьютеров.

8.1. Требования и использование ресурсов

Что мы хотим делать в виртуальной площадке:

- выполнять сборку пакетов Debian;
- загружать их в локальный репозиторий Debian;
- устанавливать пакеты на одном или нескольких серверах;
- запускать несколько простых веб-сервисов;
- запускать сценарии развертывания и настройки с помощью Ansible;
- контролировать все через сервер и агента GoCD.

Все эти задачи требуют очень мало ресурсов, за исключением последней. Серверу GoCD требуется больше всего ресурсов: минимум 1 ГБ оперативной памяти при 2 ГБ рекомендуемых. Сервер Go также является единственной системой, которая поддерживает постоянное состояние (например, конфигурацию и историю конвейера), которое обычно не хочется терять.

Итак, самый простой подход, который здесь использую, – это установка сервера Go на главном компьютере, представляющем собой ноутбук или рабочую станцию, с которой я обычно работаю.

Затем есть одна виртуальная машина для запуска агента Go, на которой будут собираться пакеты Debian. Еще две служат целевыми машинами, на них будут устанавливаться и тестироваться только что собранные пакеты. Одна из них служит средой тестирования, вторая – для производственной эксплуатации.

Для этих трех виртуальных машин половины гигабайта ОЗУ по умолчанию, которую обеспечивает инструментарий, вполне достаточно. Если вы используете эту площадку и у вас недостаточно оперативной памяти на главном компьютере, можно попытаться уменьшить объем оперативной памяти этих виртуальных машин. Для целевых компьютеров даже 200 МБ может быть достаточно для начала работы.

8.2. Знакомство с Vagrant

Vagrant – это уровень абстракции над классическими решениями для виртуализации, такими как KVM и VirtualBox. Он предлагает базовые образы (именуемые *боксами*) для вашей виртуальной машины, управляет виртуальными машинами за вас и предлагает унифицированный API для начальной конфигурации. Он также создает виртуальную частную сеть, которая позволяет главным компьютерам взаимодействовать с виртуальными машинами, и наоборот.

Чтобы установить Vagrant, можно скачать установщик на странице www.vagrantup.com/downloads.html или, если вы используете операционную систему с менеджером пакетов, например Debian или RedHat, можно установить его через менеджер пакетов. В Debian и Ubuntu его устанавливают с помощью команды `apt-get install vagrant` (избегайте серии 2.0 и установите версию 2.1 или более новые версии с сайта Vagrant, если через менеджер пакетов доступна только версия 2.0).

Вы также должны установить `virtualbox` таким же образом, который действует как бэкенд для Vagrant. После того как установили Vagrant, выполните следующую команду:

```
$ vagrant plugin install vagrant-vbguest
```

После этого будет установлен плагин Vagrant, который автоматически устанавливает *гостевые утилиты* внутри боксов Vagrant, что улучшает конфигурируемость и надежность.

Чтобы использовать Vagrant, вы пишете небольшой сценарий на языке Ruby под названием `Vagrantfile`, который инстанцирует один или несколько боксов в качестве виртуальных машин. Можете настроить проброс портов, частные или мостовые сети и совместное использование каталогов между виртуальными машинами хоста и гостя.

Инструмент командной строки `vagrant` позволяет создавать и подготавливать виртуальные машины с помощью команды `vagrant up`, подключаться к виртуальной машине с помощью команды `vagrant ssh`, получить сводку состояния, используя команду `vagrant status`, а также останавливать и удалять ВМ снова с помощью команды `vagrant destroy`. Вызов `vagrant` без каких-либо аргументов дает сводку доступных опций.

Вам, наверное, интересно, почему мы используем виртуальные машины Vagrant вместо контейнеров Docker. Причина состоит в том, что Docker оптимизирован для запуска одного процесса или группы процессов. Но в нашем варианте использования нам нужно запустить как минимум три процесса: агента GoCD и приложение, которое на самом деле хотим запустить, Aptly, чтобы управлять репозиторием Debian, а также HTTP-сервер, чтобы разрешить удаленный доступ к хранилищу.

Настройка сети и Vagrant

Мы будем использовать Vagrant с виртуальной частной IP-сетью с адресами от 172.28.128.1 до 172.28.128.254. Когда назначаете один или несколько адресов этого диапазона виртуальной машине, Vagrant автоматически назначает главному компьютеру адрес 172.28.128.1.

Я добавил эти строки в свой файл `/etc/hosts`. Это не является строго необходимым, но облегчает общение с виртуальными машинами.

```
# Vagrant
172.28.128.1 go-server.local
```

```
172.28.128.3 testing.local
172.28.128.4 production.local
172.28.128.5 go-agent.local
```

Также добавил несколько строк в свой файл ~/.ssh/config.

```
Host 172.28.128.* *.local
    User root
    StrictHostKeyChecking no
    IdentityFile /dev/null
    LogLevel ERROR
```

Не делайте этого для производственных машин. Это безопасно только в виртуальной сети на одной машине, где вы можете быть уверены, что там нет злоумышленников, только если они уже не скомпрометировали вашу машину.

Создание и уничтожение виртуальных машин распространено в Vagrant, и каждый раз, когда создаете их заново, они получают новые хост-ключи. Без такой конфигурации вы бы потратили много времени на обновление отпечатков SSH-ключей.

Представленные здесь Vagrantfile и плейбук Ansible можно найти в репозитории deploy-utils в GitHub в папке playbook. Воспользуйтесь приведенными ниже командами:

```
$ git clone https://github.com/python-ci-cd/
deployment-utils.git

$ cd deployment-utils/playground

$ vagrant up

$ ansible-playbook setup.yml
```

В листинге 8.1 показан Vagrantfile, который создает боксы для виртуальной площадки:

Листинг 8.1 ❖ Vagrantfile

```
Vagrant.configure(2) do |config|
  config.vm.box = "debian/stretch"

  {
    'testing' => "172.28.128.3",
    'production' => "172.28.128.4",
    'go-agent' => "172.28.128.5",
  }.each do |name, ip|
    config.vm.define name do |instance|
        instance.vm.network "private_network", ip: ip,
            auto_config: false
```

```
        instance.vm.hostname = name + '.local'
    end
  end

  config.vm.provision "shell" do |s|
    ssh_pub_key = File.readlines("#{Dir.home}/.ssh/id_rsa.pub")
        .first.strip
    s.inline = <<-SHELL
      mkdir -p /root/.ssh
      echo #{ssh_pub_key} >> /root/.ssh/authorized_keys
    SHELL
  end
end
```

Этот файл `Vagrantfile` предполагает, что у вас есть пара SSH-ключей, и открытый ключ находится внутри пути `.ssh/id_rsa.pub` ниже вашего домашнего каталога, который является местоположением по умолчанию для ключей RSA SSH в Linux. Он использует поставщика Vagrant `shell` для добавления открытого ключа в файл `authorized_keys` корневого пользователя внутри виртуальных машин, чтобы вы могли осуществлять вход через SSH на гостевых компьютерах. (Vagrant предлагает команду `vagrant ssh` для выполнения подключения без этого дополнительного шага, но я считаю, что проще использовать системную команду `ssh` напрямую, главным образом потому, что она не связана с присутствием `Vagrantfile` внутри текущего рабочего каталога.)

В каталоге с Vagrantfile можно выполнить команду

```
$ vagrant up
```

чтобы запустить и подготовить три виртуальные машины. Когда вы делаете это в первый раз, это занимает несколько минут, потому что Vagrant должен сначала загрузить базовый бокс.

Если все прошло нормально, можно проверить, что все три виртуальные машины работают, вызвав команду `vagrant status`:

```
$ vagrant status
Current machine states:

testing                   running (virtualbox)
production                running (virtualbox)
go-agent                  running (virtualbox)

This environment represents multiple VMs. The VMs are all listed above with
their current state. For more information about a specific VM, run `vagrant
status NAME`.
```

И (в системах Linux на базе Debian) должны увидеть недавно созданную частную сеть.

```
$ ip route | grep vboxnet
172.28.128.0/24 dev vboxnet1 proto kernel scope link
    src 172.28.128.1
```

Теперь можете войти в виртуальные машины с помощью `ssh root@go-agent.local`, используя `testing.local` и `production.local` в качестве имен хостов.

8.3. Настройка машин

Чтобы настроить виртуальные машины, мы начнем с небольшого файла `ansible.cfg` (листинг 8.2).

Листинг 8.2 ❖ ansible.cfg: файл конфигурации для среды

```
[defaults]
host_key_checking = False
inventory = hosts
pipelining=True
```

Отключение проверки ключа хоста следует выполнять только в доверенных виртуальных сетях для систем разработки, но не в производственных условиях.

Виртуальные машины и их IP-адреса перечислены в файле инвентаризации (листинг 8.3).

Листинг 8.3 ❖ Файл инвентаризации hosts

```
[all:vars]
ansible_ssh_user=root

[go-agent]
agent.local ansible_ssh_host=172.28.128.5

[aptly]
go-agent.local

[target]
testing.local ansible_ssh_host=172.28.128.3
production.local ansible_ssh_host=172.28.128.4

[testing]
testing.local

[production]
production.local
```

Затем следует плейбук (листинг 8.4), в котором выполняется вся настройка, необходимая для запуска агента GoCD, репозитория Aptly и доступа по протоколу SSH от виртуальной машины `go-agent` к целевым виртуальным машинам.

Листинг 8.4 ❖ Файл setup.yml: плейбук Ansible для настройки трех виртуальных машин

```
---
 - hosts: go-agent
   vars:
     go_server: 172.28.128.1
   tasks:
   - group: name=go system=yes
   - name: Make sure the go user has an SSH key
     user: >
        name=go system=yes group=go generate_ssh_key=yes
        home=/var/go
   - name: Fetch the ssh public key, so we can distribute it.
     fetch:
        src: /var/go/.ssh/id_rsa.pub
        dest: go-rsa.pub
        fail_on_missing: yes
        flat: yes
   - apt: >
        package=apt-transport-https state=present
        update_cache=yes
   - apt_key:
        url: https://download.gocd.org/GOCD-GPG-KEY.asc
        state: present
        validate_certs: no
   - apt_repository:
        repo: 'deb https://download.gocd.org /'
        state: present
   - apt: package={{item}} state=present force=yes
     with_items:
      - openjdk-8-jre-headless
      - go-agent
      - git
   - file:
       path: /var/lib/go-agent/config
       state: directory
       owner: go
       group: go
   - copy:
       src: files/guid.txt
       dest: /var/lib/go-agent/config/guid.txt
       owner: go
```

```
      group: go
  - name: Go agent configuration for versions 16.8 and above
    lineinfile:
       dest: /etc/default/go-agent
       regexp: ^GO_SERVER_URL=
       line: GO_SERVER_URL=https://{{ go_server }}:8154/go
  - service: name=go-agent enabled=yes state=started

- hosts: aptly
  tasks:
    - apt: package={{item}} state=present
      with_items:
       - ansible
       - aptly
       - build-essential
       - curl
       - devscripts
       - dh-systemd
       - dh-virtualenv
       - gnupg2
       - libjson-perl
       - python-setuptools
       - lighttpd
       - rng-tools
    - copy:
       src: files/key-control-file-gpg2
       dest: /var/go/key-control-file
    - command: killall rngd
      ignore_errors: yes
      changed_when: False
    - command: rngd -r /dev/urandom
      changed_when: False
    - command: gpg --gen-key --batch /var/go/key-control-file
      args:
        creates: /var/go/.gnupg/pubring.gpg
      become_user: go
      become: true
      changed_when: False
    - shell: gpg --export --armor > /var/go/pubring.asc
      args:
        creates: /var/go/pubring.asc
      become_user: go
      become: true
    - fetch:
        src: /var/go/pubring.asc
        dest: deb-key.asc
        fail_on_missing: yes
        flat: yes
```

```
    - name: Bootstrap aptly repos on the `target` machines
      copy:
       src: ../add-package
       dest: /usr/local/bin/add-package
       mode: 0755
    - name: Download an example package to fill the repo with
      get_url:
       url: https://perlgeek.de/static/dummy.deb
       dest: /tmp/dummy.deb
    - command: >
           /usr/local/bin/add-package {{item}}
           stretch /tmp/dummy.deb
      with_items:
        - testing
        - production
      become_user: go
      become: true
    - user: name=www-data groups=go

    - name: Configure lighttpd to serve the aptly directories
      copy:
           src: files/lighttpd.conf
           dest: /etc/lighttpd/conf-enabled/30-aptly.conf
    - service: name=lighttpd state=restarted enabled=yes

- hosts: target
  tasks:
    - authorized_key:
       user: root
       key: "{{ lookup('file', 'go-rsa.pub') }}"
    - apt_key:
           data: "{{ lookup('file', 'deb-key.asc') }}"
           state: present
- hosts: production
  tasks:
    - apt_repository:
        repo: >
           deb http://172.28.128.5/debian/production/stretch
           stretch main
        state: present

- hosts: testing
  tasks:
    - apt_repository:
        repo:
        deb http://172.28.128.5/debian/testing/stretch
           stretch main
        state: present
```

```
- hosts: go-agent
  tasks:
    - name: 'Checking SSH connectivity to {{item}}'
      become: True
      become_user: go
      command: >
         ssh -o StrictHostkeyChecking=No
         root@"{{ hostvars[item]['ansible_ssh_host'] }}" true
      changed_when: false
      with_items:
           - testing.local
           - production.local
```

Тут много чего происходит. Мы:

- устанавливаем и настраиваем агента GoCD:
  - копируем файл с фиксированным UID в каталог конфигурации агента Go, поэтому, когда вы разбираете машину и создаете ее заново, сервер Go идентифицирует ее как того же агента, что и раньше;
- предоставляем пользователю `go` на машине `go-agent` доступ по протоколу SSH на целевых хостах:
  - сначала убедившись, что у пользователя Go есть SSH-ключ;
  - копируя открытый SSH-ключ на главный компьютер;
  - позже распространяя его на целевые машины с помощью модуля `authorized_key`;
- создаем пару ключей GPG для пользователя `go`:
  - поскольку при создании ключей GPG используется много энтропии для случайных чисел, а виртуальные машины обычно не имеют такой энтропии, сначала устанавливаются `rng-tools`, которые используются, чтобы убедить систему использовать низкокачественную хаотичность. Опять же, не стоит этого делать в условиях производственной эксплуатации;
- копируем открытый ключ указанной пары ключей GPG в главный компьютер и распространяем его на целевые машины с помощью модуля `apt_key`;
- создаем репозитории Debian на базе Aptly на машине `go-agent`:
  - копируя сценарий `add-package` из того же репозитория в машину `go-agent`;

  - запуская его с фиктивным пакетом для фактического создания репозиториев;
  - устанавливая и настраивая `lighttpd` для обслуживания этих пакетов по протоколу HTTP;
  - настраивая целевые машины для использования этих репозиториев в качестве исходника пакета;
- проверяем, может ли пользователь Go на машине `go-agent` действительно добраться до других виртуальных машин через SSH.

После запуска плейбука с помощью `ansible-playbook setup.yml` у вас есть агент GoCD, ожидающий подключения к серверу. Установка сервера GoCD описана в следующей главе. После установки сервера GoCD необходимо активировать агента в веб-конфигурации и назначить соответствующие ресурсы (`debian-stretch`, `build` и `aptly`, если вы следуете примерам, приведенным в этой книге).

8.4. Резюме

Vagrant помогает создать виртуальную площадку для непрерывной доставки, управляя виртуальными машинами и частной сетью. Мы видели плейбук Ansible, который настраивает эти машины, чтобы предоставить всю инфраструктуру, необходимую для запуска сервера GoCD на главном компьютере.

Глава 9

Сборка в конвейере с помощью Go Continuous Delivery

В предыдущих главах мы продемонстрировали автоматизацию важных шагов от исходного кода до развертывания: сборка, распространение и развертывание. Чего сейчас не хватает, так это клея, который скрепляет их всех воедино: опрос репозиториев исходного кода, получение пакетов с сервера сборки на сервер репозитория и, как правило, управление потоком, прерывание экземпляра конвейера, когда один из этапов дает сбой, и т. д.

В качестве клея мы будем использовать Go Continuous Delivery[1] (GoCD или Go) от компании ThoughtWorks.

9.1. О Go Continuous Delivery

GoCD – проект с открытым исходным кодом, написанный на языке Java, а компоненты его веб-интерфейса созданы с использованием фреймворка Ruby on Rails. Он был запущен в качестве проприетарного программного обеспечения в 2010 году, а его исходные коды были открыты в 2014 году.

Можно скачать GoCD для Windows, OSX, Debian и дистрибутивов Linux на базе RPM и Solaris. ThoughtWorks предоставляет коммерческую поддержку.

---

[1] https://www.gocd.org.

Он состоит из серверного компонента, который содержит конфигурацию конвейера, опрашивает репозитории с исходным кодом на предмет изменений, планирует и распределяет работу, собирает артефакты, представляет веб-интерфейс для визуализации и управления всем этим и предлагает механизм для ручного утверждения шагов.

Один или несколько *агентов* подключаются к серверу и выполняют фактические задания в конвейере сборки.

Устройство конвейера

Каждая сборка, развертывание или тестовое задание, выполняемое GoCD, должно быть частью *конвейера*. Конвейер состоит из одного или нескольких линейно расположенных *этапов*. Внутри этапа одно или несколько *заданий* могут выполняться параллельно и распределяться индивидуально по агентам. *Задачи* выполняются последовательно в рамках задания.

В задаче вы можете использовать файлы, которые в предыдущих задачах выполнялись в том же задании, тогда как между заданиями и этапами должны явно захватывать, а затем извлекать их как артефакты. Подробнее об этом далее.

Самая общая задача – выполнение внешней программы. Другие задачи включают в себя поиск артефактов или специфичных для языка вещей, таких как запуск сборок Ant или Rake[1].

Конвейеры могут запускать другие конвейеры, позволяя сформировать ациклический ориентированный граф конвейеров (рис. 9.1).

Соответствие заданий агентам

Когда агент простаивает, он опрашивает сервер на предмет наличия работы. Если на сервере есть задания для запуска, используется два критерия, чтобы определить, подходит ли агент для выполнения задания: *среды* и *ресурсы*.

Каждое задание является частью конвейера, и если вы решите использовать среды, конвейер будет частью среды. С другой стороны, каждый агент настроен так, чтобы быть частью одной или

[1] Задачи `<ant>` и `<rake>` выполняют специализированные сборщики с теми же именами и позволяют указывать цели и файлы сборки. См. https://docs.gocd.org/current/configuration/configuration_reference.html#ant для получения дополнительной информации.

нескольких сред. Агент принимает задания только из конвейеров одной из своих сред.

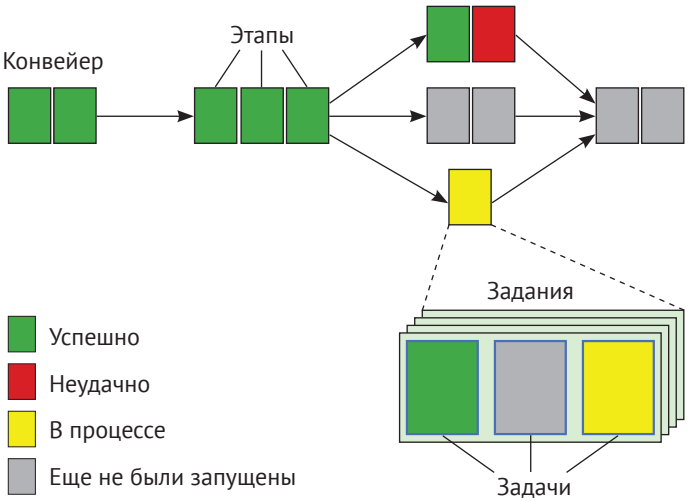


Рис. 9.1 ❖ Конвейеры GoCD могут образовывать граф. Конвейеры состоят из последовательных этапов, в которых несколько заданий могут выполняться параллельно. Задачи выполняются поочередно внутри задания

Ресурсы – это пользовательские метки, которые описывают то, что может предложить агент, а в конфигурации конвейера можно указать, какие ресурсы требуются заданию. Например, если вы определили, что для тестирования веб-приложения заданию требуется ресурс `phantomjs`, это задание будут выполнять только те агенты, которым этот ресурс назначен. Рекомендуется в качестве ресурсов добавлять операционную систему и версию. В предыдущем примере у агента могли быть ресурсы `phantomjs`, `debian` и `debian-stretch`. Тем самым автору задания предлагается выбор степени детализации для указания требуемой операционной системы.

Одно слово по поводу среды

GoCD позволяет запускать агенты в определенных средах. Например, можно запускать агента Go при каждом тестировании и на каждом производственном компьютере и сопоставлять конвейеры с агентскими средами, чтобы гарантировать, что этап установки вы-

полняется на правильном компьютере в правильной среде. Если вы используете эту модель, также можете использовать GoCD для копирования артефактов сборки на машины, для которых они нужны.

Решено не делать этого, потому что не хотелось устанавливать агента GoCD на каждой машине, на которой надо выполнить развертывание. Вместо этого используется Ansible, который выполняется на агенте GoCD, чтобы контролировать все машины в среде. Это требует управления SSH-ключами, которые использует Ansible, и распространения пакетов через репозиторий Debian. Но поскольку Debian в любом случае нужно хранилище, чтобы разрешать зависимости, это не такая уж и дополнительная нагрузка.

Материалы

Материал в GoCD служит двум целям: он запускает конвейер и предоставляет файлы, с которыми могут работать задачи в конвейере.

В качестве материалов использовались репозитории Git, а GoCD может опрашивать эти репозитории, вызывая конвейер, когда доступна новая версия. Агент GoCD также клонирует репозитории в файловую систему, в которой выполняет свои задания.

Существуют плагины для материалов, предназначенные для различных систем контроля версий, таких как Subversion (svn) и mercurial, а также плагины для обработки репозиториев пакетов Debian и RPM в качестве материалов.

Наконец, конвейер может служить материалом для других конвейеров. Используя это свойство, можно создавать графы конвейеров.

Артефакты

GoCD может собирать *артефакты*, которые представляют собой файлы или каталоги, сгенерированные заданием. Более поздние части одного и того же конвейера или даже других подсоединенных конвейеров могут извлекать эти артефакты. Извлечение артефактов не ограничивается артефактами, созданными на одном и том же компьютере агента.

Артефакты также можно получить из веб-интерфейса и REST API, предоставляемого сервером GoCD[1].

[1] https://api.gocd.org/current.

Репозиторий артефактов можно настроить для удаления более старых версий, когда места на диске становится мало.

9.2. Установка

Чтобы использовать GoCD, необходимо установить сервер GoCD на одном компьютере, а агент GoCD – как минимум на одном компьютере. Это можно сделать на том же компьютере, что и сервер, или на другом, при условии что он может подключаться к серверу GoCD с портами 8153 и 8154.

Когда ваша инфраструктура и количество конвейеров растут, вероятно, у вас будет несколько агентов Go.

Установка сервера GoCD в Debian

Чтобы установить сервер GoCD в операционной системе на базе Debian, сначала убедитесь, что вы можете скачивать пакеты Debian через HTTPS.

```
$ apt-get install -y apt-transport-https
```

Затем нужно настроить исходники пакетов.

```
$ echo 'deb https://download.gocd.org /' \
       > /etc/apt/sources.list.d/gocd.list
$ curl https://download.gocd.org/GOCD-GPG-KEY.asc \
       | apt-key add -
```

И наконец, установить его.

```
$ apt-get update && apt-get install -y go-server
```

В Debian 9 кодовое имя *Stretch*, Java 8 доступна из коробки. В более старых версиях Debian вам может потребоваться установить Java 8 из других источников, таких как Debian Backports[1].

Когда вы сейчас указываете свой браузер на порт 8154 сервера Go для HTTPS (игнорируйте предупреждения безопасности SSL) или порт 8153 для HTTP, должны увидеть веб-интерфейс сервера GoCD (рис. 9.2).

Если вы получили сообщение об ошибке `connection refused`, проверьте файлы в `/var/log/go-server/`, чтобы найти подсказки о том, что пошло не так.

---

[1] https://backports.debian.org.

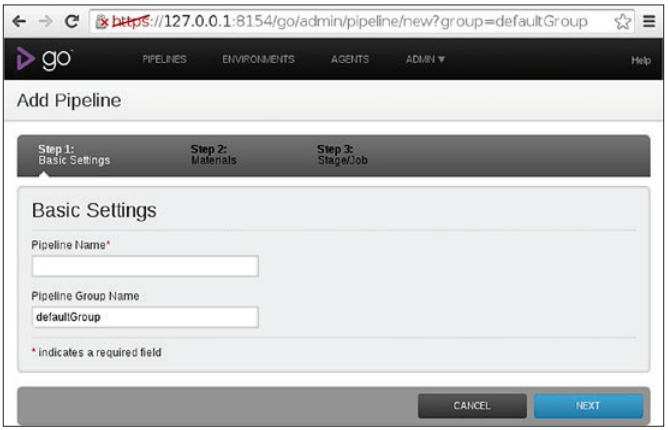


Рис. 9.2. Исходный веб-интерфейс GoCD

Чтобы предотвратить несанкционированный доступ, можно установить плагины аутентификации, например аутентификации на основе файлов паролей[1] или LDAP или аутентификации на базе Active Directory[2].

Установка агента GoCD в Debian

На одной или нескольких машинах, где хотите выполнить шаги автоматической сборки и развертывания, вы должны установить агента Go, который подключится к серверу и опросит его на предмет наличия работы.

В главе 8 приведен пример автоматической установки агента GoCD. Если вы хотите сделать это вручную, то должны выполнить те же первые три шага, что и при установке сервера GoCD, чтобы гарантировать возможность установки пакетов из репозитория пакетов GoCD. Затем, конечно же, вы установите агента Go. В системе на базе Debian это выглядит так:

```
$ apt-get install -y apt-transport-https
$ echo 'deb https://download.gocd.org /' >
    /etc/apt/sources.list.d/gocd.list
$ curl https://download.gocd.org/GOCD-GPG-KEY.asc \
    | apt-key add -
$ apt-get update && apt-get install -y go-agent
```

---

[1] https://github.com/gocd/gocd-filebased-authentication-plugin.

[2] https://github.com/gocd/gocd-ldap-authentication-plugin.

После этого отредактируйте файл `/etd/default/go-agent`. Первая строка должна выглядеть так:

```
GO_SERVER_URL=https://127.0.0.1:8154/go
```

Измените переменную так, чтобы она указывала на ваш компьютер с сервером GoCD, затем запустите агента.

```
$ service go-agent start
```

Через несколько секунд агент свяжется с сервером. Когда вы щелкаете меню **Agents** (Агенты) в веб-интерфейсе сервера GoCD, вы должны увидеть агента (рис. 9.3).

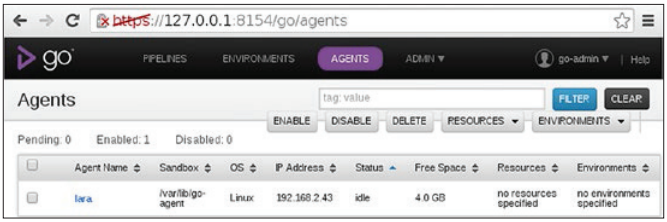


Рис. 9.3. Скриншот интерфейса управления агентами GoCD (здесь `lara` – это имя хоста агента)

Первый контакт с XML-конфигурацией GoCD

Существует два способа настройки сервера GoCD: через веб-интерфейс и через файл конфигурации в формате XML. XML-конфигурацию также можно отредактировать через веб-интерфейс[1].

Хотя веб-интерфейс является хорошим способом изучить возможности GoCD, из-за слишком большого количества нажатий он быстро становится раздражающим в использовании. Использование редактора с хорошей поддержкой XML позволяет сделать все намного быстрее, и он лучше поддается компактному объяснению, поэтому я выбрал этот путь. Вы также можете использовать оба подхода на одном и том же экземпляре сервера GoCD.

---

[1] Начиная с версии 16.7 GoCD конвейерные конфигурации можно переносить во внешние репозитории контроля версий, а с помощью плагинов их даже можно записывать в различных форматах, таких как YAML. Хотя такой подход и кажется очень многообещающим, знакомство с ним выходит за рамки этой книги.

В меню **Admin** пункт **Config XML** позволяет просматривать и редактировать конфигурацию сервера. В листинге 9.1 показано, как выглядит базовая XML-конфигурация с одним зарегистрированным агентом.

Листинг 9.1 ❖ Базовая XML-конфигурация GoCD с одним зарегистрированным агентом

```
<?xml version="1.0" encoding="utf-8"?>
<cruise
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="cruise-config.xsd"
    schemaVersion="77">
<server artifactsdir="artifacts"
        commandRepositoryLocation="default"
        serverId="b2ce4653-b333-4b74-8ee6-8670be479df9">
</server>
<agents>
    <agent hostname="lara" ipaddress="192.168.2.43"
        uuid="19e70088-927f-49cc-980f-2b1002048e09" />
</agents>
</cruise>
```

`ServerId` и данные агента в вашей установке будут другими, даже если вы выполнили те же шаги. Чтобы предоставить агенту ресурсы, можно изменить тег `<agent .../>` в разделе `<agents>` для чтения, как показано в листинге 9.2.

Листинг 9.2 ❖ XML-конфигурация GoCD для агента с ресурсами

```
<agent hostname="lara" ipaddress="192.168.2.43"
    uuid="19e70088-927f-49cc-980f-2b1002048e09">
  <resources>
    <resource>debian-stretch</resource>
    <resource>build</resource>
    <resource>aptly</resource>
  </resources>
</agent>
```

Создание SSH-ключа

Для GoCD удобно иметь SSH-ключ без пароля, например для клонирования репозиториев Git через SSH. Чтобы создать его, выполните приведенные ниже команды на сервере:

```
$ su - go
$ ssh-keygen -t rsa -b 2048 -N '' -f ~/.ssh/id_rsa
```

Скопируйте получившийся каталог .ssh и файлы в нем на каждого агента в каталог /var/go (и не забудьте указать владельца и права доступа, так как они были созданы изначально) или создайте новую пару ключей для каждого агента.

9.3. Сборка в конвейере

Для запуска сборки пакета Debian необходимо извлечь исходный код из репозитория Git, настроив его как материал GoCD, затем вызвать команду dpkg-buildpackage с некими параметрами и, наконец, собрать получившиеся файлы.

Листинг 9.3 – это первый шаг к созданию пакета python-matheval, выраженный в XML-конфигурации GoCD.

Листинг 9.3 ❖ Простой подход к созданию пакета Debian в GoCD

```
<pipelines group="deployment">
  <pipeline name="python-matheval">
    <materials>
      <git
url="https://github.com/python-ci-cd/python-matheval.git"
        dest="source" />
    </materials>
    <stage name="build" cleanWorkingDir="true">
       <jobs>
         <job name="build-deb" timeout="5">
          <tasks>
            <exec command="/bin/bash" workingdir="source">
              <arg>-c</arg>
              <arg>dpkg-buildpackage -b -us -uc</arg>
            </exec>
          </tasks>
          <artifacts>
            <artifact src="*.deb" dest="debian-packages/"
                type="build" />
          </artifacts>
          <resources>
            <resource>debian-stretch</resource>
            <resource>build</resource>
          </resources>
        </job>
      </jobs>
    </stage>
  </pipeline>
</pipelines>
```

Эту и все последующие XML-конфигурации можно найти в каталоге `gocd` репозитория `deploy-utils`[1].

Первый тег – это группа конвейеров, у которой есть имя. Ее можно использовать для классификации доступных конвейеров, а также для управления правами.

Второй уровень – это тег `<pipeline>` с именем. Он содержит список материалов и один или несколько этапов.

Макет каталога

Каждый раз, когда на этапе выполняется задание, агент GoCD, назначенный заданию, подготавливает каталог, в котором он делает материалы доступными. В Linux по умолчанию это каталог `/var/lib/go-agent/pipelines/`, после которого идет имя конвейера. Пути в конфигурации GoCD идут относительно этого пути.

Например, предыдущее определение материала содержит атрибут `dest="source"`, поэтому абсолютный путь к рабочей копии этого репозитория Git – `/var/lib/go-agent/pipelines/python-matheval/source`. Отсутствие `dest="..."` сработает и даст на один уровень каталога меньше, но это также помешает нам использовать второй материал в будущем.

См. справочные материалы по конфигурации[2], где приводится список доступных типов материалов и параметров. Доступны плагины[3], с помощью которых можно добавлять другие типы материалов.

Этапы, задания, задачи и артефакты

Все этапы конвейера осуществляются последовательно, и каждый из них выполняется только в том случае, если предыдущий этап прошел успешно. У каждого этапа есть имя, которое используется как во внешнем интерфейсе, так и для извлечения артефактов, произведенных на этом этапе.

В предыдущем примере этапу был дан атрибут `cleanWorkingDir="true"`, что позволяет GoCD удалять файлы, созданные во время предыдущей сборки, и отменять изменения в файлах под контро-

---

[1] https://github.com/python-ci-cd/deployment-utils.

[2] https://docs.gocd.org/current/configuration/configuration_reference.html#materials.

[3] https://www.gocd.org/plugins/.

лем версий. Это хороший вариант для использования; в противном случае вы можете неосознанно столкнуться с ситуацией, когда предыдущая сборка влияет на текущую, что может быть очень болезненным для отладки.

Задания потенциально выполняются параллельно в рамках этапа, и у них есть имена по тем же причинам, что и у этапов. Задания выполняются параллельно, только если для их запуска доступно несколько агентов.

Агент GoCD последовательно выполняет задачи в задании. Склоняемся в основном использовать задачи `<exec>` (и `<fetchartifact>`, которые вы увидите в следующей главе), которые вызывают системные команды. Они следуют соглашению UNIX, согласно которому код возврата, равный нулю, означает успех, а все остальное – это сбой.

Для более сложных команд создаем сценарии оболочки, Perl или Python внутри репозитория Git и добавляем репозиторий в качестве материала в конвейер, что делает их доступными в процессе сборки, без дополнительных усилий.

Задача `<exec>` в нашем примере вызывает `/bin/bash -c 'dpkg-buildpackage -b -us -uc'`. Это культ карго в программировании[1], потому что вызов команды `dpkg-buildpackage` напрямую работает так же хорошо. А, ладно, можем вернуться к этому позже…

Команда `dpkg-buildpackage -b -us -uc` собирает пакет Debian и выполняется внутри копирования исходного кода. Она производит файлы `.deb`, `.changes` и, возможно, несколько других файлов с метаданными. Они создаются на один уровень выше копирования файлов, в корневом каталоге конвейера.

Поскольку это файлы, с которыми мы хотим работать позже, по крайней мере таковым является файл `.deb`, позволяем GoCD хранить их во внутренней базе данных, именуемой *хранилищем артефактов*. Это то, что тег `<artifact>` в конфигурации поручает сделать GoCD.

Имя сгенерированных файлов пакета зависит от номера версии встроенного пакета Debian (который исходит из файла `debian/changelog` в репозитории Git), поэтому в дальнейшем будет непросто ссылаться на них по имени. Именно здесь в игру вступает `dest="debian-packages/"`: он заставляет GoCD хранить артефакты в каталоге с фиксированным именем.

---

[1] https://en.wikipedia.org/wiki/Cargo_cult_programming.

На более поздних этапах можно получить все файлы артефактов из этого каталога по фиксированному имени каталога.

Конвейер в действии

Если все идет хорошо (а так и должно быть, верно?), вы увидите, как выглядит веб-интерфейс после запуска нового конвейера, который показан на рис. 9.4.

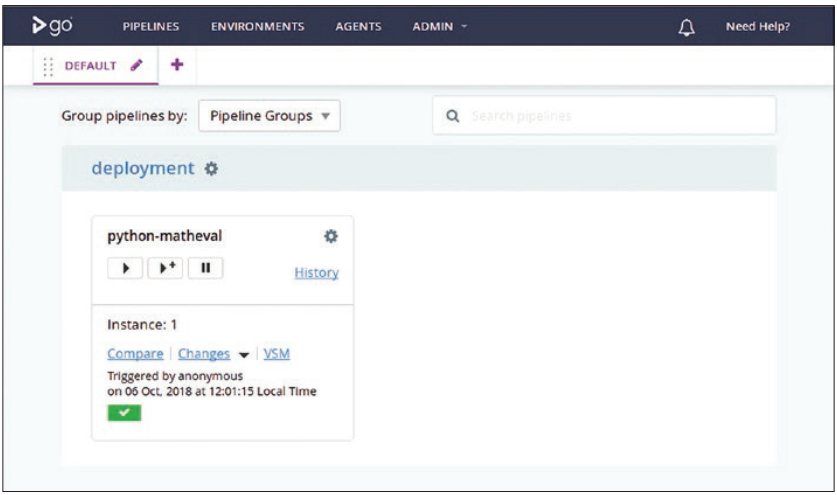


Рис. 9.4 ❖ Обзор конвейера после успешного запуска этапа сборки

Всякий раз, когда в репозитории Git появляется новая фиксация, GoCD с радостью создает пакет Debian и сохраняет его для дальнейшего использования.

Автоматизированные сборки, у-у-у!

Старый номер версии – это не полезно

При сборке пакета Debian инструментарий определяет номер версии полученного пакета, посмотрев в верхнюю часть файла `debian/changelog`. Это означает, что всякий раз, когда кто-то размещает код или изменения в документации без новой записи в журнале изменений, получившийся пакет Debian имеет тот же номер версии, что и предыдущий.

Большинство инструментов Debian предполагают, что кортеж, состоящий из имени, версии и архитектуры пакета, однозначно идентифицирует версию пакета. Заполнение новой версии пакета старым номером версии в хранилище может вызвать проблемы. Большинство программ управления репозиториями просто отказываются принимать копию пакета, который использует старый номер версии. На целевом компьютере, где должен быть установлен пакет, обновление пакета ничего не даст, если номер версии останется прежним.

Создание уникальных номеров версий

Есть несколько источников, которые вы можете нажать, чтобы создать уникальный номер версии:

- случайность (например, в форме UUID);
- текущая дата и время;
- сам Git-репозиторий;
- несколько переменных среды[1], предоставляемых GoCD и которые могут быть полезны.

Последний пункт выглядит перспективно. `GO_PIPELINE_COUNTER` – это монотонный счетчик, который увеличивается каждый раз, когда GoCD запускает конвейер, поэтому это хороший источник для номера версии. GoCD позволяет повторно запускать этапы вручную, поэтому лучше всего объединить его с `GO_STAGE_COUNTER`. С точки зрения сценариев оболочки, использование `$GO_PIPELINE_COUNTER.$GO_STAGE_COUNTER` в качестве строки версии звучит как достойный подход.

Но это еще не все. GoCD позволяет запускать конвейер с определенной версией материала, поэтому можете запустить новый конвейер для создания старой версии программного обеспечения. Если вы сделаете это, использование `GO_PIPELINE_COUNTER` в качестве первой части строки версии не будет отражать использование старой базы кода.

Команда `git describe` – это установленный способ подсчета фиксаций. По умолчанию она выводит последний тег в репозитории, и если `HEAD` не разрешает тот же коммит, что и тег, он добавляет количество коммитов, начиная с этого тега, и сокращенный хэш SHA1 с префиксом `g`, например `2016.04-32-g4232204` для коммита

[1] https://docs.gocd.org/current/faq/dev_use_current_revision_in_build.html.

4232204, то есть 32 фиксации после тега 2016.04. Опция --long заставляет его всегда выводить количество коммитов и хеш, даже когда HEAD указывает на тег.

Нам не нужен хеш фиксации для номера версии, поэтому сценарий оболочки для построения подходящего номера версии выглядит следующим образом:

```
#!/bin/bash

set -e
set -o pipefail
v=$(git describe --long |sed 's/-g[A-Fa-f0-9]*$//')
version="$v.${GO_PIPELINE_COUNTER:-0}.${GO_STAGE_COUNTER:-0}"
```

Синтаксис Bash ${VARIABLE:-default} – хороший способ заставить сценарий работать вне среды агента GoCD. Для этого сценария требуется установить тег в репозитории Git. Если его нет, вы получите следующее сообщение от git describe:

```
fatal: No names found, cannot describe anything.
```

Еще кое-что по поводу сборки

Теперь, когда у нас есть уникальная строка версии, мы должны дать команду системе сборке использовать эту строку. Для этого нужно сделать новую запись в файле debian/changelog с желаемым номером версии. Утилита debchange автоматизирует это для нас. Для надежной работы необходимо несколько опций.

```
export DEBFULLNAME='Go Debian Build Agent'
export DEBEMAIL='go-noreply@example.com'
debchange --newversion=$version --force-distribution -b \
    --distribution="${DISTRIBUTION:-stretch}" 'New Version'
```

Когда мы хотим сослаться на этот номер версии на более поздних этапах конвейера (да, их будет больше), удобно иметь его в файле.

Это также удобно иметь в выводе, поэтому нам нужно еще две строки в сценарии.

```
echo $version
echo $version > ../version
```

И конечно же, должен запускать фактическую сборку:

```
dpkg-buildpackage -b -us -uc
```

Подключение к GoCD

Чтобы сделать сценарий доступным для GoCD, а также для того, чтобы он находился в системе управления версиями, поместим его в репозиторий Git под именем `debian-autobuild` и добавим репозиторий в качестве материала в конвейер (листинг 9.4):

Листинг 9.4 ❖ Конфигурация GoCD для сборки пакетов с разными номерами версий

```
<pipeline name="python-matheval">
  <materials>
    <git
url="https://github.com/python-ci-cd/python-matheval.git"
        dest="source" materialName="python-matheval" />
    <git
url="https://github.com/python-ci-cd/deployment-utils.git"
    dest="deployment-utils" materialName="deployment-utils" />
  </materials>
  <stage name="build" cleanWorkingDir="true">
    <jobs>
      <job name="build-deb" timeout="5">
        <tasks>
          <exec command="../deployment-utils/debian-autobuild"
                workingdir="source" />
        </tasks>
        <artifacts>
          <artifact src="version" type="build"/>
          <artifact src="*.deb" dest="debian-packages/"
            type="build" />
        </artifacts>
        <resources>
          <resource>debian-stretch</resource>
          <resource>build</resource>
        </resources>
      </job>
    </jobs>
  </stage>
</pipeline>
```

Теперь GoCD автоматически создает пакеты Debian при каждой фиксации в репозиторий Git и дает каждому отдельную строку версии.

9.4. Резюме

GoCD – это инструмент с открытым исходным кодом, который может опрашивать ваши Git-репозитории и инициировать сборку через выделенных агентов. Он настраивается через веб-интерфейс, если кликнуть на помощников или предоставить XML-конфигурацию.

Необходимо позаботиться о создании значимых номеров версий для каждой сборки. Git-теги, количество фиксаций с момента последнего тега и счетчики, предоставляемые GoCD, являются полезными компонентами для создания таких номеров.

Глава 10

Распространение и развертывание пакетов в конвейере

В предыдущей главе мы приступили к созданию конвейера GoCD. Он автоматически создает пакет Debian каждый раз, когда в Git отправляются новые коммиты, и генерирует уникальный номер версии для каждой сборки. Наконец, фиксирует в качестве артефактов встроенный пакет и файл с именем `version`, содержащий номер версии. Далее нам нужно загрузить его в репозиторий Debian и развернуть на целевых компьютерах.

10.1. Загрузка в конвейер

В главе 6, посвященной распространению пакетов, вы уже познакомились с небольшой программой для создания и заполнения репозиториев Debian, управляемой с помощью Aptly. Если добавите ее в репозиторий `deploy-utils` из этой главы, то сможете автоматически загружать вновь созданные пакеты с этой дополнительной конфигурацией GoCD (листинг 10.1), которая будет вставлена после этапа сборки.

Листинг 10.1 ❖ Конфигурация GoCD для загрузки недавно созданного пакета в репозиторий `testing`

```
<stage name="upload-testing">
  <jobs>
    <job name="upload-testing">
      <tasks>
```

```
        <fetchartifact pipeline="" stage="build"
            job="build-deb" srcdir="debian-packages"
            artifactOrigin="gocd">
          <runif status="passed" />
        </fetchartifact>
        <exec command="/bin/bash">
          <arg>-c</arg>
  <arg>deployment-utils/add-package testing stretch *.deb</arg>
        </exec>
      </tasks>
      <resources>
        <resource>aptly</resource>
      </resources>
    </job>
  </jobs>
</stage>
```

Задача `fetchartifact` извлекает, как вы уже догадались, артефакт, который хранится в хранилище артефактов сервера GoCD. Здесь он получает каталог `python-matheval`, куда предыдущий этап загрузил пакет Debian. Пустая строка в имени конвейера указывает GoCD использовать текущий конвейер.

В вызове сценария `add-package` слово `testing` относится к названию среды (которую вы можете выбрать свободно, если вы непротиворечивый), а не тестовому дистрибутиву проекта Debian.

Наконец, ресурс `aptly` выбирает агента GoCD с тем же ресурсом для запуска задания (см. рис. 10.1). Если ожидаете, что установка немного вырастет, у вас должна быть отдельная машина для обслуживания этих репозиториев. Установите агента GoCD и назначьте ему этот ресурс. У вас даже могут быть отдельные машины для репозиториев для тестирования и производственных репозиториев, и можете предоставлять им более конкретные ресурсы (например, `aptly-testing` and `aptly-production`).

Учетные записи пользователей и безопасность

В предыдущем примере конфигурации сценарий `add-package` запускается как системный пользователь `go`, чей домашний каталог в системах на базе Linux по умолчанию – это `/var/go`. В результате будут созданы репозитории в каталоге, таком как `/var/go/aptly/testing/stretch/`.

В главе 6 предполагалось, что Aptly работает под собственной учетной записью системного пользователя. Вы по-прежнему долж-

ны предоставить пользователю go права для добавления пакетов в репозиторий, но можете запретить ему изменять существующие репозитории и, что более важно, получать доступ к ключу GPG, которым подписаны пакеты.

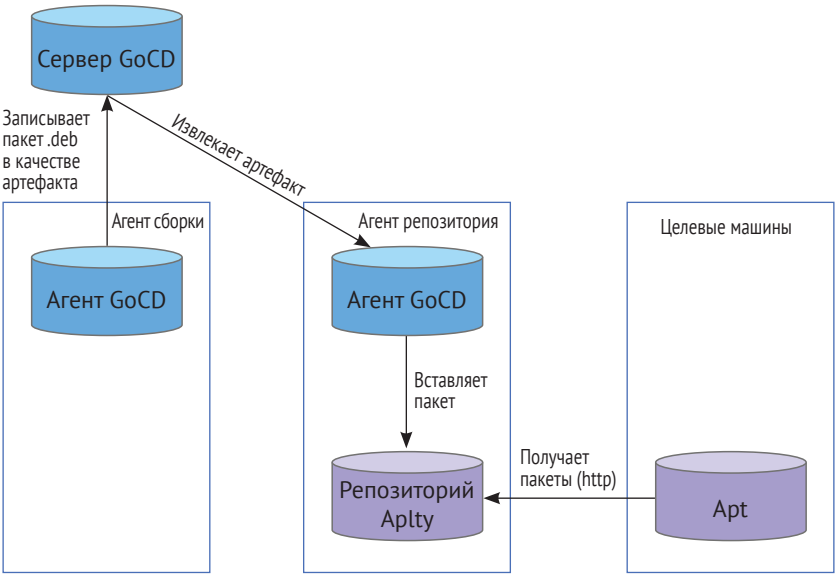


Рис. 10.1 ❖ На машине, где находится репозиторий Aptly, есть агент GoCD, который получает пакеты Debian в виде артефактов с сервера GoCD. Целевые машины настраивают хранилище как исходник пакета

Если репозиторий находится под отдельным пользователем, вам нужен способ преодолеть барьер учетной записи пользователя, и традиционный способ сделать это в случае с приложениями командной строки – разрешить пользователю go вызывать add-package с помощью команды sudo. Но чтобы получить реальную выгоду для безопасности, нужно скопировать команду add-package туда, где у пользователя go нет прав на запись. В противном случае злоумышленник, имеющий доступ к учетной записи пользователя go, может просто изменить эту команду, чтобы делать все, что посчитает нужным.

Предполагая, что вы намереваетесь скопировать его в /usr/local/bin, можно добавить строку

`/etc/sudoers`

в файл (листинг 10.2).

Листинг 10.2 ❖ Строка `/etc/sudoers`, позволяющая пользователю `go` выполнить команду `add-package` в качестве `aptly`

```
go ALL=(aptly) NOPASSWD: /usr/local/bin/add-package
```

Затем вместо вызова `add-package <environment> <distribution> <deb package>` вы меняете его на

```
$ sudo -u aptly --set-home /usr/local/bin/add-package \
    <environment> <distribution> <deb package>
```

Флаги `--set-home` дают команде `sudo` указание установить переменную окружения `HOME` в домашний каталог целевого пользователя, в данном случае `aptly`.

Если вы решили не идти по пути `sudo`, то придется адаптировать конфигурацию веб-сервера для обслуживания файлов из `/var/go/aptly/` вместо `/home/aptly/aptly`.

10.2. Развертывание в конвейере

В главе 7 мы увидели, как обновить (или установить, если он еще не установлен) пакет с помощью Ansible (см. рис. 10.2):

```
$ ansible -i testing web -m apt \
    -a 'name=python-matheval state=latest update_cache=yes'
```

где `testing` – это файл инвентаризации с тем же именем, что и у окружения, `web` – это группа хостов для развертывания, а `python-matheval` – имя пакета.

В GoCD это можно сделать как отдельный этап, после этапа `upload-testing` (листинг 10.3).

Листинг 10.3 ❖ Конфигурация GoCD для автоматической установки пакета

```
<stage name="deploy-testing">
  <jobs>
    <job name="deploy-testing">
      <tasks>
        <exec command="ansible" workingdir="deployment-utils/
        ansible/">
          <arg>--inventory-file=testing</arg>
          <arg>web</arg>
          <arg>-m</arg>
          <arg>apt</arg>
```

```
            <arg>-a</arg>
            <arg>name=python-matheval state=latest update_
            cache=yes</arg>
            <runif status="passed" />
          </exec>
        </tasks>
      </job>
    </jobs>
</stage>
```

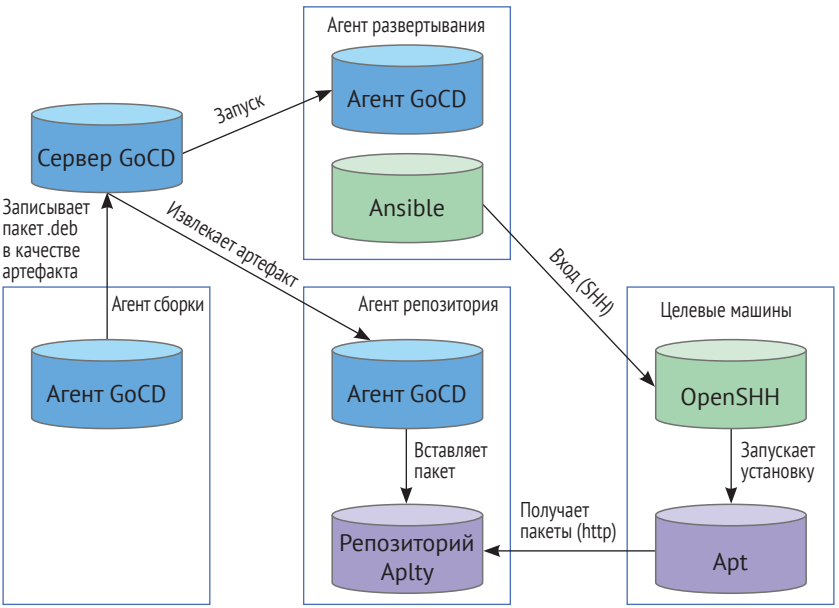


Рис. 10.2 ❖ Агент GoCD запускает Ansible для подключения к целевым машинам по протоколу SSH, для установки нужного пакета

Предполагается, что вы добавляете файлы инвентаризации в каталог `ansible` Git-репозитория `deploy-utils` и что репозиторий Debian уже настроен на целевой машине, как мы обсуждали в главе 7.

10.3. Результаты

Чтобы запустить новый этап, инициируйте полный запуск конвейера, нажав на треугольник **play** на странице конвейера в веб-интерфейсе, или выполните запуск вручную этого одного этапа

в представлении истории конвейера. Можно войти на целевой компьютер, чтобы проверить, был ли пакет успешно установлен:

```
$ dpkg -l python-matheval
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name          Version     Architecture Description
+++-==============-============-============-==============
ii python-matheval  0.1-0.7.1   all          Web service
```

и убедиться, что служба работает.

```
$ systemctl status python-matheval
  python-matheval.service - Package installation informati
    Loaded: loaded (/lib/systemd/system/python-matheval.ser
    Active: active (running) since Sun 2016-03-27 13:15:41
   Process: 4439 ExecStop=/usr/bin/hypnotoad -s /usr/lib/py
 Main PID: 4442 (/usr/lib/packag)
    CGroup: /system.slice/python-matheval.service
            ├─4442 /usr/lib/python-matheval/python-matheval
            ├─4445 /usr/lib/python-matheval/python-matheval
            ├─4446 /usr/lib/python-matheval/python-matheval
            ├─4447 /usr/lib/python-matheval/python-matheval
            └─4448 /usr/lib/python-matheval/python-matheval
```

Вы также можете проверить с главного компьютера, что служба отвечает на порту 8080, как и положено.

```
$ curl --data '["+", 5]' -XPOST http://172.28.128.3:8800
5
```

10.4. Проходя весь путь до реальных условий эксплуатации

Загрузка и развертывание на рабочем месте работают так же, как и в среде тестирования. Таким образом, все, что требуется, – это продублировать конфигурацию последних двух конвейеров, заменить `testing` на `production` и добавить кнопку ручного подтверждения, чтобы развертывание в среде эксплуатации оставалось осознанным решением (листинг 10.4).

Листинг 10.4 ❖ Конфигурация GoCD для распространения и развертывания в рабочей среде

```
<stage name="upload-production">
  <approval type="manual" />
  <jobs>
    <job name="upload-production">
      <tasks>
        <fetchartifact pipeline="" stage="build" job="build-deb"
        srcdir="debian-packages" artifactOrigin="gocd">
          <runif status="passed" />
        </fetchartifact>
        <exec command="/bin/bash">
          <arg>-c</arg>
          <arg> deployment-utils/add-package production \
                stretch *.deb</arg>
        </exec>
      </tasks>
      <resources>
        <resource>aptly</resource>
      </resources>
    </job>
  </jobs>
</stage>
<stage name="deploy-production">
  <jobs>
    <job name="deploy-production">
      <tasks>
        <exec command=" ansible" workingdir="deployment-utils/
                        ansible/">
          <arg>--inventory-file=production</arg>
          <arg>web</arg>
          <arg>-m</arg>
          <arg>apt</arg>
          <arg>-a</arg>
   <arg>name=python-matheval state=latest update_cache=yes</arg>
          <runif status="passed" />
        </exec>
      </tasks>
    </job>
  </jobs>
</stage>
```

Единственная настоящая новость здесь – вторая строка:

```
<approval type="manual" />
```

что заставляет GoCD переходить к этому этапу только тогда, когда кто-то нажимает стрелку подтверждения в веб-интерфейсе.

Вы также должны заполнить файл инвентаризации под названием `production` списком вашего сервера или серверов.

10.5. Достижение разблокировано: базовая непрерывная доставка

Резюмируя, скажем, что конвейер:

- запускается автоматически из фиксаций в исходном коде;
- автоматически собирает пакет Debian из каждой фиксации;
- загружает его в репозиторий среды тестирования;
- автоматически устанавливает его в среде тестирования;
- загружает его после утверждения вручную в хранилище эксплуатационной среды;
- автоматически устанавливает новую версию в эксплуатационной среде.

Базовая структура автоматизированного развертывания от коммита Git в исходных кодах до программного обеспечения, работающего в эксплуатационной среде, теперь готова.

Глава 11

Улучшаем конвейер

Конвейер из предыдущей главы уже вполне пригоден для использования и значительно предпочтительнее сборок, распространения и установки вручную. Тем не менее тут есть возможности для улучшения. Расскажу, как изменить его, чтобы всегда развертывать точную версию, созданную в одном и том же экземпляре конвейера, как запускать «дымовые» тесты после установки и как извлечь шаблон из конфигурации GoCD, чтобы его можно было с легкостью использовать повторно.

11.1. Откаты и установка определенных версий

Конвейер развертывания, разработанный в предыдущих главах, всегда устанавливает последнюю версию пакета. Поскольку логика построения номеров версий обычно приводит к монотонно увеличивающимся номерам версий, обычно это пакет, ранее созданный в том же экземпляре конвейера.

Однако мы действительно хотим, чтобы конвейер развернул точную версию, которая была собрана внутри того же экземпляра конвейера. Очевидное преимущество заключается в том, что это позволяет повторно запускать более старые версии конвейера, устанавливать более старые версии, эффективно предоставляя откат.

Кроме того, можете создать второй конвейер для исправлений на базе того же репозитория Git, но в другой ветке. Если нужно исправление, просто приостанавливаете работу обычного конвейера и запускаете конвейер с исправлениями. В этом случае, если вы всегда устанавливали самую новую версию, найти правильную строку версии для исправления было бы практически невозможно, поскольку она должна быть выше, чем та, что установлена

в настоящее время, но также и ниже, чем при последующей обычной сборке. А – и все это должно делаться автоматически – пожалуйста.

Менее очевидным преимуществом установки очень конкретной версии является то, что она обнаруживает ошибки в конфигурации исходника пакета целевых машин. Если сценарий развертывания устанавливает только новейшую доступную версию и из-за какой-то ошибки репозиторий на целевом компьютере не настроен, процесс установки превращается в пустую операцию, если пакет уже установлен в более старой версии.

Реализация

Необходимо сделать две вещи: выяснить, какую версию пакета установить, а затем сделать это. Мы уже объясняли, как установить конкретную версию пакета с помощью Ansible (листинг 11.1), в главе 7.

Листинг 11.1 ❖ Фрагмент плейбука Ansible для установки версии 1.00 пакета `foo`

```
- apt: name=foo=1.00 state=present force=yes
```

Более общий способ заключается в использовании роли `custom_package_installation`, о которой шла речь в той же главе.

```
- hosts: web roles:
  role: custom_package_installation
  package: python-matheval
```

Ее можно вызвать с помощью `ansible-playbook --extra-vars=package_version=1.00....`

Добавьте этот плейбук в Git-репозиторий `deploy-utils` в виде файла `ansible/deploy-python-matheval.yml`. Для того чтобы найти номер версии для установки, также есть простое, хотя, возможно, и неочевидное решение: записать номер версии в файл; собрать этот файл в качестве артефакта в GoCD; а затем, когда придет время для установки, извлечь этот артефакт и прочитать из него номер версии. На момент написания этих строк у GoCD не было более прямого способа распространения метаданных по конвейерам.

Конфигурация GoCD для передачи версии в плейбук Ansible показана в листинге 11.2.

Листинг 11.2 ❖ Конфигурация GoCD для установки версии из файла `version`

```
<job name="deploy-testing">
  <tasks>
    <fetchartifact pipeline="" stage="build" job="build-deb"
          srcfile="version" artifactOrigin="gocd" />
    <exec command="/bin/bash" workingdir="deployment-utils/
          ansible/">
      <arg>-c</arg>
      <arg>ansible-playbook --inventory-file=testing
--extra-va rs="package_version=$(&lt; ../../version)" deploy-
          python-matheval.yml</arg>
    </exec>
  </tasks>
</job>
```

(XML-тег `<arg>...</ arg>` должен быть на одной строке, поэтому Bash интерпретирует его как единую команду. Здесь он находится на разных строках просто для удобства чтения.)

`$(...)` открывает подпроцесс, который, опять же, является процессом Bash, и вставляет вывод этого подпроцесса в командную строку. `<../../version` – это короткий способ чтения файла, и, поскольку это XML, знак «меньше чем» должен быть экранирован.

Конфигурация производственного развертывания выглядит примерно так же, только с `--inventory-file=production`.

Давайте попробуем!

Чтобы протестировать установку пакета для конкретной версии, у вас должно быть как минимум два запуска конвейера, которые зафиксировали артефакт `version`. В противном случае можно отправить коммиты в исходный репозиторий, а GoCD автоматически подберет их.

Можете запросить установленную версию на целевой машине с помощью команды `dpkg -l python-matheval`. После последнего запуска должна быть установлена версия, собранная в этом экземпляре конвейера.

Затем можете повторно запустить этап развертывания из предыдущего конвейера, например в историческом представлении конвейера, наведя указатель мыши на этап, а затем щелкнув на кружок со стрелкой, который инициирует повторный запуск (рис. 11.1).

Когда этап завершится, вы можете снова проверить установленную версию пакета на целевом компьютере, чтобы убедиться, что была действительно развернута более старая версия.

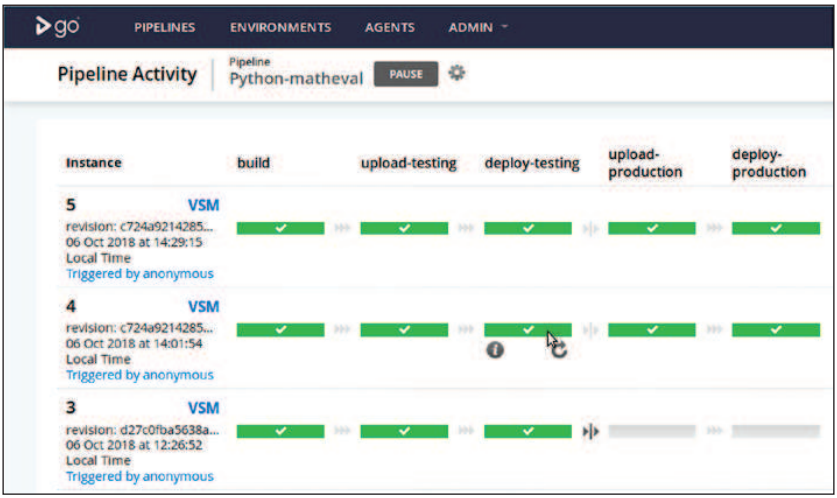


Рис. 11.1 ❖ В историческом представлении конвейера при наведении курсора на полный этап (успешный или нет) отображается значок для повторного запуска этапа

11.2. Проведение дымовых тестов в конвейере

При развертывании приложения важно проверить, действительно ли новая версия приложения работает. Как правило, это делается с помощью дымового теста – довольно простого теста, который, тем не менее, проверяет многие аспекты приложения: он проверяет, запускается ли процесс приложения, привязан ли он к нужному порту и может ли он отвечать на запросы. Часто это подразумевает, что и конфигурация, и соединение с базой данных также являются нормальными.

Когда проводить такой тест?

Дымовые тесты охватывают множество вопросов сразу. Для одного теста может потребоваться работающая сеть, правильно настроенный межсетевой экран, веб-сервер, сервер приложений, база данных и т. д. С одной стороны, это преимущество, потому что означает, что они могут обнаружить множество классов ошибок, а с другой – недостаток, потому что значит, что диагностиче-

ские возможности являются низкими. Когда происходит сбой, вы не знаете, какой компонент виноват, и вам придется заново расследовать каждый сбой.

Дымовые тесты также намного дороже по сравнению с модульными тестами. Обычно на их написание и выполнение уходит больше времени, и они более хрупкие перед лицом изменений конфигурации или данных. Таким образом, типичный совет – иметь небольшое количество дымовых тестов, может быть от 1 до 20, или около 1 % от модульных тестов, которые у вас есть.

Например, если бы вам нужно было разработать сайт по поиску авиарейсов с рекомендациями, ваши модульные тесты охватывали бы различные сценарии, с которыми может столкнуться пользователь, а сайт выдает наиболее подходящие возможные предложения. В дымовых тестах вы бы просто проверили, что можно ввести отправную точку, пункт назначения и дату поездки, а также получить список предложений по рейсам. Если на этом сайте есть раздел для членов, можно было бы убедиться, что к нему нельзя получить доступ без учетных данных и что к нему можно получить доступ после входа в систему. И таким образом, у нас есть три дымовых теста, плюс-минус.

Тестирование белого ящика

Упомянутые выше примеры – это в основном тестирование черного ящика, поскольку они не заботятся о внутреннем устройстве приложения и подходят к приложению так же, как и пользователь. Это очень ценно, потому что в конечном счете вы заботитесь об опыте своего пользователя.

Иногда есть аспекты приложения, которые нелегко проверить с помощью дымовых тестов, к тому же они достаточно часто выходят из строя, чтобы гарантировать автоматическое тестирование. Например, приложение может кешировать ответы от внешних служб, поэтому простое использование определенного функционала не гарантирует использование этого конкретного канала связи.

Практическое решение – когда приложение предоставляет некий тип самодиагностики, например веб-страницу, на которой приложение проверяет собственную конфигурацию на предмет согласованности, проверяет, существуют ли все необходимые таблицы базы данных и доступны ли внешние службы. Один дымовой тест может затем вызвать страницу состояния и ошибку, когда

страница состояния либо недоступна, либо сообщает об ошибке. Это тестирование белого ящика.

Страницы состояния для таких тестов можно использовать повторно при проверках мониторинга, но все же рекомендуется явно проверять их как часть процесса развертывания. Тестирование белого ящика должно не заменять тестирование черного ящика, а скорее дополнять его.

Образец дымового теста черного ящика

Приложение `python-matheval` предлагает простую конечную точку HTTP, поэтому для тестирования подойдет любой HTTP-клиент. При использовании команды `curl` запрос может выглядеть так:

```
$ curl --silent -H "Accept: application/json" \
       --data '["+", 37, 5]' \
       -XPOST http://127.0.0.1:8800/
42
```

Простой способ проверить, что результат соответствует ожиданиям, – передать его через `grep`.

```
$ curl --silent -H "Accept: application/json" \
       --data '["+", 37, 5]' \
       -XPOST http://127.0.0.1:8800/ | grep ^42$
42
```

Результат такой же, как и раньше, но статус выхода не равен нулю, если результат отклоняется от ожидания.

Добавление дымовых тестов в конвейер и роллинг-релизы

Простая интеграция дымовых тестов в конвейер доставки – добавлять этап дымового тестирования после каждого этапа развертывания (то есть один после тестового развертывания и один после производственного развертывания). Такая настройка предотвращает попадание версии вашего приложения в производственное окружение, если она не прошла тестирование. Поскольку дымовой тест – это всего лишь команда оболочки, которая указывает на сбой с ненулевым статусом выхода, добавить его в качестве команды в вашу систему развертывания несложно.

Если у вас запущен только один экземпляр вашего приложения, это лучшее, что вы можете сделать. Тем не менее если у вас несколько машин и несколько экземпляров приложения работают

при участии некоего балансировщика нагрузки, то во время обновления каждый тест можно выполнить в отдельности, и можно отменить обновление, если слишком большое количество экземпляров не прошло тест.

Все крупные, успешные технологические компании охраняют свои производственные системы с помощью таких частичных обновлений, которые охраняются проверками или даже их более сложными версиями.

Простой подход к такому последовательному обновлению – расширение плейбука Ansible для развертывания каждого пакета и запуск дымовых тестов для каждой машины перед переходом к следующему (листинги 11.3 и 11.4).

Листинг 11.3 ❖ Файл `smoke-tests/python-matheval`: простой дымовой тест на базе HTTP

```
#!/bin/bash
curl --silent -H "Accept: application/json" \
     --data '["+", 37, 5]' -XPOST http://$1:8800/ \
     | grep ^42$
```

Листинг 11.4 ❖ Файл `ansible/deploy-python-matheval.yml`: плейбук динамического развертывания с интегрированным дымовым тестом

```
---
- hosts: web
  serial: 1
  max_fail_percentage: 1
  tasks:
    - apt:
        update_cache: yes
        package: python-matheval={{package_version}}
        state: present
        force: yes
    - local_action: >
        command ../smoke-tests/python-matheval
        "{{ansible_host}}"
      changed_when: False
```

Поскольку количество дымовых тестов со временем растет, втискивать их все в плейбук Ansible не практично. Это также ограничивает повторное использование. Вместо этого здесь они находятся в отдельном файле в репозитории deployments utils[1]. Еще

[1] https://github.com/python-ci-cd/deployment-utils.

один вариант – собрать пакет из дымовых тестов и установить его на компьютере, где работает Ansible.

Хотя выполнить команду дымового теста на машине, где установлена служба, было бы легко, при выполнении ее в качестве локального действия (то есть на контрольном хосте, на котором запущен плейбук Ansible) также тестируется часть сети и брандмауэра и, таким образом, более реалистично имитируется реальный сценарий использования.

11.3. Шаблоны конфигурации

Если у вас есть несколько программных пакетов для развертывания, вы создаете конвейер для каждого из них. Пока конвейеры развертывания достаточно похожи по структуре – в основном они используют один и тот же формат упаковки и ту же технологию для установки, – можете повторно использовать структуру, извлекая *шаблон* из первого конвейера и инстанцируя его несколько раз, чтобы создать отдельные конвейеры с той же структурой.

Если внимательно посмотрите на ранее разработанную XML-конфигурацию конвейера, вы, наверное, заметите, что она не очень специфична для проекта `python-matheval`. Помимо дистрибутива Debian и имени deployment playbook все, что здесь есть, может быть повторно использовано для любого программного обеспечения, которое было упаковано в Debian.

Чтобы сделать конвейер более общим, можно определить *параметры* (*params*) как первое, что есть в ваших конвейерах, перед разделом `<materials>` (листинг 11.5).

Листинг 11.5 ❖ Блок параметров для конвейера python-matheval, который должен быть размещен перед материалами

```
<params>
  <param name="distribution">stretch</param>
  <param name="deployment_playbook">deploy-python-matheval.yml
  </param>
</params>
```

Затем замените все вхождения `stretch` внутри определения каждого этапа заполнителем `#{distribution}`, а `deploy-python-matheval.yml` на `#{deploy_playbook}`, что даст вам следующие фрагменты XML-кода:

```
<exec command="/bin/bash">
  <arg>-c</arg>
  <arg>deployment-utils/add-package \
        testing #{distribution} *.deb</arg>
</exec>
```

и

```
<exec command="/bin/bash" workingdir="deployment-utils/
ansible/">
  <arg>-c</arg>
  <arg>ansible-playbook --inventory-file=testing
      --extra-vars="package_version=$(&lt; ../../version)"
      #{deployment_playbook}</arg>
</exec>
```

Следующий шаг к обобщению – перемещение этапов в *шаблон*.

Опять-таки это можно сделать, отредактировав XML-конфигурацию или веб-интерфейс (**Admin** ⇨ **Pipelines**), а затем щелкнуть ссылку **Extract Template** (Извлечь шаблон) рядом с конвейером python-matheval.

Результат в XML выглядит так, как показано в листинге 11.6, если вы выбрали `debian-base` в качестве имени шаблона.

Листинг 11.6 ❖ Конфигурация GoCD для конвейера `matheval` с использованием шаблона

```
<pipelines group="deployment">
  <pipeline name="python-matheval" template="debian-base">
    <materials>
      <git url=
        "https://github.com/python-ci-cd/python-matheval.git"
        dest="source" materialName="python-matheval" />
      <git url=
        "https://github.com/python-ci-cd/deployment-utils.git"
        dest="deployment-utils"
        materialName="deployment-utils" />
    </materials>
    <params>
      <param name="distribution">stretch</param>
      <param name="deployment_playbook">deploy-python-matheval.
                  yml</param>
    </params>
</pipelines>
<templates>
  <pipeline name="debian-base">
      <!—Здесь идут определения этапов -->
  </pipeline>
</templates>
```

Все, что специфично для этого одного программного пакета, теперь находится в определении конвейера, а повторно используемые фрагменты – в шаблоне. Единственное исключение – репозиторий `deploy-utils`, который необходимо добавлять в каждый конвейер отдельно, потому что GoCD не может переместить материал в шаблон.

Чтобы добавить конвейер развертывания для другого приложения, достаточно указать URL-адрес, цель (то есть имя группы в файле инвентаризации Ansible) и дистрибутив. Вы увидите пример этого в следующей главе. На это уходит менее пяти минут работы, когда вы привыкнете к этому инструментарию.

11.4. Как избежать шквала повторных сборок

Когда у вас будет значительное количество конвейеров, вы обратите внимание на неудачный паттерн. Всякий раз, когда вы отправляете коммит в репозиторий `deploy-utils`, он инициирует повторную сборку всех конвейеров. Это бесполезная трата ресурсов, при которой агент сборки или агенты остаются занятыми, поэтому создание пакетов на основе фактических изменений исходного кода откладывается до завершения всех заданий сборки.

У материалов GoCD есть фильтр *игнорирования*, предназначенный для того, чтобы избежать дорогостоящих повторных сборок, когда изменилась только документация (листинг 11.7). Можно использовать его, чтобы игнорировать изменения во всех файлах в репозитории, таким образом избегая шквала повторных сборок.

Листинг 11.7 ❖ Определение материалов GoCD, позволяющее избежать инициализации конвейера

```
<git url="https://github.com/python-ci-cd/deployment-utils.git"
      dest="deployment-utils" materialName="deployment-utils">
    <filter>
        <ignore pattern="*" />
        <ignore pattern="**/*" />
    </filter>
</git>
```

Фильтр * сопоставляет все файлы в каталоге верхнего уровня, а **/* – все файлы в подкаталогах.

Когда вы меняете конфигурацию материала `deploy-utils` во всех конвейерах, чтобы заполучить эти фильтры игнорирования, при новой фиксации в репозиторий `deploy-utils` никакие конвейеры

не запускаются. GoCD по-прежнему опрашивает материал и использует новейшую версию при запуске конвейера. Как и для всех конвейеров, версия материала одинакова на всех этапах.

Игнорирование всех файлов в репозиторий является тупым инструментом и требует от вас вручную запускать конвейер для проекта и вносить изменения в плейбуки развертывания. Таким образом, начиная с версии 16.6 GoCD условия фильтра можно инвертировать с помощью `invertFilter="true"`, чтобы создавать белые списки (листинг 11.8).

Листинг 11.8 ❖ Использование белых списков в материалах GoCD для выборочного запуска изменений для определенных файлов

```
<git url="https://github.com/python-ci-cd/deployment-utils.git"
        invertFilter="true" dest="deployment-utils"
        materialName="deployment-utils">
  <filter>
    <ignore pattern="ansible/deploy-python-matheval.yml" />
  </filter>
</git>
```

Такая конфигурация белого списка для каждого конвейера приводит к фиксациям в репозиторий `deploy-utils`, чтобы инициировать только те конвейеры, для которых изменения актуальны.

11.5. Резюме

Когда вы настраиваете свои конвейеры для развертывания точно такой же версии, которая была собрана в том же экземпляре конвейера, можно использовать это для установки старых версий или выполнения отката.

Шаблоны конвейеров позволяют извлекать схожие черты, которые есть у конвейеров, и сохранять их только один раз. Параметры вносят разнообразие, необходимое для поддержки различных пакетов программного обеспечения.

Глава 12

Безопасность

Как автоматизированное развертывание влияет на безопасность ваших приложений и инфраструктуры? Оказывается, тут есть и преимущества в плане безопасности, и есть вещи, которых следует опасаться.

12.1. Опасности централизации

В конвейере развертывания машина, которая управляет развертыванием, должна иметь доступ к целевым машинам, на которых развернуто программное обеспечение. В простейшем случае на машине развертывания имеется закрытый SSH-ключ, и целевые машины предоставляют доступ владельцу этого ключа.

Это очевидный риск, потому что злоумышленник, который получает доступ к машине развертывания (агент GoCD или сервер GoCD, управляющий агентом), может использовать этот ключ для подключения ко всем целевым машинам, получая полный контроль над ними.

Вот некоторые возможные варианты:

- реализовать усиленную настройку машины развертывания (например, с помощью SELinux или grsecurity);
- защитить паролем SSH-ключ и вводить пароль через тот же канал, который запускает развертывание, например через зашифрованную переменную с сервера GoCD;
- использовать аппаратный токен для хранения ключей развертывания SSH. Аппаратные токены могут быть защищены от программного извлечения ключей;
- разделить развертывание и хосты сборки. Хосты сборки, как правило, требуют гораздо большего количества установленного программного обеспечения, что увеличивает площадь атаки;

- у вас также могут быть отдельные машины развертывания для каждой среды, с отдельными учетными данными;
- на целевых машинах разрешите только непривилегированный доступ через указанный ключ SSH и используйте что-то вроде `sudo`, чтобы разрешить только определенные привилегированные операции.

Каждый из этих вариантов имеет свои издержки и недостатки. Чтобы проиллюстрировать этот момент, обратите внимание, что защищающие паролем ключи SSH помогают, если злоумышленнику удается получить только копию файловой системы, но не тогда, когда злоумышленник получает права суперпользователя на компьютере и, таким образом, может получить дамп памяти, включающий в себя дешифрованные SSH-ключи.

Аппаратное хранилище секретов обеспечивает хорошую защиту от кражи ключей, но затрудняет использование виртуальных систем, и его нужно приобретать и настраивать.

Подход с использованием `sudo` очень эффективен для ограничения распространения атаки, но требует обширной конфигурации на целевой машине, и нужен безопасный способ для ее развертывания. Итак, вы столкнулись с проблемой курицы и яйца, которая требует дополнительных усилий.

С другой стороны, если у вас нет конвейера доставки, развертывание должно происходить вручную. Итак, теперь у вас та же проблема, связанная с необходимостью предоставить людям доступ к целевым машинам. Большинство организаций предлагают некий защищенный компьютер, на котором хранятся SSH-ключи оператора, и на этом компьютере вы сталкиваетесь с тем же риском, что и на машине развертывания.

12.2. Время до выхода на рынок для исправлений безопасности

По сравнению с ручным развертыванием, даже относительно медленный конвейер развертывания по-прежнему довольно быстрый. При обнаружении уязвимости этот быстрый и автоматизированный процесс развертывания может существенно повлиять на сокращение времени до развертывания исправления.

Не менее важным является тот факт, что неуклюжий ручной процесс выпуска соблазняет операторов, и они ведутся на быстрое устранение ошибок, пропуская некоторые этапы процесса обес-

печения качества. Когда этот процесс автоматизирован и быстр, легче присоединиться к процессу, чем пропустить его, поэтому он действительно будет выполняться даже в стрессовых ситуациях.

12.3. Аудит и спецификация ПО

Хороший конвейер развертывания отслеживает, когда какая версия программного пакета была собрана и развернута. Это позволяет ответить на такие вопросы, как «Как долго у нас была эта дыра в безопасности?», «Как скоро после того, как о проблеме сообщили, была исправлена уязвимость в рабочей среде?», и, возможно, даже «Кто одобрил изменение, которое представило уязвимость?».

Если вы также используете управление конфигурацией на базе файлов, которые хранятся в системе контроля версий, то можете ответить на эти вопросы, даже когда речь идет о конфигурации, а не только о версиях программного обеспечения.

Говоря кратко, конвейер развертывания предоставляет достаточно данных для аудита.

В соответствии с некоторыми законами вы должны записать спецификацию ПО[1] в нескольких контекстах, например для программного обеспечения медицинских устройств. Это запись компонентов, содержащихся в вашем программном обеспечении, таких как список библиотек и их версий. Хотя это важно для оценки воздействия нарушения лицензии, это также важно для определения того, какие приложения подвержены уязвимости в определенной версии библиотеки.

Отчет HP Security за 2015 год показал, что 44 % исследованных нарушений стали возможными благодаря известным уязвимостям, которые были известны (и предположительно исправлены) на протяжении как минимум двух лет. Это, в свою очередь, означает, что вы можете почти вдвое снизить риск для безопасности, отслеживая, где и какую версию программного обеспечения вы используете, подписаться на новостную рассылку или ленту, рассказывающую об известных уязвимостях, а также регулярно выполнять повторную сборку и развертывание своего программного обеспечения с исправленными версиями.

Система непрерывной доставки не создает автоматически такую спецификацию ПО за вас, но она дает место, где можно подключить систему, которая это делает.

---

[1] https://en.wikipedia.org/wiki/Software_bill_of_materials.

12.4. Резюме

Непрерывная доставка позволяет быстро и предсказуемо реагировать на вновь обнаруженные уязвимости. В то же время сам конвейер развертывания представляет собой поверхность для атаки, которая, если не будет защищена должным образом, может стать привлекательной целью для злоумышленника.

Наконец, конвейер развертывания может помочь вам собрать данные, которые могут дать представление об использовании программного обеспечения с известными уязвимостями, что позволит вам быть внимательным при исправлении этих брешей в безопасности.

Глава 13

Управление состояниями

Непрерывная доставка удобна и проста для приложения без сохранения состояния, то есть для приложения, в котором данные не хранятся постоянно. Установка новой версии приложения – это простая задача, которая требует установки новых двоичных файлов (или исходных файлов, в случае если язык некомпилируемый), остановки старого экземпляра и запуска нового.

Как только возникает постоянное состояние, становится сложнее. Здесь я рассмотрю традиционные реляционные базы данных со схемами. Можно избежать некоторых проблем, используя неструктурированную базу данных noSQL, но не всегда можно позволить себе такую роскошь. Если вы используете неструктурированную базу данных, придется иметь дело со старыми структурами данных внутри кода приложения, а не с помощью процесса развертывания.

Наряду с изменениями схемы вам, возможно, придется учитывать миграции данных, которые могут включать такие вещи, как заполнение пропущенных значений значением по умолчанию или импорт данных из другого источника данных. В целом такие миграции данных соответствуют тому же шаблону, что и миграции схем, которые должны выполнять либо часть SQL и язык описания данных (DDL)[1], либо внешнюю команду, которая напрямую взаимодействует с базой данных.

---

[1] https://en.wikipedia.org/wiki/Data_definition_language.

13.1. Синхронизация кода и версий базы данных

Управление состояниями – вещь непростая, потому что код обычно привязан к версии схемы базы данных. Есть несколько случаев, когда это может вызвать проблемы.

- Изменения в базе данных часто происходят медленнее, чем обновления приложений. Если версия 1 вашего приложения может работать только с версией схемы 1, а версия приложения 2 может работать только с версией схемы 2, необходимо остановить версию приложения 1, выполнить обновление базы данных и запустить версию приложения 2 только после завершения миграции базы данных.
- Откат к предыдущей версии приложения и, следовательно, к его версии схемы базы данных становится болезненным. Как правило, изменение базы данных или откат может привести к потере данных, поэтому нельзя просто выполнить автоматический выпуск и откат поверх этих границ.

Чтобы уточнить последний пункт, рассмотрим случай, когда столбец добавляется в таблицу в базе данных. В этом случае при откате изменения (повторном удалении столбца) данные теряются. И наоборот, если исходным изменением является удаление столбца, этот шаг обычно нельзя отменить. Можно заново создать столбец того же типа, но данные будут потеряны. Даже если вы архивируете данные удаленных столбцов, в таблицу могут быть добавлены новые строки, а для этих строк архивированных данных нет.

13.2. Разделение версий приложения и базы данных

Существуют инструменты, которые могут помочь вам воспроизвести вашу схему базы данных в определенном состоянии, но они не решают проблему потенциальной потери данных из-за откатов. Единственный практический подход – установить сотрудничество между разработчиками приложений и администраторами базы данных и разбить проблемные изменения на несколько этапов.

Предположим, что ваше желаемое изменение – удалить столбец с ограничением `NOT NULL`. Простое удаление столбца за один шаг со-

провождается проблемами, описанными в предыдущем разделе. Вместо этого можно выполнить следующие шаги.

1. Разверните версию приложения, которая может считывать значения NULL из столбца, даже если значения NULL еще не разрешены.
2. Подождите, пока вы не убедитесь, что не хотите откатиться до значения приложения, которое не может работать со значениями NULL.
3. Разверните изменение базы данных, которое делает столбец обнуляемым (или присвойте ему значение по умолчанию).
4. Подождите, пока вы не убедитесь, что не хотите откатиться до версии схемы, в которой этот столбец – NOT NULL.
5. Разверните новую версию приложения, которое больше не использует столбец.
6. Подождите, пока не убедитесь, что не хотите откатиться до версии вашего приложения, в которой используется этот столбец.
7. Разверните изменение базы данных, которое полностью удаляет столбец.

Некоторые сценарии позволяют пропустить кое-какие из этих шагов или объединить несколько шагов в один. Добавление столбца в таблицу – похожий процесс:

1) разверните изменение базы данных, которое добавляет новый столбец со значением по умолчанию (или допускает значения NULL);
2) разверните версию приложения, которая выполняет запись в новый столбец;
3) при необходимости запустите миграции, которые заполняют столбец для старых строк;
4) если нужно, разверните изменение базы данных, которое добавляет ограничения (например, NOT NULL), что было невозможно в начале,

…с соответствующими ожиданиями между этапами.

Пример изменения схемы

Предположим, у вас есть веб-приложение, поддерживаемое базой данных PostgreSQL, и в настоящее время приложение регистрирует попытки входа в базу данных. Итак, схема выглядит так:

```
CREATE TABLE users (
    id          SERIAL,
    email       VARCHAR NOT NULL,
```

```
    PRIMARY KEY(id)
);

CREATE TABLE login_attempts (
    id          SERIAL,
    user_id     INTEGER NOT NULL REFERENCES users (id),
    success     BOOLEAN NOT NULL,
    timestamp   TIMESTAMP NOT NULL DEFAULT NOW(),
    source_ip   VARCHAR NOT NULL,
    PRIMARY KEY(id)
);
```

Поскольку нагрузка на веб-приложение увеличивается, вы понимаете, что создаете ненужную нагрузку записи для базы данных и начинаете запись во внешнюю службу журнала. Единственное, что вам действительно нужно в базе данных, – это дата и время последнего успешного входа в систему (ваш генеральный директор настаивает, чтобы вы показывали их при каждом входе в систему, потому что аудитор был уверен, что это улучшит безопасность).

Итак, схема, которую вы хотите получить, выглядит так:

```
CREATE TABLE users (
    id          SERIAL,
    email       VARCHAR NOT NULL,
    last_login  TIMESTAMP NOT NULL,
    PRIMARY KEY(id)
);
```

Сценарий прямого изменения базы данных будет выглядеть так:

```
DROP TABLE login_attempts;

ALTER TABLE users
     ADD COLUMN last_login TIMESTAMP NOT NULL;
```

но он страдает от проблемы, описанной ранее, поскольку это привязывает версию схемы к версии приложения, но вы также не можете ввести столбец `NOT NULL` без значения по умолчанию и без предоставления значений для него.

Давайте разберем его на отдельные этапы, у которых нет этих проблем.

Создание нового столбца, допускающего значение NULL

Первым шагом является добавление нового столбца `users.last_login` как необязательного (путем разрешения значений `NULL`). Если отправной точкой была версия схемы 1, это версия 2:

```
CREATE TABLE users (
    id         SERIAL,
    email      VARCHAR NOT NULL,
    last_login  TIMESTAMP,
    PRIMARY KEY(id)
);

-- Таблица login_attempts опущена, потому что она не изменилась.
```

Запуск apgdiff, Another PostgreSQL Diff Tool[1], для двух файлов схем дает нам следующее:

```
$ apgdiff schma-1.sql schema-2.sql

ALTER TABLE users
    ADD COLUMN last_login TIMESTAMP;
```

Это сценарий прямой миграции из схемы 1 в схему 2. Обратите внимание, что не обязательно нужен сценарий отката, потому что каждая версия приложения, которая может иметь дело с версией схемы 1, может также иметь дело с версией схемы 2 (только если приложение не делает что-нибудь глупое, например выполняет команду `SELECT * FROM users` и ожидает определенного числа или порядка результатов. Я предполагаю, что это приложение не настолько глупо).

Такой сценарий миграции может быть применен к базе данных во время работы веб-приложения без простоев.

У MySQL есть одно прискорбное свойство: изменения схемы не являются транзакционными, и они блокируют всю таблицу во время изменений схемы, что сводит на нет преимущества, которые вы получаете в результате пошаговых обновлений базы данных.

Чтобы решить эту ситуацию, существует ряд внешних утилит, которые обходят эту проблему, создавая измененную копию таблицы, постепенно копируя данные из старой таблицы в новую, а затем, наконец, выполняют переименование, чтобы заменить старую таблицу. Один из таких инструментов – gh-ost[2] из GitHub.

Обычно эти утилиты имеют ограниченную поддержку ограничений внешнего ключа, поэтому внимательно оцените их, прежде чем использовать.

Когда изменение схемы завершено, можно развернуть новую версию веб-приложения, которое делает запись в столбец `users.last_login` всякий раз при успешном входе. Обратите внимание,

---

[1] https://www.apgdiff.com.

[2] https://github.com/github/gh-ost/.

что эта версия приложения должна иметь возможность считывать значения NULL из данного столбца, например, возвращаясь к таблице login_attempts, чтобы определить последнюю попытку входа в систему.

Эта версия приложения также может прекратить вставку новых записей в таблицу login_attempts. Более консервативный подход – отложить данный шаг на некоторое время, чтобы вы могли безопасно вернуться к более старой версии приложения.

Миграция данных

В конце концов, столбец users.last_login должен быть NOT NULL, поэтому вы должны генерировать значения для того, где он равен NULL. Здесь таблица last_login является источником таких данных.

```
UPDATE users
  SET last_login = (
        SELECT login_attempts.timestamp
          FROM login_attempts
         WHERE login_attempts.user_id = users.id
           AND login_attempts.success
      ORDER BY login_attempts.timestamp DESC
         LIMIT 1
     )
WHERE users.last_login IS NULL;
```

Если значения NULL остаются, скажем, из-за того, что пользователь ни разу не вошел в систему успешно, или потому, что в таблице last_login нет данных для всех учетных записей в базе данных, у вас должен быть некий запасной вариант, который может быть фиксированным значением. Здесь я выбираю легкий путь и просто использую NOW() в качестве запасного варианта.

```
UPDATE users SET last_login = NOW() WHERE last_login IS NULL;
```

Эти два обновления могут снова работать в фоновом режиме, пока приложение работает. После этого обновления больше в users.last_login не должно отображаться никаких значений NULL. После нескольких дней ожидания и проверки того, что это действительно так, пришло время применить необходимое ограничение.

Применение ограничений и очистка

Если вы уверены, что в столбце last_login нет строк, пропускающих значения, и что вы не собираетесь выполнять откат до версии приложения, в которой указаны пропущенные значения, можно

развернуть версию приложения, которая перестает использовать таблицу `login_attempts`, избавиться от этой таблицы, а затем применить ограничение `NOT NULL` (см. также рис. 13.1).

```
DROP TABLE login_attempts;

ALTER TABLE users
    ALTER COLUMN last_login SET NOT NULL;
```

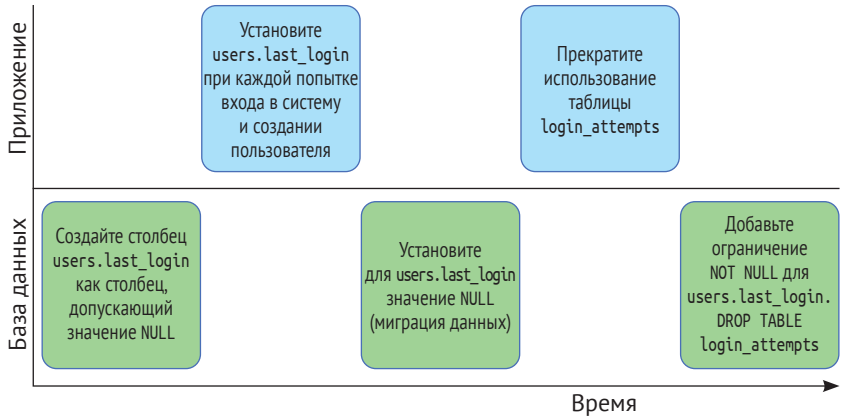


Рис. 13.1 ❖ Последовательность шагов обновления приложения и базы данных. Каждая версия базы данных совместима с версиями приложения до и после, и наоборот

Таким образом, одно логическое изменение базы данных было распространено на три обновления базы данных (два обновления схемы и одна миграция данных) и два обновления приложения.

Это вынуждает приложить немного больше усилий при разработке приложений, однако вы получаете эксплуатационные преимущества. Одним из таких преимуществ является постоянная доступность приложения для производства.

Предварительные условия

Если вы развертываете одно логическое изменение базы данных в несколько этапов, то должны выполнить несколько развертываний вместо одного большого развертывания, в котором одновременно происходят изменения кода и схемы. Это практично, только если развертывания (по крайней мере в основном) являются автоматизированными и если организация предлагает достаточную

непрерывность, чтобы вы могли фактически завершить процесс изменений.

Если разработчики постоянно решают срочные проблемы, велика вероятность, что они так и не дойдут до добавления этого конечного желаемого ограничения `NOT NULL`, и какая-либо необнаруженная ошибка приведет к отсутствию информации в будущем.

Также следует установить какой-нибудь трекер проблем, с помощью которого можно проследить путь миграций схем, чтобы убедиться, что все было доделано, например в случае, если разработчик покидает компанию.

Инструментарий

К сожалению, я не знаю ни одного инструмента, который бы полностью поддерживал описанный мною взаимосвязанный цикл выпуска баз данных и приложений. Есть утилиты, которые управляют изменениями схемы в целом. Например, Sqitch[1] и Flyway[2] являются довольно общими фреймворками, которые используются для управления изменениями базы данных и откатами.

На нижнем уровне существуют такие инструменты, как `apgdiff`, которые сравнивают старую и новую схемы и используют это сравнение для генерации операторов DDL, которые переносят вас из одной версии в другую. Такие автоматически генерируемые операторы DDL могут сформировать основу для сценариев обновления, которыми затем управляют Sqitch или Flyway.

Некоторые системы объектно-реляционного отображения также поставляются с фреймворками, которые обещают управлять схемами миграции за вас. Тщательно проверьте, позволяют ли они выполнять откат без потери данных.

Структура

Если вы отделяете развертывания приложений от развертываний схемы, из этого следует, что у вас должно быть как минимум два отдельно развертываемых пакета: один – для приложения и один – для сценариев схемы базы данных и миграций схемы. Если вы хотите или должны поддерживать откат схем базы данных, нужно помнить, что вам нужны метаданные, связанные с новой схемой, чтобы иметь возможность выполнить откат к старой версии.

---

[1] https://sqitch.org.

[2] https://flywaydb.org.

Описание базы данных для версии схемы 5 не знает, как откатиться с версии 6 на версию 5, потому что ничего не знает о версии 6. Таким образом, всегда должна быть установлена самая новая версия пакета с файлами схемы и вы должны отделять установленную версию от текущей активной версии базы данных. Инструменты, управляющие миграцией схемы, могут быть независимыми от приложения и его схемы и поэтому должны находиться в стороннем программном пакете.

Единого решения не существует

Не существует единого решения, которое бы автоматически управляло всеми вашими миграциями данных во время развертываний. Вы должны тщательно спроектировать изменения приложения и базы данных, чтобы разделить их и развернуть отдельно. Обычно на стороне разработки приложения работы больше, но это предоставляет вам возможность развертывания и отката без блокировки при изменении базы данных.

Инструментарий доступен для некоторых частей, но, как правило, не для общей картины. Кто-то должен отслеживать версии приложения и схемы или автоматизировать их.

13.3. Резюме

Состояние в базе данных может усложнить обновление приложения.

Несовместимые изменения структуры данных и схемы можно разбить на несколько более мелких этапов, каждый из которых совместим с предыдущим.

Это позволяет обновлять приложения без простоев, за счет необходимости выполнять несколько развертываний приложений и схем.

Глава 14

Выводы и перспективы

Прочитав эту книгу, вы должны иметь четкое представление о том, как и зачем реализовывать непрерывную интеграцию и непрерывную доставку для проекта Python. Это не простая тема, но многие примеры должны помочь довольно быстро сориентироваться, и даже реализация лишь некоторых аспектов может дать вам преимущества. В среде взаимодействия, демонстрируя эти преимущества, вам будет легче убедить других, что стоит потратить свое время на набор инструментальных средств и улучшения процесса.

14.1. Что дальше?

В этой заключительной главе давайте рассмотрим концепции, которые могут помочь вам развить еще более зрелый процесс разработки программного обеспечения, связанный с непрерывной интеграцией и непрерывной доставкой.

Улучшенное обеспечение качества

Повысить качество своего программного обеспечения можно так же просто, как увеличить покрытие своего приложения модульными тестами. Однако не все классы ошибок можно обнаружить таким образом, например регрессия производительности или ошибки для случаев, о которых вы раньше и не думали.

Чтобы зафиксировать регрессию производительности, можно создать отдельную среду тестирования производительности и за-

пустить в ней предопределенный набор нагрузочных тестов и тестов производительности. Можно добавить это как еще один этап в свой конвейер развертывания.

Обрабатывать неожиданные случаи сложнее, потому что они по определению застают вас врасплох. Для определенных типов приложений автоматический фаззинг может найти входные данные, которые приводят к аварийному завершению работы вашего приложения, и предоставить эти данные разработчикам в качестве примеров.

Существуют архитектурные подходы, позволяющие сделать ваши приложения надежными, защищая их от неожиданного пользовательского ввода и ошибочных сценариев, но с точки зрения инструментария лучшее, что вы можете сделать, – это сделать реакцию приложения на подобные ошибки более устойчивой.

Специализированные средства отслеживания ошибок могут помочь вам выявлять такие ошибки. Они дают разработчикам больше понимания того, как воспроизводить и диагностировать эти ошибки. Например, Sentry[1] – это централизованный трекер ошибок с открытым исходным кодом, с доступным размещенным решением.

Метрики

В более крупных системах и организациях сбор и агрегирование метрик необходимы для поддержания управляемости системы. Существует даже тенденция строить мониторинг на данных временных рядов.

В контексте системы развертывания некоторые точки данных, которые можно собрать, включают в себя дату начала и продолжительность каждого этапа или задачи, какая версия была создана, а также характеристики этой конкретной версии, в том числе данные о производительности и использовании, такие как показатели вовлеченности, размер созданных артефактов, обнаруженные дефекты и уязвимости и т. д.

Осмыслить собранные данные не всегда просто, и об этом написаны целые книги. Тем не менее полезно установить способ сбора метрик всех типов и создать информационные панели, которые помогают интерпретировать их.

---

[1] https://sentry.io/welcome/.

Автоматизация инфраструктуры

Управление конфигурацией важно для масштабирования вашей инфраструктуры, но такого инструмента, как Ansible, недостаточно для удовлетворения всех нужд и масштабов.

Конфигурация в базе данных и управление секретами

По мере увеличения объема данных конфигурации хранить их в виде простых текстовых файлов становится непрактично. Поэтому вам придется поддерживать базу данных конфигурации и использовать механизм динамической инвентаризации Ansible[1] для распространения данных в систему управления конфигурациями.

Тем не менее хранение паролей, закрытых ключей и других секретов в базе данных – вопрос всегда деликатный. Вам необходимо приложение поверх базы данных, чтобы эти секреты не утекали к пользователям, которые не должны иметь к ним доступа.

Такие приложения уже существуют. Выделенные системы управления секретами хранят секреты в зашифрованном виде и тщательно контролируют доступ к ним. Примерами таких приложений являются Keywhiz[2] от компании Square или Vault[3] от HashiCorp, авторов Vagrant.

Системы управления секретами обычно предлагают плагины для создания сервисных учетных записей, такие как учетные записи базы данных MySQL или PostgreSQL, и сменяют пароли без участия человека. Это также означает, что никто не должен видеть случайно сгенерированные пароли.

Вместо того чтобы вставлять конфигурацию приложения в машины или контейнеры, в которых работает приложение, вы также можете создавать приложения для получения конфигурации из центрального расположения. Подобное центральное расположение обычно называют системой обнаружения сервисов. Такие инструменты, как etcd[4] от проекта CoreOS и Consul[5] от HashiCorp, облегчают управление большими объемами конфигурации. Они

---

1 https://docs.ansible.com/ansible/latest/dev_guide/developing_inventory.html.

2 https://square.github.io/keywhiz/.

3 https://www.hashicorp.com/blog/vault-announcement/.

4 https://github.com/etcd-io/etcd.

5 https://www.consul.io.

также предоставляют дополнительные функции, такие как базовый мониторинг сервисов и предоставление потребителям только рабочих экземпляров конечной точки службы.

В качестве иллюстрации давайте представим, что приложению, требующему больших объемов данных конфигурации, можно предоставить только секретный ключ для аутентификации в системе обнаружения сервисов и информацию о среде, в которой оно работает. Затем приложение считывает все остальные конфигурации из центральной службы. Если приложению требуется доступ к службе хранения и существует несколько экземпляров, которые могут предоставить эту службу, служба мониторинга гарантирует, что приложение получит адрес рабочего экземпляра.

Такой подход к обнаружению сервисов позволяет использовать шаблон, именуемый *неизменной инфраструктурой*. Это означает, что вы создаете контейнер (такой как контейнер Docker или даже образ виртуальной машины), а затем, вместо того чтобы распространять только ваше приложение через различные среды тестирования, вы распространяете через них весь контейнер. Система управления кластерами предоставляет учетные данные для подключения к системе обнаружения сервисов; в противном случае контейнеры остаются без изменений.

Инфраструктура в качестве кода

Традиционная система непрерывной доставки, как было описано в предыдущих главах, обычно ограничена одной ветвью в системе управления версиями, потому что для развертывания кода существует только одна среда тестирования.

Облачная инфраструктура меняет игру. Она позволяет декларативное описание целых сред, состоящих из нескольких баз данных, сервисов и виртуальных серверов. Тогда создание новой среды становится вопросом выполнения одной команды, позволяющей развернуть каждую ветку в новой отдельной среде.

Ведущими инструментами для создания новых сред являются Terraform[1] и CloudFormation[2].

---

[1] http://www.terraform.io.

[2] https://aws.amazon.com/ru/cloudformation/.

14.2. Заключение

Автоматизация развертываний делает разработку программного обеспечения и эксплуатацию эффективнее и приятнее. Я показал вам аккуратное и практическое введение в этот процесс, что, в свою очередь, позволяет вам внедрить непрерывную доставку в своей компании.

Это большой шаг для компании, которая разрабатывает программное обеспечение, а также небольшая часть автоматизации вашей инфраструктуры и часть пути к эффективному и гибкому процессу разработки программного обеспечения.

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу
в книготорговой компании **«Галактика»**
(представляет интересы издательств «ДМК Пресс»,
«СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: (499) **782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Мориц Ленц

Python: Непрерывная интеграция и доставка

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Мамонов А. Е., Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 10,5. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**