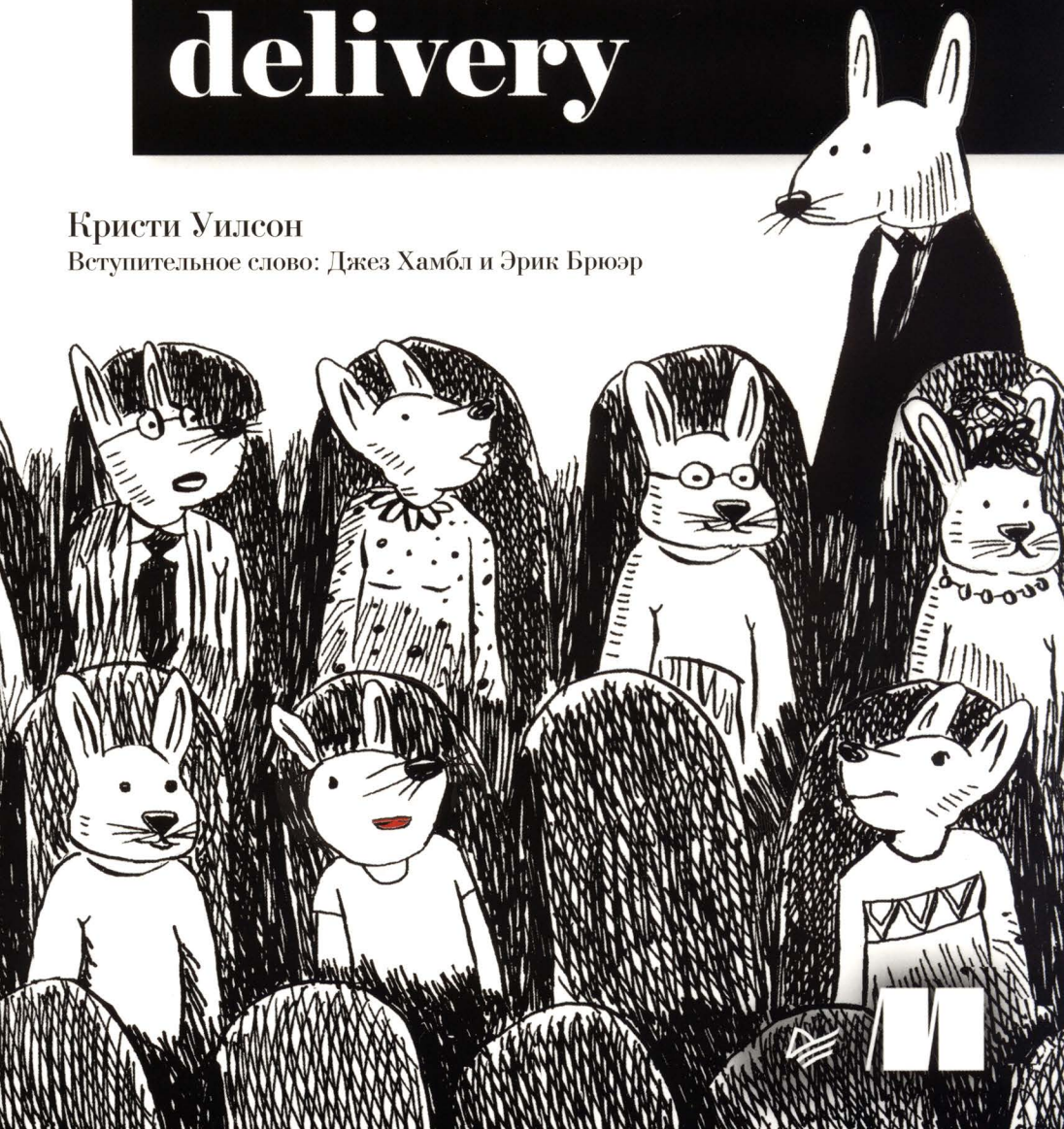
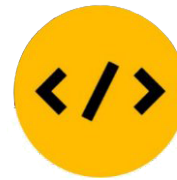
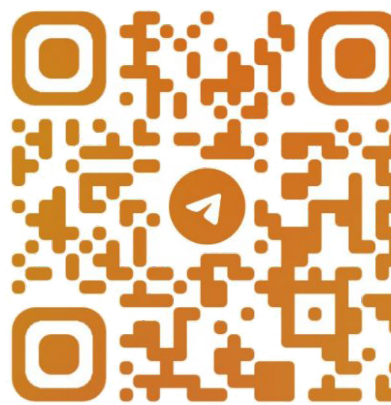
грокаем

continuous delivery

Кристи Уилсон
Вступительное слово: Джез Хамбл и Эрик Брюэр







@CODELIBRARY_IT

ПИТЕР®

grokking continuous delivery

Christie Wilson

Forewords by Jez Humble and Eric Brewer

MANNING

SHELTER ISLAND

грокаем
continuous delivery

Кристи Уилсон

Вступительное слово: Джез Хамбл и Эрик Брюэр

ПИТЕР®

Санкт-Петербург · Москва · Минск

2024

Кристи Уилсон

Грокаем Continuous Delivery

Серия «Библиотека программиста»

Перевела с английского Н. Григорьева

ББК 32.973.2-018
УДК 004.41

Уилсон Кристи

У36 Грокаем Continuous Delivery. — СПб.: Питер, 2024. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2372-8

Код должен быть готов к релизу всегда!

Пайплайн Continuous Delivery автоматизирует процессы контроля версий, тестирования и развертывания при минимальном вмешательстве разработчика. Освойте инструменты и методы непрерывной доставки, и вы сможете быстро и последовательно добавлять функции и выпускать обновления.

«Грокаем Continuous Delivery» — это руководство по настройке и работе с пайплайном непрерывной доставки. В каждой главе рассматривается отдельный сценарий, с которым вы столкнетесь при создании системы CD, и приводятся реальные примеры, например автоматическое масштабирование и тестирование унаследованных приложений. Кристи Уилсон сопровождает каждый шаг иллюстрациями, кристально четкими объяснениями и практическими упражнениями для закрепления полученных знаний.

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

ISBN 978-1617298257 англ.

ISBN 978-5-4461-2372-8

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.01.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 32,250. Тираж 1500 экз. Заказ Х-274.
Отпечатано в типографии ООО «Экопейпер», 420044, Россия, г. Казань, пр. Ямашева, д. 36Б.

Краткое содержание

ЧАСТЬ 1
ЧТО ТАКОЕ НЕПРЕРЫВНАЯ ДОСТАВКА

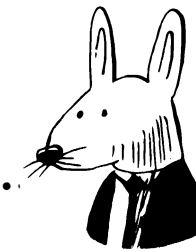
ЧАСТЬ 2
ПОДДЕРЖАНИЕ ПО В СОСТОЯНИИ ГОТОВНОСТИ К ДОСТАВКЕ

ЧАСТЬ 3
СДЕЛАЕМ ПРОЦЕСС ДОСТАВКИ ПРОЩЕ

ЧАСТЬ 4
РЕАЛИЗАЦИЯ НЕПРЕРЫВНОЙ ДОСТАВКИ

ПРИЛОЖЕНИЯ

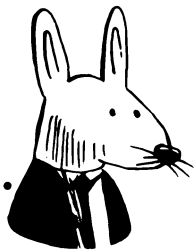
Оглавление



Моей дочери Александре —
самой важной для меня «доставке»[1]*!*

[1] Непереводимая игра слов. Delivery (*англ.*) означает и доставку, и рождение ребенка. — *Примеч. ред.*

Вступительное слово



Когда мы с Дэвидом Фарли (David Farley) писали книгу «Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation» (Addison-Wesley, 2010)[1], то благодаря многолетнему применению описанных в ней принципов мы знали, что предлагаем современный, целостный подход к доставке программного обеспечения, дающий существенные преимущества командам и компаниям, которые его используют. Многочисленные исследования (в том числе то, в котором я принимал участие под руководством доктора Николь Форсгрен (Dr. Nicole Forsgren), — о нем рассказывается в главах 8 и 10 этой книги) показали, что применение такого подхода приводит к повышению качества и стабильности разработки ПО, а также к более оперативной его доставке.

Невзирая на то что непрерывная интеграция и непрерывная доставка (continuous integration/continuous delivery, сокращенно CI/CD) сегодня считаются стандартной практикой, их по-прежнему на удивление трудно внедрить и правильно реализовать. Все еще слишком много команд (и пользователей!) сталкиваются с тем, что релизы выходят редко и с высоким риском, по вечерам или в выходные дни; случаются плановые и внеплановые простои, откаты и проблемы с производительностью, работоспособностью и безопасностью. Всего этого можно избежать, но нужны постоянные инвестиции в команды, инструменты и корпоративную культуру.

Важно отметить, что многие новички в нашей сфере не знакомы с основополагающими методиками и способами их применения. Книга «Грокаем Continuous Delivery» отлично восполняет этот пробел. Кристи Уилсон, эксперт в области непрерывной доставки, возглавляющая в компании Google CI/CD-проект Tekton с открытым исходным кодом, написала всеобъемлющее, ясное и подробное руководство, в котором тщательно разобраны нюансы технологии и реализации современной доставки программных продуктов. Кристи не только рассказывает о принципах и их применении, но и наглядно показывает, почему они важны, а также приводит

[1] ФарлиД., Хамбл Дж. «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий».

пошаговые решения самых сложных проблем, с которыми, по моим наблюдениям, сталкиваются многие команды, например итеративная разработка функций и работа с унаследованным кодом.

Я надеюсь, что эта книга займет достойное место в списке пособий для новичков в каждой команде разработчиков. Она также станет весьма полезным руководством для более опытных инженеров, осваивающих новый для них стиль работы. Я благодарен Кристи за создание источника информации, который, без сомнения, поможет лучше понять, как правильно реализовать современный процесс доставки программного обеспечения на благо всей нашей отрасли и широкой общественности — им, в конечном итоге, мы и служим.

— Джез Хамбл (Jez Humble)
соавтор книг «Continuous Delivery»,
«The DevOps Handbook»[1] и «Accelerate!»[2]

Прелесть программ в том, что со временем их можно улучшать. Но в этом и их проклятие — мы, по сути, все время что-то меняем, потому что можем это изменить. Неустанное стремление к внедрению новых функций и другим улучшениям приводит к необходимости максимально быстрой интеграции изменений, их тестирования и доставки пользователям.

Кристи Уилсон живет этим процессом и наблюдает за ним с разных сторон, и она написала книгу о том, как командам разработчиков получить необходимое ускорение. Действительно, команда, способная на ускоренную разработку и внедрившая высокую степень автоматизации, получает конкурентное преимущество. Такие команды не только со временем завоевывают значительную часть рынка — в них выше моральный дух и ниже уровень текучки персонала. Ведь быть продуктивным приятно!

Бытует заблуждение, что более медленные процессы, в которых, возможно, развертыванию что-то препятствует, являются более безопасными или надежными. Многие команды не любят перемены и поэтому выпускают изменения, например, раз в квартал. У такого подхода есть два серьезных недостатка. Во-первых, сложная задача интеграции большого количества изменений в этом случае откладывается до самого конца разработки, но если изменений, которые нужно интегрировать с момента последнего выпуска, накапливается много, их внедрение чревато сбоями и большими задержками. Во-вторых, низкая скорость затрудняет быстрое исправление проблем безопасности, что критически важно для большинства команд. Подход, представленный в этой книге, основан на непрерывных (небольших) интеграциях, позволяющих быстро получать обратную связь по возникающим проблемам и эффективно вносить исправления безопасности.

За последние несколько лет внимание к проблемам безопасности резко возросло, особенно в связи с «атаками через цепочки поставок» (supply chain attacks). В совре-

[1] Ким Дж., Дебуа П., Уиллис Дж. и Хамбл Дж. «Руководство по DevOps».

[2] Форсгрен Н., Ким Дж. и Хамбл Дж. «Ускоряйся!»

менное ПО входят компоненты из различных источников — других команд, других компаний — и компоненты с открытым исходным кодом. Оно запросто может состоять из тысячи частей, и все их необходимо интегрировать вместе. Однако это требует совершенно нового уровня автоматизации: нам нужно знать все исходные данные, откуда они взялись и как они совместно использовались. Книга Кристи — одна из первых, в которой освещаются эти вопросы и рассказывается, как повысить уровень безопасности систем.

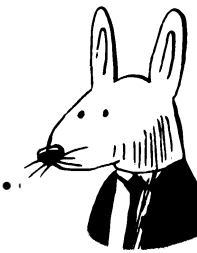
И наконец, несмотря на то что в рассматриваемой области существует огромное количество инструментов и опций, эта книга отлично описывает ее ключевые концепции и цели, одновременно приводя практические примеры и возможные альтернативы.

Для меня эта книга стала настоящим глотком свежего воздуха; надеюсь, что она понравится и вам.

— Эрик Брюэр (Eric Brewer),
вице-президент по инфраструктуре
и fellow-разработчик[1] в компании Google

[1] Fellow — высшая ступень инженерного пути развития карьеры, например заслуженный разработчик и т. д. По степени влиятельности соответствует высшему руководству организации. — *Примеч. ред.*

Предисловие



Программирование увлекло меня с того самого момента, как я узнала о его существовании. Я помню, как (примерно 300 лет назад) друг рассказал мне о программе игры в шахматы, которую он написал; хотя я совершенно не понимала, о чем он говорит, я осознала, что (а) я никогда не задумывалась о том, как работают компьютеры, и (б) теперь мне совершенно необходимо узнать об этом как можно бóльше. То, что случилось дальше, иногда было удивительно, а иногда приводило в замешательство («переменные подобны почтовому ящику» — это сравнение для меня теперь полно смысла, но сначала оно просто сразу вылетело у меня из головы). А после первого урока по Turbo Pascal в старших классах и множества самоучителей по Java я влипла окончательно.

Хотя программирование было увлекательным само по себе, процессы организации разработки заинтриговали меня едва ли не больше. На протяжении по крайней мере половины своей карьеры я с разочарованием замечала, как мало внимания им уделяется, несмотря на то что они чрезвычайно важны не только для качества создаваемых программ, но и для благополучия и эффективности создателей этих программ. Более того, я была обескуражена, когда поняла, что сами разработчики и менеджеры не считают такое отношение проблемой. Часто оно диктовалось мнением, что выдавать код как можно быстрее — лучший способ окупить вложения по максимуму.

По иронии судьбы время и исследования показали, что скорость действительно определяет успех, но чтобы усилия разработчиков на самом деле были эффективными, а их работа — стабильной, скорость должна сочетаться с безопасностью. Максимальная скорость и безопасность программной разработки составляют основу непрерывной доставки, именно поэтому данная концепция и соответствующие методики оказались мне близки. Тем не менее до недавнего времени я ничего не знала о непрерывной доставке.

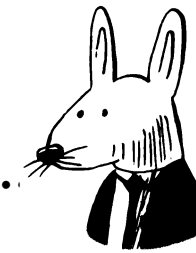
В первую очередь меня привлекли тесты и автоматизация. Я до сих пор помню чувство свободы, которое я испытала, когда познакомилась с тестами, и особенно с разработкой через тестирование; тогда я поняла, что могу проверять програм-

мы, которые пишу, в процессе их создания. Казалось, что у меня с плеч свалился огромный камень, когда я смогла проверять свою работу прямо по ходу дела, — так я избавилась от назойливого внутреннего голоса, который иногда убеждал меня, что я не знаю, что делаю, и что ничего из того, что я понаписала, работать не будет. Инструменты и автоматизация также помогали мне чувствовать себя уверенно: они брали на себя очень ответственные задачи, которые поначалу пугали. Казалось, будто рядом с тобой сидит друг и подсказывает тебе, что делать.

Концепция непрерывной доставки берет все лучшее из тестирования и автоматизации, которые всегда помогали мне в моей работе, и формирует набор практик, дающих возможность любому разработчику усовершенствовать свои методы. Я хочу помочь всем, особенно тем, кто иногда сомневается в себе или борется со страхом сделать что-то не то (и я думаю, подобные чувства хотя бы иногда испытывает большинство из нас), — ощутить ту же свободу, самостоятельность и уверенность в себе, что и я, когда написала свой первый тест.

Спасибо, что нашли время прочитать эту книгу. Я надеюсь, что по крайней мере вы сделаете правильный вывод: большинство багов и ошибок в коде практически не имеют отношения к самому коду (и уж точно не имеют отношения к человеку, который его написал). На самом деле их причины лежат в процессах разработки, которым просто нужно уделить немного больше внимания, и тогда усилия, приложенные к обновлению и исправлению этих процессов, окажутся не напрасны.

Благодарности



Прежде всего, спасибо моему мужу Торину Сэндаллу (Torin Sandall), который почему-то до сих пор меня поддерживает (мы теперь половинки одного целого, но все еще никак не привыкнем к этому!) и не только вдохновлял меня все время, пока я работала над этой книгой, но и взял на себя уйму забот, чтобы я смогла закончить работу в самый напряженный период нашей жизни. (Скажем так, переезд из Нью-Йорка в Ванкувер, свадьба и рождение ребенка в течение одного года — это только часть истории!)

Спасибо Берту Бейтсу (Bert Bates) за то, что он навсегда изменил мое представление о преподавании и подаче материала. Я надеюсь, что вы найдете в этой книге отголоски своего чуткого и действенного стиля! Мне еще многое предстоит пройти, но я буду применять то, чему вы меня научили, всю оставшуюся жизнь во всем, что я пишу, и при каждом выступлении на конференции.

Спасибо моим друзьям, не знакомым с технической стороной дела, которые постоянно поддерживали меня (хотя я не уверена, что мне удалось объяснить им, о чем эта книга) и даже смотрели мои стримы в Twitch; особенно Саре Таплин (Sarah Taplin) и Саше Бурдену (Sasha Burden), которых я с удовольствием воплотила в альтернативной книжной реальности в качестве основателей стартапов в главах 3 и 8.

Выражаю благодарность моим учителям, с которыми мне посчастливилось пересечься и которые так изменили мою жизнь: Стюарту Гейтту (Stuart Gaitt) — за то, что подбадривал маленькую странную девочку; Шеннон Роджерс (Shannon Rodgers) — за то, что научила меня по-настоящему думать; и Аману Абдулле (Aman Abdulla) — за то, что дал мне практические технические навыки и привил высокие стандарты, без которых я бы не смогла достичь того, что имею сегодня.

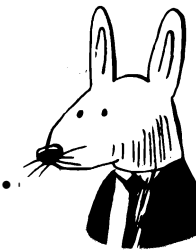
Огромное спасибо всем сотрудникам издательства Manning за предоставленную мне возможность написать эту книгу, за воплощение моей мечты! Спасибо издателю Марьян Бейс (Marjan Bace); Майку Стивенсу (Mike Stephens) за то, что связался со мной и дал начало этому безумному приключению; Иэну Хоуфу (Ian Hough) — за тесное и терпеливое сотрудничество со мной глава за главой; Марку Элстону (Mark Elston) — за редактирование неуклюжих первых набросков (с большими недочета-

ми); Нинославу Черкезу (Ninoslav Cerkez) за тщательное научное рецензирование; а также ревьюеру Александру Драгосавлевичу (Aleksandar Dragosavlijevic). Спасибо Шэрон Уилки (Sharon Wilkey), которая помогла мне исправить и осмыслить множество грамматических (и прочих) ошибок, Кэтлин Россланд (Kathleen Rossland) за терпеливое руководство издательским процессом, а также множеству других людей, оставшихся за кадром, которые помогали в создании этой книги. Спасибо всем редакторам: Андреа С. Гранате (Andrea C. Granata), Барнаби Норману (Barnaby Norman), Билли О'Каллагану (Billy O'Callaghan), Бренту Хонаделю (Brent Honadel), Крису Винеру (Chris Viner), Клиффорду Турберу (Clifford Thurber), Крейгу Смиту (Craig Smith), Даниэлю Васкесу (Daniel Vasquez), Джавиду Аскерову (Javid Asgarov), Джону Гатри (John Guthrie), Хорхе Бо (Jorge Bo), Камешу Ганесану (Kamesh Ganesan), Майку Халлеру (Mike Haller), Нинославу Черкезу (Ninoslav Cerkez), Оливеру Кортену (Oliver Korten), Прабхути Пракашу (Prabhuti Prakash), Раймонду Чунгу (Raymond Cheung), Серхио Фернандесу Гонсалесу (Sergio Fernández González), Сваминатану Субраманиану (Swaminathan Subramanian), Тобиасу Гетросту (Tobias Getrost), Тони Свитсу (Tony Sweets), Вадиму Туркову (Vadim Turkov), Уильяму Джамиру Сильве (William Jamir Silva) и Зородзайи Мукуя (Zorodzayi Mukuya) — ваши предложения помогли сделать эту книгу лучше. Спасибо также команде маркетинга издательства Manning, особенно Радмиле Ерчеговац (Radmila Ercegovac) за то, что помогла мне выйти из зоны комфорта и засветиться в подкастах, а также Степану Юрековичу (Stjepan Jureković) и Лукасу Веберу (Lucas Weber) за мой дебют в Twitch, который оказался довольно увлекательным.

Огромное спасибо всем, кто терпеливо рецензировал книгу по мере ее написания и высказывал свои замечания, особенно всем сотрудникам Google, которые нашли время и помогли мне пройти огонь, воду и медные трубы, в том числе Джоэлу Фридману (Joel Friedman), Дамите Карунаратне (Damith Karunaratne), Дэну Лоренку (Dan Lorenc) и Майку Далину (Mike Dahlin). Миллион благодарностей Стивену Эрнесту (Steven Ernest) за то, что научил меня, как важны комментарии к коммитам и примечания к релизам, и дал мне понять, что я ужасно непоследовательно использую кавычки. И спасибо Джеропу Кипруто (Jerop Kipruto) за то, что он не только прочитал главы этой книги, но и увлекся содержанием и сразу же применил его на практике!

И наконец, спасибо Эрику Брюеру за поддержку и отзывы на протяжении всего пути, а также за то, что не только поверил в книгу, но и нашел время, чтобы написать к ней вдохновляющее вступительное слово. Спасибо также Джезу Хамблу за всю мудрость, которой он поделился со мной в начале этого пути — и которую я, к сожалению, полностью проигнорировала, а теперь усвоила на собственном горьком опыте. Я считаю, что лучше поздно, чем никогда! Знайте: ваше одобрение во вступлении к этой книге — лучшее доказательство моих профессиональных заслуг.

О книге



Задача книги — стать недостающим руководством по началу работы с непрерывной доставкой и ее эффективному применению. Книга описывает практики, составляющие непрерывную доставку, и рассказывает о необходимых компонентах автоматизации, которая поддерживает эти практики. Подобные знания приходят постепенно, с годами упорного труда. Надеюсь, что моя книга поможет сократить этот путь и вам не придется получать их на собственном опыте!

Для кого эта книга

Книга «Грокаем Continuous Delivery» предназначена для всех, кто день ото дня занимается созданием программного обеспечения. Чтобы польза от книги была максимальной, вы должны быть знакомы с основами написания shell-скриптов, владеть хотя бы одним языком программирования и иметь опыт тестирования. Вам также понадобится опыт работы с системой контроля версий, HTTP-серверами и контейнерами. Однако глубоко разбираться во всех этих темах не обязательно; вы сможете изучить их по ходу чтения, если понадобится.

Структура и план книги

Книга состоит из 13 глав, разбитых на четыре части. Первые две главы представляют собой введение в концепцию непрерывной доставки и содержат список терминов, которые вам понадобятся на протяжении всей книги:

- В главе 1 дается определение *непрерывной доставки* и объясняется ее связь со смежными понятиями, такими как *непрерывная интеграция* и *непрерывное развертывание*.
- В главе 2 перечислены основные элементы автоматизации непрерывной доставки, а также дана терминология, используемая в остальных частях книги.

Часть 2 посвящена процессам, составляющим непрерывную интеграцию и необходимым для непрерывной доставки:

- Глава 3 поясняет жизненно важную роль системы контроля версий в процессе непрерывной доставки; без этой системы непрерывная доставка невозможна.
- В главе 4 рассматривается эффективный, но редко обсуждаемый элемент непрерывной интеграции: статический анализ, в частности линтинг, и то, как его применять к унаследованному коду.
- Главы 5 и 6 посвящены тестированию — важнейшему элементу проверки в рамках непрерывной интеграции. Мы не будем учить вас тестировать (этому посвящено множество других книг), а сосредоточимся на самых частых проблемах, которые со временем накапливаются в наборах тестов, в частности на нестабильных и медленных тестах.
- В главе 7 рассматривается жизненный цикл изменений кода и исследуются все места, где могут возникнуть баги, а также способы настройки автоматизации для обнаружения и устранения этих багов по мере их появления.

В части 3 мы переходим от верификации изменений, вносимых в продукты с помощью непрерывной интеграции, к выпуску релизов:

- В главе 8 через призму метрик DORA рассматривается, как использование системы контроля версий влияет на скорость выпуска релизов.
- В главе 9 продемонстрировано, как создавать артефакты безопасно, применяя принципы, определенные стандартом SLSA, и объясняется важность версионирования.
- В главе 10 мы возвращаемся к метрикам DORA, основное внимание уделяя метрикам стабильности, и рассматриваем методы развертывания, которые можно использовать для повышения стабильности продукта.

В части 4 мы рассмотрим концепции, относящиеся к автоматизации непрерывной доставки в целом:

- В главе 11 мы вернемся к элементам непрерывной доставки, которые изучались в предыдущих главах, и рассмотрим, как эффективно применять эти элементы в проектах, реализуемых с нуля, и в унаследованных проектах.
- В главе 12 основное внимание уделено «рабочей лошадке», часто лежащей в основе любой автоматизации непрерывной доставки: shell-скриптам. Вы увидите, как применять к скриптам, обеспечивающим безопасную и корректную доставку продукта, те же лучшие практики, что и к остальной части кода.
- В главе 13 представлена общая структура автоматизированных пайплайнов, которые необходимы для поддержки непрерывной доставки, и моделируются функции систем автоматизации непрерывной доставки, обеспечивающие их эффективность.

В приложениях в конце книги описаны особенности наиболее распространенных на момент написания книги систем непрерывной доставки и контроля версий.

Я рекомендую начать с чтения главы 1; такие термины, как *непрерывная доставка*, в реальной жизни используются непоследовательно, и понимание их контекста для настоящей книги поможет вам лучше разобраться в остальных главах.

Удобнее всего осваивать материал, читая части 2 и 3 по порядку, поскольку все последующие главы основываются одна на другой. В частности, предполагается, что при переходе к части 3 вам известны практики непрерывной интеграции, описанные в части 2. Тем не менее вы можете изучать главы по своему усмотрению, так как в каждой главе вы найдете ссылки на соответствующий материал других глав.

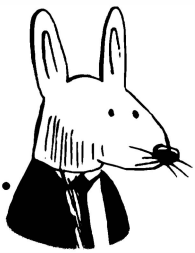
Часть 4 — это продвинутый раздел книги. Каждая глава в ней ссылается на концепции, рассмотренные ранее, а некоторые материалы (например, глава 12) полезнее будет изучить, уже приобретя некоторый опыт работы с системами непрерывной доставки.

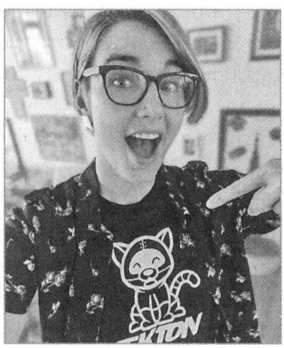
Форум liveBook

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

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

Об авторе





Кристи Уилсон (Christie Wilson) — инженер-разработчик программного обеспечения. Она часто выступает с докладами по непрерывной интеграции и непрерывной доставке (CI/CD) и смежным тематикам на таких конференциях, как Kubecon, OSCON, QCon, PyCon и многих других. Кристи начала карьерный путь с разработки мобильных веб-приложений, работая над бэкенд-сервисами для AAA-игр, где она писала отложенные функции, которые начинали использоваться все вместе во время большого запуска. Для этого она создала системы нагрузочного и системного тестирования.

Имея опыт, полученный в компаниях, которые работали со сложными средами развертывания, системами с высоким уровнем критичности и скачкообразным трафиком, она перешла на работу в Google, где создавала внутренние инструменты для повышения производительности AppEngine, запустила Knative и участвовала в создании Tekton, облачной платформы CI/CD на основе Kubernetes (в настоящее время ее используют более 65 компаний).

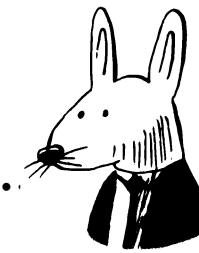
От издательства

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

Мы будем рады узнать ваше мнение!

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

Часть 1
Что такое непрерывная доставка

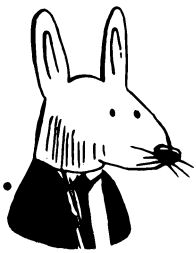


Добро пожаловать в «Грокаем Continuous Delivery»! Первые две главы познакомят вас с принципом непрерывной доставки (CD, Continuous Delivery) и терминологией, необходимой для освоения остальной части книги.

Глава 1 дает определение *непрерывной доставки* и объясняет ее связь со смежными понятиями, такими как *непрерывная интеграция* и *непрерывное развертывание*.

В главе 2 представлены основные элементы автоматизации процесса непрерывной доставки, а также терминология, используемая в остальной части книги.

1 Добро пожаловать в «Грокаем Continuous Delivery»



В этой главе

- ✓ Почему важна непрерывная доставка
- ✓ История развития технологий непрерывной доставки, непрерывной интеграции, непрерывного развертывания и CI/CD
- ✓ К какому ПО можно применять технологию непрерывной доставки и как это правильно делать
- ✓ Условия непрерывной доставки, а именно: поддерживать ПО всегда готовым к доставке и осуществлять его доставку как можно проще

Приветствую вас на страницах моей книги! Я очень рада, что вы решили не только узнать о непрерывной доставке, но и как следует разобраться в ней. Эта книга поможет вам использовать все преимущества непрерывной доставки в своей работе.

Нужна ли вам непрерывная доставка

Первый вопрос, который вы, возможно, себе задаете, — стоит ли тратить время на изучение непрерывной доставки, и даже если да, стоит ли применять ее в проектах. Ответ — *сто́ит*, если:

- вы профессионально занимаетесь разработкой программного обеспечения;
- в вашем проекте участвует больше одного человека.

Если справедливы оба этих условия, вам непременно следует вложиться в непрерывную доставку. *Даже если выполняется только одно из них* (например, вы работаете над проектом с друзьями ради интереса или разрабатываете профессиональное ПО в одиночку), вы не пожалеете об использовании непрерывной доставки.

> *Но ведь вы даже не спросили, чем я занимаюсь. Что, если я работаю над драйверами ядра, прошивкой или микросервисами? Вы уверены, что мне нужна непрерывная доставка? — спросите вы.*

Это неважно! Какой бы продукт вы ни создавали, принципы, изложенные в этой книге, будут вам полезны. В их основе лежат идеи, которые сформировались еще на заре программной разработки. Это не модные тренды, которые со временем исчезают или теряют популярность; это основы, которые останутся неизменными, независимо от того, создаете ли вы микросервисы, монолитные приложения, распределенные сервисы на основе контейнеров или что-то еще.

> ***Но мне не нужно ничего развертывать!***
>
> Совершенно верно! Развертывание и связанная с ним автоматизация применимы не ко всем видам ПО, однако непрерывная доставка — это гораздо больше, чем просто развертывание. Мы поговорим об этом позже в этой главе.

В этой книге рассматриваются основные принципы непрерывной доставки и приводятся примеры их практического использования. Конкретные детали реализации в проекте наверняка будут уникальными, и возможно, вы не найдете здесь их точного описания, но что вы точно найдете, так это компоненты, необходимые для автоматизации непрерывной доставки, и принципы, соблюдая которые вы добьетесь наибольшего успеха.

Зачем нужна непрерывная доставка

Итак, что же это за штука такая? Начну с того, что непрерывная доставка (CD) означает для меня и почему я считаю ее такой важной:

Непрерывная доставка — это один из процессов современной профессиональной программной разработки.

Разберем это определение по частям:

- *Современная* — профессиональная разработка существует гораздо дольше, чем непрерывная доставка, хотя ребята, которые работали с перфокартами, были бы в восторге от CD! Одна из причин, по которой мы можем сейчас использовать непрерывную доставку, а тогда не могли, заключается в том, что CD отнимает много ресурсов процессора. Непрерывная доставка требует выполнения большого объема кода!

> Я даже не могу себе представить, сколько перфокарт потребовалось бы для описания типичного рабочего процесса CD!

- *Профессиональная* — если вы пишете программы для интереса, то вопрос о том, стоит ли вам возиться с непрерывной доставкой, остается открытым. Как правило, ее используют, когда действительно важно, чтобы программный продукт работал. Чем важнее создаваемое ПО, тем тщательнее следует продумывать непрерывную доставку. Кроме того, говоря о профессиональной разработке, мы вряд ли будем иметь в виду одного программиста, самостоятельно пишущего код. Обычно над продуктом работает несколько человек, а иногда даже сотни.
- *Программная разработка* — в других инженерных областях существуют своды стандартов и сертификатов, которые в разаботке, как правило, отсутствуют. Итак, проще говоря, программная разработка — это написание программ. Когда мы добавляем слово «*профессиональная*», мы имеем в виду профессиональное занятие разработкой ПО.
- *Процесс* — профессиональная разработка требует соблюдения определенных подходов, чтобы написанный нами код делал то, что мы задумали. Эти процессы касаются не столько способов, которыми отдельно взятый программист пишет код (хотя и это важно), сколько способности этого человека сотрудничать с другими разработчиками, чтобы создавать продукт профессионального уровня.

Непрерывная доставка — это совокупность процессов, которые необходимы группе профессиональных разработчиков, чтобы создавать программный продукт, соответствующий исходным целям его создателей.

Подождите, вы хотите сказать, что CD означает «непрерывная доставка»? Я думал, что это непрерывное развертывание!

Некоторые действительно расшифровывают эту аббревиатуру именно так (continuous deployment), и тот факт, что оба термина появились примерно в одно и то же время, вносит большую путаницу. В большинстве известных мне публикаций (не говоря уже о сообществе Continuous Delivery Foundation!) авторы предпочитают употреблять аббревиатуру CD для обозначения непрерывной доставки, поэтому в этой книге мы будем делать то же самое.

Непрерывная доставка

Непрерывная доставка — это совокупность процессов, которые необходимы группе профессиональных разработчиков, чтобы создавать программный продукт, соответствующий исходным целям его создателей.

Мое определение отражает то, что я считаю действительно крутым функционалом CD, но оно слишком далеко от стандартной формулировки, которую вы можете встретить. Посмотрим на определение, данное организацией Continuous Delivery Foundation (CDF) (http://mng.bz/YGXN):

Практика программной разработки, при которой команды выпускают релизы изменений ПО для пользователей безопасным, быстрым и устойчивым образом благодаря:

- **способности осуществлять выпуски в любое время;**
- **автоматизации выпуска релизов.**

Заметьте, что CD подразумевает два основных условия. Вы занимаетесь непрерывной доставкой, если:

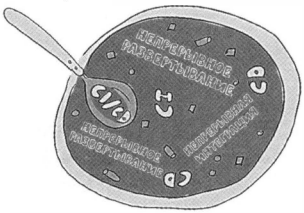
- выпускаете релизы ПО безопасно и в любое время;
- делаете выпуск максимально просто, буквально одним нажатием кнопки.

В этой книге подробно описаны необходимые действия и средства автоматизации, которые помогут вам добиться этих двух целей, а именно:

- Чтобы выпускать изменения безопасно и в любое время, ПО должно всегда находиться в состоянии готовности к релизу (в состоянии

Важным принципиальным изменением, произошедшим в CD по сравнению с CI, является переопределение значения фразы «функция готова». В CD «готова» означает «выпущена» (то есть осуществлен релиз). А процесс перехода от реализации изменений к выпуску автоматизирован, прост и быстр.

История «непрерывных» терминов



- **1994:** *термин «Непрерывная интеграция» (Continuous integration) вводится в книге Грейди Буча (Grady Booch) Object-Oriented Analysis and Design with Applications* [1] *(Addison-Wesley).*
- **1999:** *методика непрерывной интеграции впервые описана в книге Кента Бека (Kent Beck) Extreme Programming Explained*[2] *(Addison-Wesley).*
- **2007:** *дальнейшее развитие методика непрерывной интеграции получила в книге Пола Дюваля (Paul M. Duvall) Continuous Integration*[3] *(Addison-Wesley).*
- **2007:** *термин «непрерывное развертывание» определен в той же книге Дюваля.*
- **2009:** *принципы непрерывного развертывания популярно описаны в блоге Тимоти Фитца (Timothy Fitz) (http://mng.bz/2nmw).*
- **2010:** *методика непрерывной доставки, вдохновленная agile-манифестом*[4]*, описана в книге Continuous delivery*[5] *Джеза Хамбла (Jez Humble) и Дэвида Фарли (David Farley) (Addison-Wesley).*
- **2014:** *первая статья, дающая определение технологии CI/CD, Test Automation and Continuous Integration & Deployment (CI/CD) («Автоматизация тестирования и непрерывная интеграция и развертывание (CI/CD)»), опубликована сообществом Ravello (http://mng.bz/1opR).*
- **2016:** *в Википедию добавлена статья «CI/CD» (http://mng.bz/J2RQ).*

Вы, должно быть, думаете: «О'кей, Кристи, это все хорошо и замечательно, но что же на самом деле означает *доставлять*»? А *непрерывное развертывание*? И что такое *CI/CD*?

Мы действительно вынуждены оперировать множеством терминов! И что еще хуже, разные люди используют их по-разному. В их защиту можно сказать одно: это происходит потому, что некоторые из таких терминов даже не имеют определений!

Остановимся вкратце на эволюции этих терминов, чтобы лучше понять их значение. Непрерывная интеграция, непрерывная доставка и непрерывное развертывание — это термины, которые были созданы намеренно (или, в случае непрерывной интеграции, эволюционировали), а их авторы придавали им совершенно конкретные значения.

CI/CD — особый случай: похоже, этот термин никто не создавал и он появился на свет только потому, что люди, говорившие одновременно обо всех «непрерывных» операциях, нуждались в кратком термине (при этом полная аббревиатура CI/CD/CD почему-то не прижилась!).

Термин CI/CD в том виде, в каком он используется сегодня, означает инструменты и средства автоматизации, применяемые во всех непрерывных операциях — интеграции, доставке и развертывании.

---

[1] Буч Г. и др. «Объектно-ориентированный анализ и проектирование с примерами приложений».

[2] Бек К. «Экстремальное программирование». Санкт-Петербург, издательство «Питер».

[3] Дюваль Пол М. и др. «Непрерывная интеграция: улучшение качества программного обеспечения и снижение риска».

[4] Манифест разработки программного обеспечения по методологии agile. — *Примеч. пер.*

[5] Хамбл Д., Фарли Д. «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий».

релиза). В этом нам поможет непрерывная интеграция (continuous integration, CI).

- После проверки изменений методами CI процесс выпуска изменений должен проходить автоматически и его должно быть легко повторить.

Прежде чем я начну подробно рассказывать о том, как достичь этих целей, разберемся в терминологии.

Непрерывная доставка — это набор целей, к которым мы стремимся; способ их достижения может меняться от проекта к проекту. Тем не менее достигать этих целей наиболее эффективно помогает определенная последовательность действий, и именно им посвящена моя книга!

Интеграция

Непрерывная интеграция (Continuous Integration, CI) — старейшее из понятий, с которыми мы познакомились, однако это по-прежнему ключевой элемент непрерывной доставки. Начнем с простого — рассмотрим пока только интеграцию.

Что значит *интегрировать* программу? На самом деле часть фразы опущена: интегрировать объект нужно во что-то другое. В разработке интегрируемый объект — это изменение программного кода. Когда мы говорим об интеграции программного обеспечения, фактически мы имеем в виду

интеграцию изменений кода в существующее программное обеспечение.

Это именно то, чем в основном каждый день занимаются разработчики: изменяют код существующих частей программы. Этот процесс особенно интересен в команде: ее члены постоянно вносят изменения в код, и зачастую в одну и ту же часть продукта. Объединение этих изменений в единое целое и есть *интеграция*.

Иногда вам приходится создавать программу с нуля, но после первой успешной компиляции вы раз за разом будете интегрировать новые изменения в уже существующий программный продукт.

Интеграция программного обеспечения — это объединение изменений кода, сделанных несколькими людьми.

Как вы, вероятно, знаете по своему опыту, иногда этот процесс действительно идет криво. Например, когда я изменяю ту же строку кода, что и вы, и мы пытаемся объединить наши изменения, возникает конфликт и нам приходится вручную решать, как их интегрировать.

И еще одна маленькая деталь. Когда мы интегрируем изменения кода, мы не только просто соединяем их вместе; *мы еще и проверяем работоспособность измененного кода.* Можно сказать, что в аббревиатуре CI не хватает буквы *V* (Verification) —

верификация! Верификация уже заложена в процесс интеграции, поэтому, говоря об интеграции ПО, мы имеем в виду следующее:

Интеграция программного обеспечения — это объединение изменений кода, сделанных несколькими людьми, и проверка того, что получившийся код делает именно то, для чего он предназначен.

Кого волнуют все эти определения? Покажите мне уже наконец код!

Трудно выполнять свою работу последовательно и методично, если ей даже нельзя дать четкое определение. Уделить время тому, чтобы прийти к общему пониманию (через определения), а затем вернуться к основным принципам — это самый эффективный способ подняться на новый уровень!

Непрерывная интеграция

Посмотрим, как придать *интеграции* «*непрерывность*», на примере, не связанном с разработкой. Холли, шеф-повар, готовит соус для макарон. Она начинает с подбора продуктов: лука, чеснока, помидоров, специй. Чтобы приготовить соус, ей нужно *интегрировать* эти продукты вместе в правильном порядке и в нужном количестве.

Для этого каждый раз, когда она добавляет новый ингредиент, *она быстро пробует соус на вкус*. Исходя из вкуса, она может добавить больше какого-то продукта либо понимает, что забыла какой-то нужный ингредиент.

Дегустация помогает ей менять блюдо, проводя его через серию интеграций. Интеграция в данном случае подразумевает две вещи:

- объединение ингредиентов;
- проверку для подтверждения результата.

И это именно то, что означает слово «*интеграция*» в словосочетании *непрерывная интеграция*: объединение изменений кода, а также проверка его работоспособности, то есть объединение и верификация.

Холли повторяет эти шаги во время готовки. Если бы она попробовала соус только в конце, она гораздо меньше контролировала бы процесс и было бы уже поздно вносить необходимые изменения. Именно здесь вступает в дело «*непрерывность*». Вы должны интегрировать (объединять и верифицировать) свои изменения так часто, как только можете, и делать это как можно быстрее.

А как часто вы можете объединять и верифицировать программный продукт? Каждый раз, как только вы внесете изменения!

Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.

Объединение изменений кода означает, что разработчики, использующие непрерывную интеграцию, фиксируют и переносят код в общую систему контроля версий каждый раз, когда вносят изменения, и проверяют их корректную работу, применяя автоматические средства верификации, такие как тесты и линтинг[1].

Автоматическая верификация? Линтинг? Не волнуйтесь, если вы не знаете, что это, эта книга поможет вам разобраться! Позже мы рассмотрим, как создавать автоматические проверки, благодаря которым работает непрерывная интеграция.

Что мы доставляем

От непрерывной интеграции мы переходим к непрерывной доставке, и для этого нужно вернуться немного назад. Почти в каждом определении, которое мы рассматриваем, упоминается доставка какой-то программы (например, я собираюсь начать рассказ об интеграции и доставке изменений в программу). Неплохо бы убедиться, что мы все имеем в виду одно и то же, когда говорим о программах (software), ведь в зависимости от проекта этот термин может обозначать совершенно разные вещи.

Словарик

Термин «*программное обеспечение*» (software) возник в противоположность термину «*аппаратное обеспечение*» (hardware). Аппаратное обеспечение — это собственно физические части компьютеров. Мы производим действия с этими частями, снабжая их инструкциями. Инструкции могут быть встроены непосредственно в аппаратное обеспечение или переданы ему во время работы с помощью программного обеспечения.

Доставляемые программы могут иметь различные формы (интеграция и доставка для каждой из них тоже будут проходить по-разному):

- *Библиотека* — если программа ничего не делает сама по себе, а предназначена только для использования в составе другого ПО, то это, скорее всего, библиотека.
- *Двоичный файл* — если программа предназначена для выполнения какой-то задачи, вероятно, это двоичный исполняемый файл. Это может быть сервис, или приложение, или инструмент, который выполняет определенные действия и затем завершает работу, или приложение для мобильного устройства, например планшета или телефона.
- *Конфигурация* — это понятие относится к информации, которую можно передать в двоичный файл, чтобы изменить его работу без перекомпилирования.

[1] Линтинг — проверка кода на соответствие стандартам. — *Примеч. пер.*

Как правило, оно обозначает средства, доступные системному администратору для внесения изменений в работающее ПО.

- *Образ* — образы контейнеров представляют собой особый тип двоичных файлов, ставший чрезвычайно популярным форматом совместного использования и распространения сервисов вместе с их конфигурацией, чтобы они могли выполняться независимо от операционной системы.
- *Сервис* — как правило, сервисы — это двоичные файлы, которые постоянно находятся в рабочем состоянии, ожидая запросов, на которые они отвечают, выполняя какие-либо действия или возвращая информацию. Иногда их также называют *приложениями*.

На разных этапах карьеры вы можете работать с отдельными видами программ или со всеми сразу. Но вне зависимости от того, с какой программой вы будете иметь дело, для ее создания вам потребуется осуществлять *интеграцию* и *доставку* вносимых в нее изменений.

Доставка

Что означает *доставить* изменения в программу, зависит от того, какой программный продукт вы создаете, кто его использует и как. Обычно доставка изменений имеет отношение либо только к одному из процессов сборки, релиза и развертывания ПО, либо сразу ко всем:

- *Сборка* — ряд действий, направленных на получение кода (включая его изменения) и преобразование его в пригодную для использования форму. Обычно это означает компиляцию кода, написанного на языке программирования, в машинный язык. Иногда это также означает помещение кода в пакет, например в образ контейнера или похожий, который менеджер пакетов сможет распознать (например, в пакет PyPI для Python).
- *Публикация* — копирование программного обеспечения в репозиторий (место хранения программ), например, путем загрузки образа или библиотеки в реестр пакетов.
- *Развертывание* — копирование ПО в место, где оно должно быть запущено, и приведение его в рабочее состояние.

Сборка осуществляется в рамках процесса интеграции, чтобы проверить, что все внесенные изменения корректно работают вместе.

Можно проводить развертывание без выпуска релиза: например, развернуть новую версию программы, но не направлять на нее трафик. Однако чаще всего развертывание все же предполагает релиз; все зависит от того, где оно происходит. Если вы развертываете ПО в производственной среде (в продакшене), то выпуск релиза происходит одновременно с развертыванием. Подробнее о развертывании см. в главе 10.

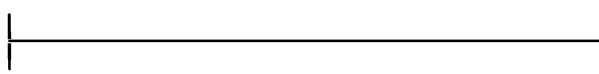
- *Выпуск релиза* — предоставление продукта пользователям. Релиз можно выпускать путем загрузки образа или библиотеки в репозиторий или установки параметров конфигурации для направления определенного процента трафика на развернутый экземпляр программы.

Словарик

Мы осуществляем *сборку* программного обеспечения с тех пор, как у нас появились языки программирования. Это настолько распространенный процесс, что первые системы, которые выполняли то, что мы сейчас называем *непрерывной доставкой*, именовались *системами сборки* (build systems). Этот термин стал таким привычным, что даже сегодня вы слышите слово *сборка* (или *билд*), хотя при этом имеется в виду последовательность задач в пайплайне развертывания, которая выполняется для преобразования ПО (подробнее об этом см. в главе 2).

Непрерывная доставка/непрерывное развертывание

Теперь вы знаете, что значит доставлять изменения в программное обеспечение, но что значит делать это непрерывно? В контексте CI мы узнали, что «*непрерывно*» означает «*как можно быстрее*». Так ли это в случае CD? И да и нет. Непрерывность в CD лучше представить в виде диапазона:



Возможность выпускать релизы безопасно и в любое время

Безопасный релиз при каждом изменении

Создаваемое вами ПО, должно находиться в состоянии, при котором в любое время можно выполнить его сборку, релиз и/или развертывание. Но как часто вы решите доставлять это ПО, зависит только от вас.

2009: статья в блоге, описывающая принципы непрерывного развертывания

2010: методика непрерывной доставки описана в книге с аналогичным названием

«А как насчет непрерывного развертывания?» — спросите вы. Отличный вопрос. Если снова обратиться к истории, то можно заметить, что эти два термина, *непрерывная доставка* и *непрерывное развертывание*, получили широкое распространение практически одновременно. Что же происходило в то время, когда эти термины вошли в оборот?

Это был переломный этап в разработке программного обеспечения: старые способы создания программных продуктов, когда все зависело от людей и ручных операций, жесткое разделение разработки и эксплуатации программ (интересно, что термин *DevOps* (development&operations, разработка и эксплуатация) появился примерно в то же время) и четко определенные процессы разработки ПО (например, *этап тестирования*) — все это начало меняться (со сдвигом влево). Непрерывное развертывание и непрерывная доставка совместно легли в основу практик, появившихся в это время. *Непрерывное развертывание* означает, что:

выпуск релиза рабочей версии ПО для пользователей осуществляется автоматически при каждом коммите.

Словарик

Сдвиг влево — это процесс поиска дефектов на начальных этапах создания программного продукта.

Непрерывное развертывание — это дополнительный этап непрерывной доставки. Непрерывная доставка позволяет осуществлять также и непрерывное развертывание. Постоянное пребывание ПО в состоянии готовности к релизу и автоматизация процесса доставки освобождают вас от необходимости выбирать, что будет лучше для проекта.

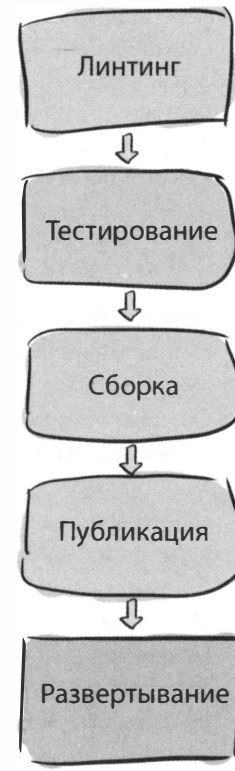
Если непрерывное развертывание на самом деле относится к выпуску релизов, почему бы не назвать его непрерывным релизом?

Отличная мысль! *Непрерывный релиз* — более точное название, и оно прояснило бы, как эту методику можно применять к ПО, которое не нужно развертывать. Однако прижился именно термин «*непрерывное развертывание*». Пример непрерывного релиза см. в главе 9.

Элементы непрерывной доставки

Остальная часть этой книги посвящена описанию основных составляющих процесса CD:

> **Методика программной разработки, при которой релиз рабочего программного продукта для пользователей осуществляется так быстро, как того требует проект, а его сборка производится таким образом, чтобы релизы можно было гарантированно безопасно выпускать в любое время.**



Вы узнаете, как использовать CI, чтобы ваш продукт всегда был готов к релизу, и как сделать доставку автоматической и повторяемой. Это сочетание позволяет вам выбрать, пойти ли на крайние меры и осуществлять релиз продукта при каждом изменении (непрерывное развертывание) или же предпочесть выпуск релизов в другом режиме. В любом случае вы можете быть уверены, что весь процесс автоматизирован, что позволит осуществлять доставку с необходимой частотой.

В основе такой автоматизации будет лежать пайплайн непрерывной доставки. В этой книге я подробно расскажу, что собой представляет каждая из этих задач. Вы обнаружите, что вне зависимости от того, какой программный продукт вы создаете, многие из них будут вам полезны.

> *Пайплайн? Задача? Что это?*
>
> Чтобы узнать, читайте следующую главу!

В следующей таблице перечислены виды программ, которые мы рассмотрели выше, и указано, что включает доставка каждого из них.

	Доставка включает сборку?	Доставка включает публикацию?	Доставка включает развертывание?	Доставка включает релиз?
Библиотека	Зависит от проекта	Да	Нет	Да
Двоичный файл	Да	Обычно	Зависит от проекта	Да
Конфигурация	Нет	Скорее всего, нет	Обычно	Да
Образ контейнера	Да	Да	Зависит от проекта	Да
Сервис	Да	Обычно	Да	Да

Заключение

В сфере непрерывной доставки существует множество терминов и противоречивых определений. В этой книге мы используем аббревиатуру CD для обозначения *непрерывной доставки*, которая включает в себя непрерывную интеграцию (CI), развертывание и выпуск релизов. Основное внимание я уделю автоматизации, которая необходима для использования CD вне зависимости от типа доставляемого программного обеспечения.

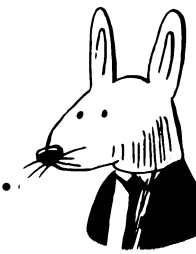
Итоги

- Непрерывная доставка полезна при разработке любого типа программ.
- Непрерывная доставка необходима, чтобы создавать профессиональные продукты, работая в команде.
- Осуществляя непрерывную доставку, используйте непрерывную интеграцию, чтобы гарантировать, что ваш продукт всегда находится в состоянии готовности к доставке.
- Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.
- Другим условием непрерывной доставки является автоматизация, позволяющая выпускать релизы максимально просто, буквально одним нажатием кнопки.
- Непрерывное развертывание — это дополнительный этап, который можно добавить по желанию, если это важно для проекта; в этом случае доставка ПО будет происходить автоматически при каждом коммите.

Далее...

Вы познакомитесь с основами автоматизации непрерывной доставки и терминологией, принятой в этой области, и тем самым подготовитесь к дальнейшему изучению материала.

2 Базовый пайплайн



В этой главе

- ✓ Работа с основными составляющими процесса CD: пайплайнами и задачами
- ✓ Элементы базового пайплайна CD: линтинг, тестирование, сборка, публикация и развертывание
- ✓ Роль автоматизации в выполнении пайплайнов: веб-хуки, события и запуск
- ✓ Знакомство с терминологией в области CD

Прежде чем перейти к конкретным деталям создания крутых пайплайнов непрерывной доставки (CD pipelines), узнаем, что вообще такое пайплайн. В этой главе мы в общих чертах рассмотрим несколько пайплайнов и определим основные элементы, которые должны присутствовать в большинстве пайплайнов CD.

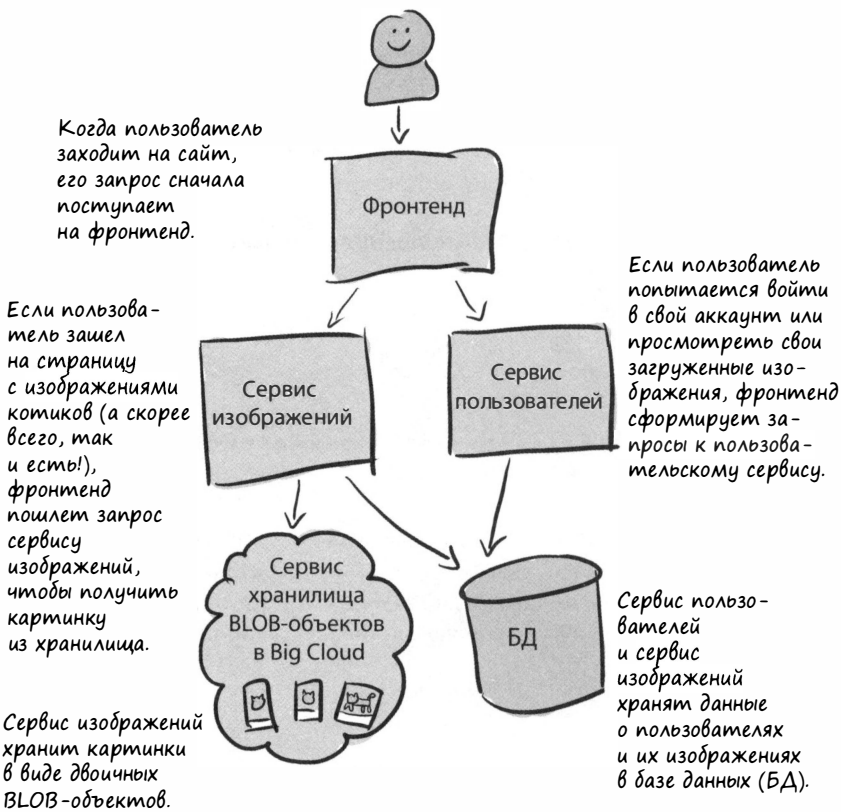
Веб-сайт Cat Picture

Чтобы понять, из чего состоят базовые пайплайны CD, рассмотрим сначала пайплайны, используемые для веб-сайта Cat Picture. Cat Picture — это лучший сайт для поиска и обмена фотографиями котиков! Он довольно прост, но, поскольку это популярный ресурс, управляющая им компания (Cat Picture, Inc.) создала его в виде нескольких сервисов.

Компания запускает сайт Cat Picture в облаке (ее облачный провайдер называется Big Cloud, Inc.) и использует сервисы Big Cloud, такие как Big Cloud Blob Storage (хранилище BLOB-объектов[1] Big Cloud).

> ***Что это опять за CD?***
>
> В этой книге аббревиатурой CD мы обозначаем непрерывную доставку. Подробнее см. главу 1.

> ***Что такое пайплайн?***
>
> Не волнуйтесь, об этом мы поговорим через пару страниц!



[1] BLOB (Binary Large Object) — массив двоичных данных. Здесь — специальный тип данных, предназначенный для хранения изображений. Хранилище BLOB-объектов оптимизировано для хранения больших объемов неструктурированных данных. — *Примеч. науч. ред.*

Исходный код сайта Cat Picture

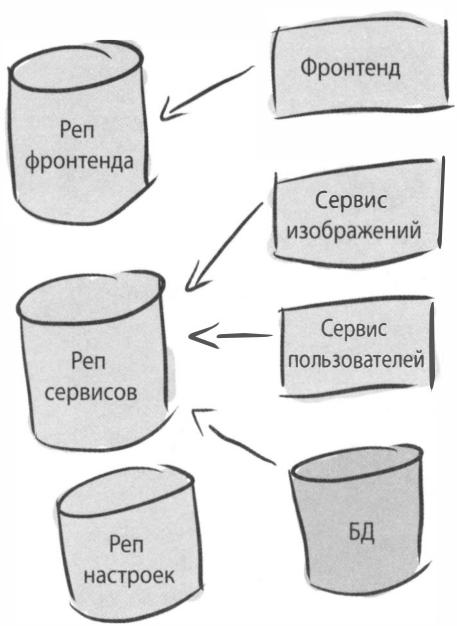
Схема архитектуры показывает, как устроен сайт Cat Picture, но для понимания пайплайна CD необходимо обсудить один важный момент: где находится код?

В главе 1 мы рассмотрели элементы CD, половина из которых связана с использованием непрерывной интеграции (CI) для обеспечения постоянной готовности ПО к релизу. Вспомним определение CI:

> **Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.**

Если проанализировать, что мы на самом деле делаем, осуществляя CD, то мы увидим, что в основном CD означает изменение кода. То есть входными данными для пайплайнов CD является исходный код. Фактически именно это и отличает пайплайны CD от других видов автоматизации рабочих процессов: пайплайны CD почти всегда используют исходный код в качестве входных данных.

Прежде чем рассматривать пайплайны CD сайта Cat Picture, нам нужно понять, как организован и где хранится его исходный код. Разработчики сайта Cat Picture хранят свой код в нескольких репозиториях (репах):

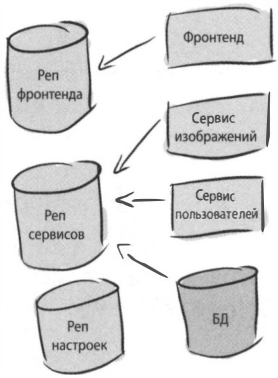


- В репозитории фронтенда хранится код внешнего интерфейса.
- Сервис изображений, сервис пользователей и схемы баз данных хранятся в репозитории сервисов.
- И наконец, сайт Cat Picture использует подход «конфигурация как код» для управления конфигурацией (подробнее об этом в главе 3), храня ее в репозитории настроек.

Организовать код сайта Cat Picture можно множеством других способов, у каждого из которых есть свои плюсы и минусы.

> ***Система контроля версий***
>
> Использование системы контроля версий, такой как Git, является необходимым условием для реализации CD. Без хранения кода вместе с историей и системой обнаружения конфликтов осуществлять CD практически невозможно. Подробнее об этом в главе 3.

Пайплайны сайта Cat Picture



Поскольку сайт Cat Picture состоит из нескольких сервисов, а весь код и конфигурация, необходимые для него, распределены по нескольким репозиториям, он управляется несколькими пайплайнами CD. Мы подробно рассмотрим все эти пайплайны в следующих главах, когда будем изучать более продвинутые их варианты, а пока остановимся на базовом пайплайне, который используется для сервиса пользователей и сервиса изображений.

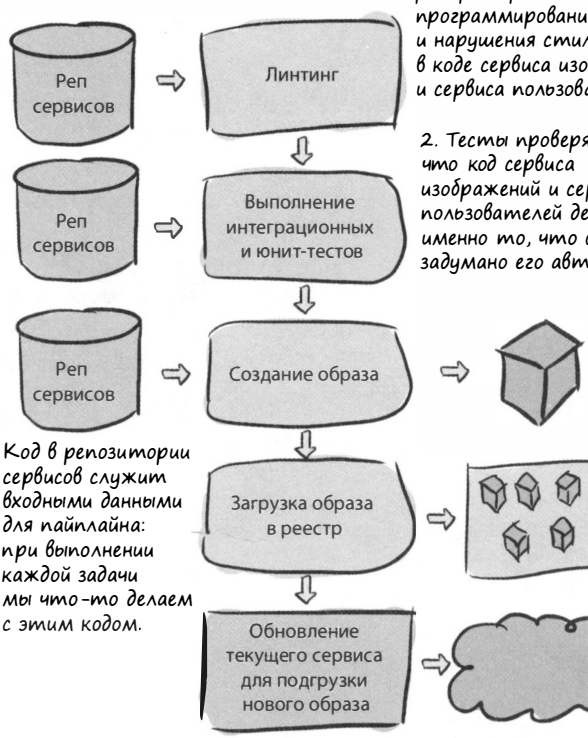
Так как эти два сервиса очень похожи, для них используется один и тот же пайплайн, и он содержит все основные элементы любого пайплайна.

Этот пайплайн не только используется для веб-сайта Cat Picture, но и содержит основные элементы пайплайнов, описанных далее в книге!

> Когда все это на самом деле выполняется? Об этом мы кратко поговорим чуть позже, а более подробно — в главе 10.

Словарик

Образы контейнеров — это исполняемые программные пакеты, содержащие все необходимое для работы программы.



1. Линтинг вылавливает распространенные ошибки программирования и нарушения стиля в коде сервиса изображений и сервиса пользователей.

2. Тесты проверяют, что код сервиса изображений и сервиса пользователей делает именно то, что было задумано его авторами.

3. После линтинга и тестирования кода создаются образы контейнеров для каждого из сервисов.

4. Образы контейнеров загружаются в реестр образов.

5. Наконец, рабочая версия ПО обновляется для использования нового образа.

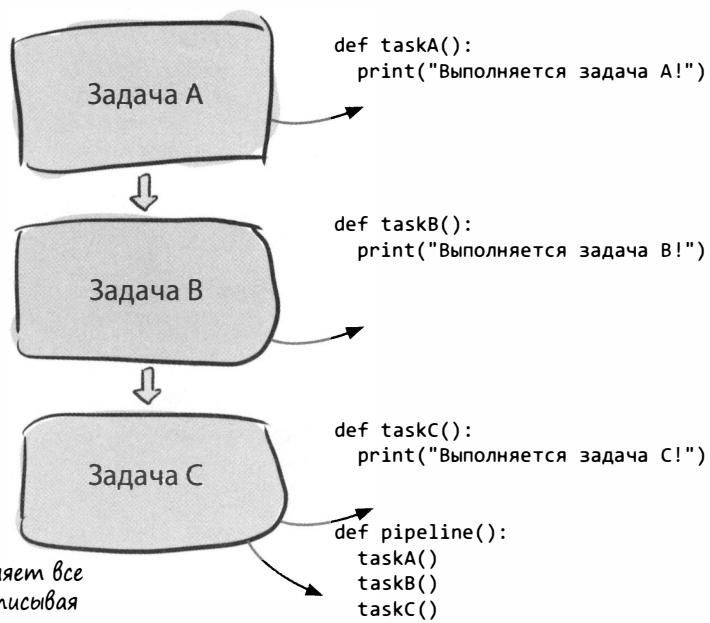
Что же такое пайплайн и что такое задача

На нескольких предыдущих страницах мы рассмотрели пайплайн веб-сайта Cat Picture, но что такое пайплайн вообще? В сфере CD существует множество терминов. Хотя здесь мы используем термин «*пайплайн*» (pipeline, конвейер), иногда используют и другие термины, например «*рабочий процесс*» (workflow). Мы подробнее обсудим терминологию в конце главы, а пока остановимся на пайплайнах и задачах.

Задачи (tasks) — это отдельные действия, которые вы можете выполнять; их можно представить себе как некие функции. А *пайплайн* — это как точка входа, дающая доступ к коду, который вызывает все функции в нужное время и в определенной последовательности.

Ниже приведен пример пайплайна в виде кода на языке Python, состоящий из трех задач: сначала выполняется задача A, затем задача B, а заканчивается пайплайн задачей C.

Каждая задача подобна функции.



Пайплайн объединяет все задачи вместе, описывая порядок их вызова.

Пайплайны CD запускаются снова и снова; о том, когда именно происходит их запуск, мы поговорим чуть позже. Если бы мы запустили функцию `pipeline()` (представляющую собой пайплайн, показанный на рисунке), то получили бы следующий результат:

```
Выполняется задача A!
Выполняется задача B!
Выполняется задача C!
```

Основные задачи в пайплайне CD

Пайплайн сайта Cat Picture включает все типичные задачи большинства пайплайнов. Мы подробно рассмотрим их в следующих главах. А сейчас узнаем, для чего нужна каждая из задач пайплайна Cat Picture:

- *Линтинг* выявляет распространенные ошибки программирования и нарушения стиля в коде сервиса изображений и сервиса пользователей.
- *Интеграционные и юнит-тесты* проверяют, что код сервиса изображений и код сервиса пользователей делает именно то, что было задумано его создателями.
- После линтинга и тестирования кода задача *создания образа* производит сборку образов контейнеров для каждого из сервисов.
- Далее *образы контейнеров загружаются* в реестр образов.
- И наконец, текущая версия программы *обновляется для подгрузки новых образов*.

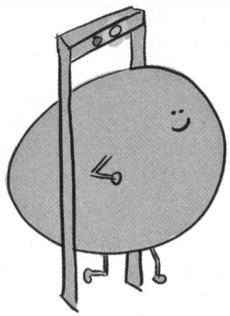
Каждая задача в пайплайне Cat Picture представляет собой базовый элемент пайплайна:

- *Линтинг* — наиболее распространенная форма статического анализа пайплайнов CD.
- Интеграционные и юнит-тесты — это формы *тестов*.
- Сервисы собраны в виде образов; для большинства программ их необходимо *собрать* в другом виде, прежде чем использовать.
- Образы контейнеров хранятся в реестрах и извлекаются из реестров; как следует из главы 1, некоторые виды программ необходимо *опубликовать*, прежде чем их можно будет использовать.
- Сайт Cat Picture должен быть доступен и находиться в рабочем состоянии, чтобы пользователи могли с ним взаимодействовать. Обновление работающего сервиса для использования нового образа — это своего рода *развертывание* веб-сайта.

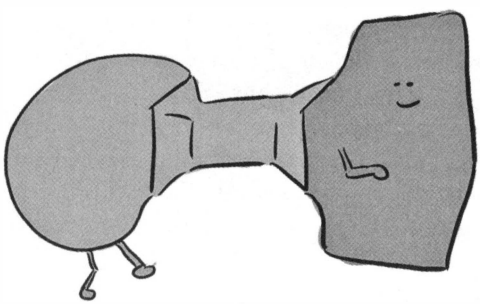
Вот основные типичные задачи пайплайна CI/CD:

Шлюзы и преобразования

Некоторые задачи связаны с *проверкой* кода. Это *шлюзы* проверки качества (система контроля качества, quality gates), через которые должен пройти код.



Другие задачи подразумевают трансформацию кода. Это *преобразования* кода: код поступает на вход в одной форме, а выходит в другой.



Представление задач в пайплайне CD как шлюзов и преобразований неразрывно связано с элементами CD. В главе 1 мы говорили о том, что CD необходима, чтобы:

- безопасно вносить изменения в ПО в любое время;
- осуществлять доставку ПО максимально просто, буквально нажатием кнопки.

Если приглядеться, можно заметить, что эти цели один в один соответствуют задачам шлюзов и преобразований:

- *Шлюзы* проверяют качество изменений кода, гарантируя безопасность их доставки.
- *Преобразования* осуществляют сборку, публикацию и, в зависимости от типа программы, развертывание изменений.

А по сути, шлюзы обычно составляют часть пайплайна, относящуюся к CI!

> CI — это проверка кода! Вы часто слышите, как кто-то говорит о «выполнении CI» или «сбое CI», и обычно в этом случае имеют в виду *шлюзы*.

CD: шлюзы и преобразования

Вернемся к основным задачам CD и посмотрим, как они связаны со шлюзами и преобразованиями:

- Код попадает на этап *тестирования* (на котором выполняется *шлюзовая задача*, gating task) и либо проходит тест, либо нет. Если тест не пройден, код не должен продолжать движение по пайплайну.
- Код переходит на этап *преобразования* и либо полностью трансформируется, либо с его помощью вносятся изменения в какую-то часть системы.

Основные задачи CD можно соотнести со шлюзами и преобразованиями следующим образом:

- *Линтинг* — это просмотр кода и выявление распространенных ошибок и багов, но без фактического выполнения кода. По мне, так это *шлюз*!
- *Тестирование* — это проверка правильности выполнения кода, то есть проверка того, делает ли код именно то, для чего был предназначен. Поскольку это еще один пример верификации кода, это тоже похоже на *шлюз*.
- *Сборка* кода — это процесс получения кода в одной форме и трансформации его в другую для удобства использования. Иногда эта операция позволяет выявить проблемы с кодом, поэтому она имеет признаки CI. Однако чтобы протестировать код, нам, вероятно, потребуется его собрать, поэтому основной целью здесь является *преобразование* (сборка) кода.
- *Публикация* кода — это размещение собранной программы в определенном месте, чтобы ее можно было использовать. Это часть релиза продукта (для некоторых видов кода, например библиотек, публикация и есть релиз!) и тоже своего рода *преобразование*.
- И наконец, *развертывание* кода (для программ, которые необходимо запустить и выполнить) — это своего рода *преобразование* состояния собранной версии программы.

Хорошо, допустим, шлюзы — это задачи CI. Вы хотите сказать, что CI — это только тесты и линтинг? Я помню, что до появления непрерывной доставки CI включала в себя и сборку.

Да, я вас поняла! Процесс CI действительно часто включает в себя сборку, а иногда и публикацию. Важно заранее определиться с содержанием этих процессов, поэтому в этой книге я предпочитаю относить к CI только верификацию, но не сборку/публикацию/развертывание/релиз.

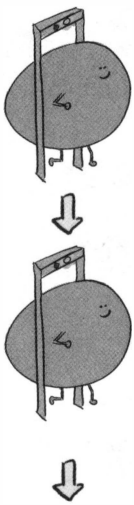
Пайплайн сервисов сайта Cat Picture

Как будет выглядеть пайплайн сервисов сайта Cat Picture, если рассмотреть его как пайплайн шлюзов и преобразований?

Первый шлюз, через который должен пройти код, — это *линтинг*. Если в коде на этом этапе возникают проблемы, не следует начинать преобразование кода и его доставку; сначала необходимо исправить ошибки.

Следующий шлюз — интеграционные и юнит-тесты. Как и в случае с линтингом, если они показывают, что результат выполнения кода не соответствует замыслу авторов, необходимо исправить ошибки и лишь затем начинать преобразование кода и его доставку.

Когда код прошел через все *шлюзы* и мы убедились, что в нем нет ошибок, можно приступать к его *преобразованию*.

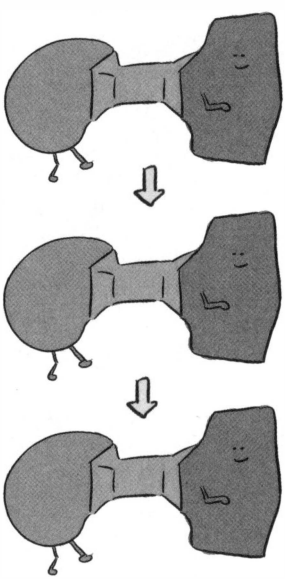


Первое преобразование — это *создание образа* из исходного кода. Код компилируется и упаковывается в образ контейнера, пригодный для выполнения.

В ходе следующего преобразования созданный образ *загружается в реестр образов*, превращаясь из образа на диске в образ в реестре, пригодный для загрузки и использования.

Последнее преобразование обновит запущенный сервис для работы с данным образом.

Готово!



Запуск пайплайна

Вам уже интересно, как и когда запускается пайплайн? Отлично! Команда Cat Picture, Inc. несколько раз меняла этот процесс.

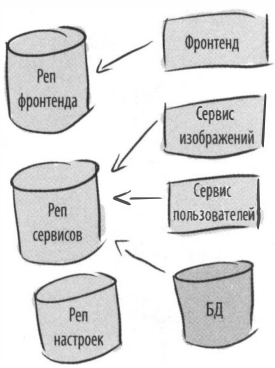
Когда компания Cat Picture, Inc. только начинала свою работу, у нее было всего несколько разработчиков: Тофер, Анжела и Сато. Анжела написала на Python пайплайн для сайта Cat Picture. Вот как он выглядит:

```
def pipeline(source_repo, config_repo):
  linting(source_repo)
  unit_and_integration_tests(source_repo)
  image = build_image(source_repo)
  image_url = upload_image_to_registry(image)
  update_running_service(image_url, config_repo)
```

Это упрощенный вариант кода, написанного Анжелой, но нам пока достаточно и его.

Функция `pipeline()` в этом коде выполняет каждую из задач сайта Cat Picture в виде функции.

Как линтинг, так и тестирование выполняется на исходном коде, и при создании образа сборка осуществляется из исходного кода. Выходные данные на каждом этапе преобразования (сборки, загрузки, обновления) передаются последовательно по мере их выполнения.



Замечательно, но как же запустить этот процесс? Кто-то (или, как мы скоро увидим, что-то) должен выполнить функцию `pipeline()`.

Тофер вызвался отвечать за запуск пайплайна, поэтому он написал исполняемый файл на Python, который выглядит так:

```
if __name__ == "__main__":
  pipeline("https://10.10.10.10/catpicturewebsite/service.git",
           "https://10.10.10.10/catpicturewebsite/config.git")
```

Этот исполняемый файл вызывает функцию `pipeline()`, передавая в качестве аргументов адреса Git-репозиториев: репозитория сервиса (Service repo) и репозитория настроек (Config repo). Тоферу остается только запустить исполняемый файл, который, в свою очередь, запустит пайплайн со всеми его задачами.

Обязательно ли писать пайплайны и задачи на Python, как Анжела и Тофер?

Вообще нет! Вам не нужно изобретать систему CD заново, достаточно выбрать один из уже существующих инструментов. В приложении А представлен краткий обзор некоторых имеющихся систем. Мы будем использовать язык Python для иллюстрации принципов работы этих систем, не ограничиваясь конкретным инструментом, а в дальнейшем мы также используем GitHub Actions. У всех систем CD свои преимущества и недостатки; выбирайте ту, которая лучше всего подойдет для вашего проекта.

Запуск один раз в сутки

Словарик

Фраза «*произошел сбой* пайплайна» (pipeline breaks) означает, что одна из задач пайплайна завершилась с ошибкой и его выполнение остановлено.

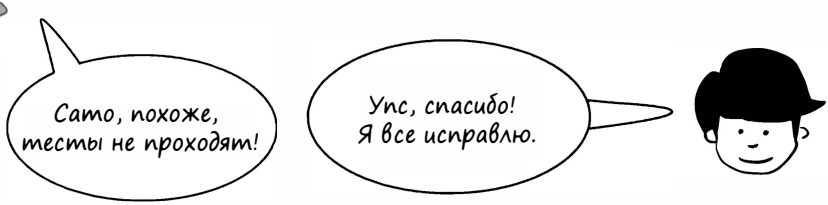
Тофер запускает пайплайн путем запуска исполняемого файла на языке Python:

```
def pipeline(source_repo, config_repo):
  linting(source_repo)
  unit_and_integration_tests(source_repo)
  image = build_image(source_repo)
  image_url = upload_image_to_registry(image)
  update_running_service(image_url, config_repo)

if __name__ == "__main__":
  pipeline("https://10.10.10.10/catpicturewebsite/service.git",
         "https://10.10.10.10/catpicturewebsite/config.git")
```

Когда Тофер будет его запускать? Он решил, что будет делать это каждое утро, в начале рабочего дня. Посмотрим, как это происходит:

Вторник, 10:00 утра

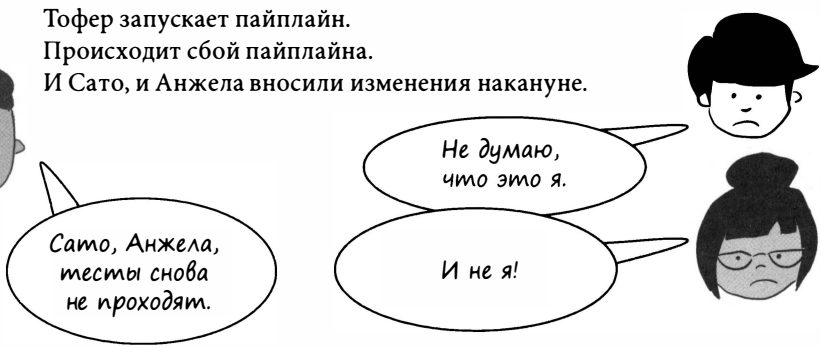
Тофер запускает пайплайн.
Происходит сбой пайплайна.
Тофер видит, что последнее изменение внес Сато.



Это сработало, но посмотрите, что произошло на следующий день.

Среда, 10:00 утра

Тофер запускает пайплайн.
Происходит сбой пайплайна.
И Сато, и Анжела вносили изменения накануне.



Все работает не так, как надеялся Тофер. Поскольку он запускает пайплайн один раз в сутки, он собирает все изменения, сделанные накануне. Когда что-то идет не так, он не может точно сказать, какое изменение вызвало проблему.

Пробуем использовать непрерывную интеграцию

Поскольку Тофер запускает пайплайн один раз в сутки, он собирает все изменения за предыдущий день. Если мы обратимся к определению CI, мы поймем, что именно идет не так:

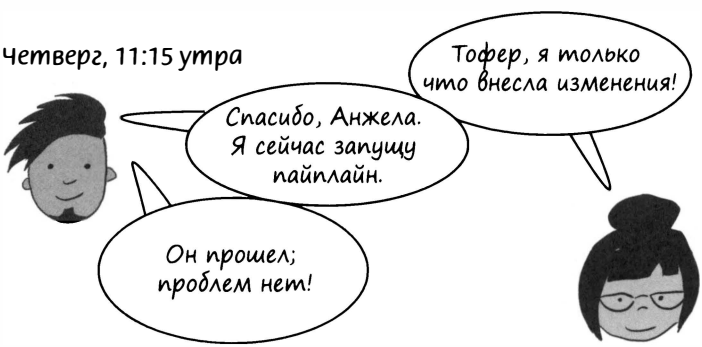
> **Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и *каждое изменение проверяется при внесении*.**

То есть Тоферу следует запускать пайплайн при каждом изменении. Таким образом, каждый раз, когда в код будут вноситься изменения, команда будет знать, вызвало ли это изменение проблемы.

Тофер просит членов команды сообщать ему каждый раз, когда они вносят изменения, чтобы он немедленно запускал пайплайн. Теперь пайплайн запускается при каждом изменении, и команда получает обратную связь сразу после внесения изменений.

Словарик

Когда говорят, что пайплайн *прошел* (pipeline passes), — это означает, что все прошло успешно, то есть *сбоев* не было.

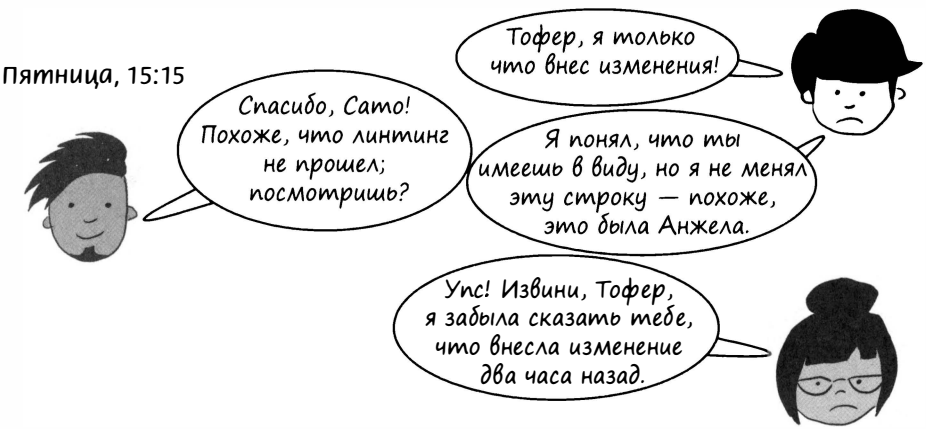


Непрерывное развертывание

Запуская весь пайплайн, включая задачи преобразования, Тофер осуществляет одновременно и непрерывное развертывание. Большинство разработчиков предпочитают выполнять задачи интеграции и преобразования в разных пайплайнах. В главе 13 мы рассмотрим компромиссы между этими подходами.

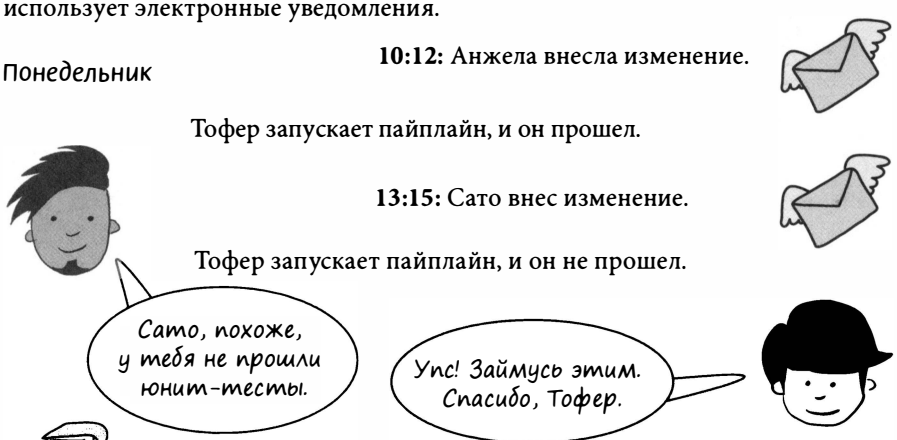
Использование уведомлений

Прошло несколько недель, и члены команды стали сообщать Тоферу о каждом изменении. Посмотрим, как это происходит:



И снова все пошло не так гладко, как надеялся Тофер. Анжела внесла изменения и забыла сообщить ему об этом, и теперь команде приходится возвращаться назад. Как Тоферу быть уверенным, что он не пропустит никаких изменений?

Тофер изучает проблему и понимает, что может получать уведомления от *системы контроля версий* каждый раз, когда кто-то вносит изменения. Теперь, вместо того чтобы просить сотрудников сообщать ему, когда они вносят изменения, он использует электронные уведомления.



Словарик

Управление контролем версий — термин, применимый для систем, подобных GitHub, которые сочетают систему контроля версий с дополнительными функциями, такими как инструменты анализа кода. Другие примеры — GitLab и Bitbucket. См. приложение B.

Масштабирование в ручном режиме

Дела у команды шли настолько хорошо, что в нее приняли еще двух человек. Давайте навестим Тофера?

Пятница

9:00: Анжела внесла изменение.

9:14: Сато внес изменение.

9:27: Ми-Джун внесла изменение.

10:03: Роберт внес изменение.

Тофер теперь весь день запускает пайплайн, и у него не остается времени на другую работу. У него масса идей, как улучшить пайплайн, и он хотел бы реализовать несколько функций, но не успевает этим заняться!

Он решает притормозить и подумать о том, что вообще происходит, чтобы найти способ сэкономить время:

1. Тоферу приходит электронное письмо.
2. Приложение электронной почты уведомляет его о новом письме.
3. Тофер видит уведомление.
4. Он запускает скрипт пайплайна.
5. И сообщает остальным, если происходит сбой.

Тофер анализирует свою роль в этом процессе. Что он должен сделать лично?

1. Увидеть уведомление электронной почты.
2. Ввести команду для запуска скрипта.
3. Сообщить остальным о результатах.

Можно ли исключить Тофера из процесса? Для этого ему понадобится что-то, что может делать следующее:

1. Просматривать уведомления.
2. Запускать скрипт пайплайна.
3. Сообщать остальным о результатах.

Тоферу нужно найти то, что сможет получать уведомления и запускать скрипт вместо него.

Автоматизация посредством веб-хуков

Время — деньги! Тофер понимает, что весь его день уходит на запуск пайплайна, но может исключить себя из процесса, если найдет инструменты для выполнения следующих задач:

1. Просмотр уведомлений.
2. Запуск скрипта пайплайна.
3. Оповещение сотрудников о результатах.

Словарик

Используйте *веб-хуки* (webhooks), чтобы система, которую вам не надо контролировать, запускала код при наступлении определенного события. Обычно это реализуется путем предоставления системе URL-адреса конечной точки HTTP, которой вы управляете.

Тофер исследует этот вопрос и обнаруживает, что система контроля версий поддерживает веб-хуки. Написав простой веб-сервер, он может сделать все, что нужно:

1. Система контроля версий будет отправлять запрос на его веб-сервер каждый раз, когда кто-то вносит изменения. (Тоферу не нужно видеть это уведомление!)
2. Когда веб-сервер получает запрос, он сам запускает скрипт пайплайна. (Тоферу не нужно этого делать!)
3. Запрос, который система отправляет на веб-сервер, содержит адрес электронной почты того, кто внес изменение, поэтому в случае ошибки выполнения скрипта пайплайна веб-сервер отправляет электронное письмо человеку, сделавшему ошибку.

```
class Webhook(BaseHTTPRequestHandler):
  def do_POST(self):
    respond(self)
    email = get_email_from_request(self)
    success, logs = run_pipeline()
    if not success:
      send_email(email, logs)

if __name__ == '__main__':
  httpd = HTTPServer(('', 8080), Webhook)
  httpd.serve_forever()
```

Тофер запускает веб-сервер на своей рабочей станции — и работа пайплайна автоматизирована!

Как получить доступ к уведомлениям и событиям системы контроля версий?

Вам придется обратиться к документации по своей системе контроля версий, чтобы узнать, как ее настроить, но уведомления об изменениях и запуск веб-хуков — основные функции большинства таких систем. Если в вашей системе их нет, переходите на другую! (Некоторые доступные варианты перечислены в приложении B.)

Масштабирование при наличии веб-хуков

Посмотрим, что происходит теперь, когда Тофер автоматизировал процесс выполнения пайплайна с помощью веб-хука:

Понедельник

9:14: Анжела внесла изменение.

Система контроля версий активирует веб-хук Тофера.

Веб-хук Тофера запускает пайплайн: происходит сбой!

Веб-хук Тофера отправляет электронное письмо Анжеле, сообщая ей, что ее изменение привело к сбою пайплайна.

События системы контроля версий и веб-хуки берут на себя всю работу, которую раньше выполнял Тофер. Теперь он может спокойно заниматься тем, чем действительно хочет!

Словарик

Вызов веб-хука системой контроля версий при наступлении определенного события часто называют *запуском, или срабатыванием* (triggering) пайплайна.

Обязательно ли самому писать веб-хуки, как Тофер?

Опять же нет! Мы используем Python, чтобы показать, как работают системы CD в целом, но их не обязательно создавать самому, можно выбрать одну из существующих систем CD, перечисленных в приложениях в конце книги. Поддержка веб-хуков — ключевая функция, на которую следует обращать внимание при выборе системы CD!

Не отправляйте изменения при сбое пайплайна

У Тофера возникает еще несколько проблем. Рассмотрим пару из них сейчас, а остальные отложим до главы 7. Что, если Анжела внесла изменение и не успела исправить его до того, как было внесено другое изменение?

Понедельник

Веб-хук Тофера отправляет электронное письмо Анжеле, сообщая ей, что после ее изменения произошел сбой пайплайна.

9:15: Анжела начинает устранять проблему.

9:20: Сато отправляет свое изменение.

Система контроля версий запускает веб-хук Тофера.

Веб-хук Тофера запускает пайплайн: он снова не проходит, поскольку Анжела все еще исправляет проблему, которую вызвало ее изменение.

Веб-хук Тофера отправляет электронное письмо Сато, сообщая ему, что его изменение привело к сбою пайплайна, хотя на самом деле виновата Анжела!

Пока Анжела исправляет свою проблему, Сато отправляет изменение. Система думает, что именно Сато стал причиной сбоя в работе пайплайна, но в действительности виновата Анжела, и поэтому бедняга Сато в замешательстве.

Кроме того, каждое изменение, которое добавляется поверх существующей проблемы, потенциально может усложнить ее решение. Бороться с этим можно, соблюдая простое правило:

Не отправлять изменения, когда произошел сбой пайплайна.

Это правило можно реализовать в самой системе CD, а также путем отправки оповещений о сбое пайплайна остальным разработчикам. Оставайтесь с нами до главы 7, и вы узнаете больше!

> ***Зачем вообще ждать сбоя пайплайна?***
>
> Не лучше ли, если бы Анжела узнала о существовании проблемы до того, как отправить изменения? Так она могла бы устранить ее до отправки, и это не помешало бы работе Сато. Да, это определенно лучше! Мы вернемся к этой теме в следующей главе, а более подробно разберем ее в главе 7.

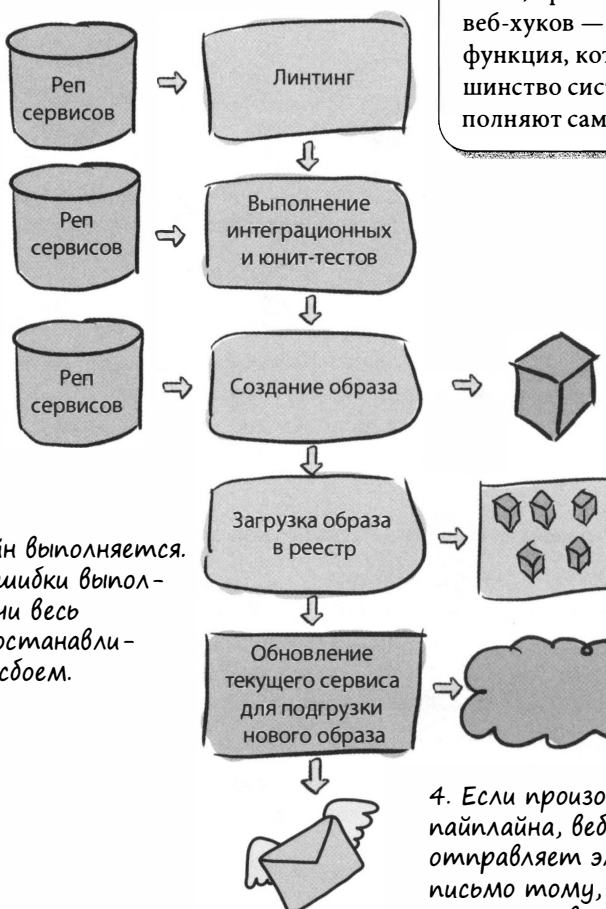
Непрерывная доставка для сайта Cat Picture

Ура! Теперь мы знаем все о том, как организовать непрерывную доставку сайта Cat Picture: о пайплайне, который разработчики используют для своих сервисов, а также о том, как его автоматизировать и запустить.

1. При внесении изменения создается событие, инициирующее активацию веб-хука Тофера в системе контроля версий.

2. Веб-хук Тофера, запущенный на его рабочей станции, запускает пайплайн.

Обязательно ли запускать веб-хуки прямо на своей рабочей станции? Нет. В главе 9 мы узнаем, почему не стоит делать этого; кроме того, запуск веб-хуков — это еще одна функция, которую большинство систем CD выполняют самостоятельно.



3. Пайплайн выполняется. В случае ошибки выполнения задачи весь пайплайн останавливается со сбоем.

4. Если произошел сбой пайплайна, веб-хук Тофера отправляет электронное письмо тому, кто внес изменение, вызвавшее сбой.

Как насчет терминологии?

Начав работать с инструментами CD, вы можете встретить термины, отличные от тех, которые мы использовали в этой главе и будем использовать в дальнейшем. Ниже представлены термины, относящиеся к непрерывной доставке, и их варианты, которыми мы будем пользоваться:

События (events) и *триггеры* (triggers) иногда используются как взаимозаменяемые понятия; иногда слово «триггер» относится к комбинации события и действия, которое необходимо предпринять.

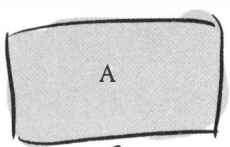
Веб-хуки — это обычно просто *веб-хуки*.

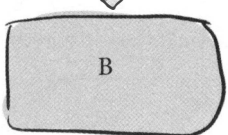
Машины, на которых выполняются пайплайны, можно называть так:

- *узлы* (nodes)
- *раннеры*, или *средства запуска* (runners)
- *исполнители* (executors)
- *агенты* (agents)

Задачи могут называться:

- *задачами* (tasks)
- *этапами* (stages)
- *заданиями* (jobs)
- *сборками*, или *билдами* (builds)
- *шагами** (steps)

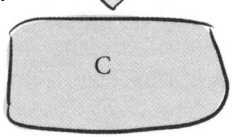




Пайплайны могут называться:

- *пайплайнами*
- *рабочими процессами*

* Иногда задачи разбиваются на еще более мелкие части, которые также называют *шагами*.



Система CD, которая объединяет все перечисленное выше, может называться:

- *CD-платформой*
- *платформой CI/CD*
- *сервером автоматизации*
- *сервером сборки*

Уведомления иногда также называют *событиями*.

Поскольку большая часть этого процесса автоматизации изначально разрабатывалась для сборки кода в исполняемые файлы, термин «*сборка*» (*билд*) прижился и может использоваться для обозначения задач, пайплайнов или даже комбинаций «триггер + задача + пайплайн».

Заключение

Пайплайн, используемый сервисами сайта Cat Picture, содержит все структурные элементы, типичные для большинства пайплайнов CD. Рассмотрев, как команда сайта Cat Picture работает с пайплайном, мы поняли, насколько автоматизация важна для масштабирования CD, особенно по мере роста компании. В следующей части книги мы подробно рассмотрим каждый элемент пайплайна и то, как использовать их совместно.

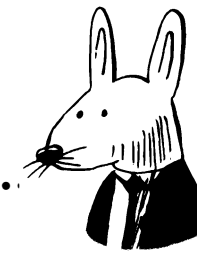
Итоги

- В этой книге термины «*пайплайны*» и «*задачи*» используются для обозначения основных структурных элементов CD, которые могут называться по-разному.
- Задачи похожи на функции. Задачи также могут называться *этапами*, *заданиями*, *сборками* (*билдами*) и *шагами*.
- Пайплайны — это механизмы управления, объединяющие задачи. Пайплайны также могут называться *рабочими процессами*.
- Основными составляющими пайплайна CD являются линтинг (статический анализ), тестирование, сборка, доставка и развертывание.
- Линтинг и тестирование — это шлюзы (они же задачи непрерывной интеграции), а сборка, доставка и развертывание — это преобразования.
- Системы контроля версий предоставляют такие инструменты, как события и веб-хуки, позволяющие автоматизировать выполнение пайплайна.
- Когда в пайплайне происходит сбой, следует прекратить внесение изменений!

Далее...

В следующей главе мы рассмотрим, почему в непрерывной доставке важен контроль версий и почему все текстовые данные, из которых состоит программа, должны храниться в системе контроля версий. Эту практику часто называют «*конфигурация как код*» (config-as-code).

Часть 2
Поддержание ПО в состоянии готовности к доставке



Изучив основные понятия непрерывной доставки, узнаем, как поддерживать ПО в постоянной готовности к доставке с помощью непрерывной интеграции.

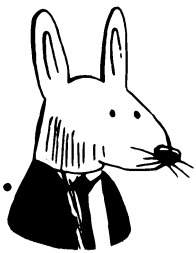
Глава 3 рассказывает о том, насколько важную роль играет система контроля версий при непрерывной доставке; без управления версиями непрерывная доставка невозможна.

В главе 4 рассматривается мощный, но редко обсуждаемый элемент непрерывной интеграции — статический анализ, в частности линтинг, и то, как его применять к унаследованному коду.

Главы 5 и 6 посвящены тестированию: ключевому элементу проверки при непрерывной интеграции. Мы не будем учить вас проводить тестирование (этой теме посвящено огромное количество литературы), а сосредоточимся на распространенных проблемах, которые со временем накапливаются в наборах тестов, в частности на медленной и нестабильной их работе.

В главе 7 рассматривается жизненный цикл изменения кода и исследуются все точки, где могут возникнуть баги. Вы узнаете, как настроить систему автоматизации на поиск и устранение багов по мере их появления.

3 Контроль версий — единственно верный путь



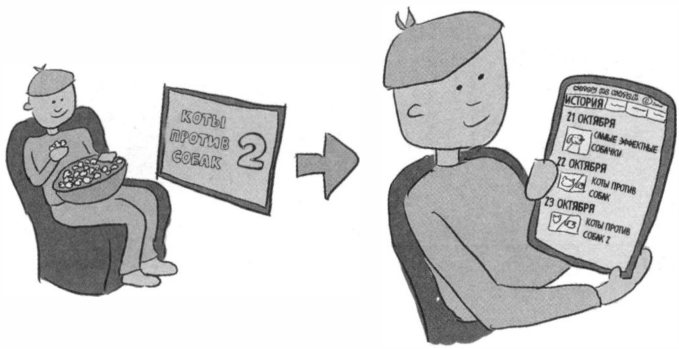
В этой главе

- ✓ Почему для CD необходим контроль версий
- ✓ Как поддерживать ПО в готовом к релизу состоянии за счет сохранения системы контроля версий в зеленом режиме и запуска пайплайнов на основе изменений в этой системе
- ✓ Что такое «*конфигурация как код*» (*config-as-code*)
- ✓ Автоматизация путем хранения всей конфигурации в системе контроля версий

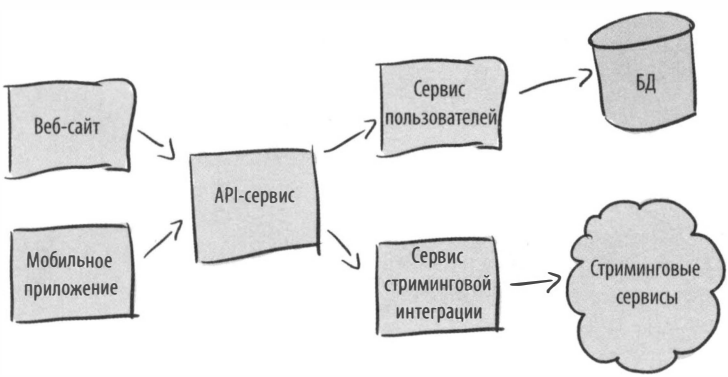
Мы начнем наше путешествие по миру непрерывной доставки с самого начала, то есть с инструмента, который лежит в основе абсолютно всего, что мы будем делать дальше, а именно с системы контроля версий. В этой главе вы узнаете, почему управление версиями имеет решающее значение для CD и как его использовать, чтобы обеспечить успех себе и своей команде.

Стартап Саши и Сары

Недавние выпускники университета Саша и Сара только что получили финансирование на реализацию амбициозного стартапа: сайта социальной сети Watch Me Watch, в основе которой лежат предпочтения пользователей, касающиеся просмотра телепередач и фильмов. В сети Watch Me Watch пользователи могут оценивать просмотренные фильмы и телешоу, видеть, что нравится их друзьям, и получать персональные рекомендации о том, что смотреть дальше.



Саша и Сара должны обеспечить бесперебойное взаимодействие с пользователями, поэтому они объединяются с популярными стриминг-провайдерами. Пользователям не придется долго и нудно добавлять в свой профиль просмотренные фильмы и шоу — весь этот контент будет автоматически загружаться в приложение! Прежде чем приступить к работе, Саша и Сара набросали общую схему желаемой архитектуры.



Они собираются разбить логику бэкенда на три сервиса:

- API-сервис Watch Me Watch, который обрабатывает все запросы от фронтендов;
- сервис пользователей, который хранит данные о пользователях;
- сервис стриминговой интеграции, осуществляющий интеграцию с популярными стриминг-провайдерами.

Саша и Сара также планируют создать два фронтенда для взаимодействия с Watch Me Watch — веб-сайт и мобильное приложение.

Все виды данных

С гордостью глядя на эту схему, нарисованную на новенькой маркерной доске, они понимают, что весь код, который они создают, нужно где-то размещать. Причем они оба будут вносить в него изменения, поэтому им потребуется определенная координация. Они собираются создать три сервиса с примерно одинаковым дизайном и структурой: они написаны на языке Go и работают как запускаемые контейнеры.

Они также запустят сайт и создадут и распространят мобильное приложение — оба этих продукта станут для пользователей средствами доступа к сервисам Watch Me Watch.

Данные для определения трех сервисов, приложения и веб-сайта будут включать в себя следующее:

- исходный код и тесты, написанные на Go;
- README и другие документы в формате Markdown;
- определение образов контейнеров (файлы Dockerfile) для сервисов;
- изображения для веб-сайта и мобильного приложения;
- определение задач и пайплайнов для тестирования, сборки и развертывания.

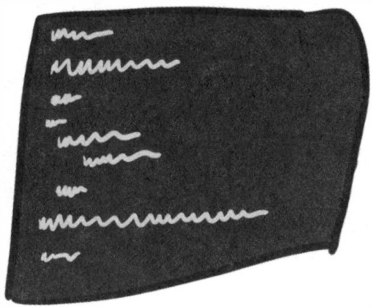
Для базы данных (которая будет работать в облаке) потребуется следующее:

- схемы версионирования;
- определение задач и пайплайнов для развертывания.

Для подключения к потоковым сервисам, с которыми будут работать Саша и Сара, им также понадобятся API-ключи и информация о подключении.

Исходный код и программы

Еще до того, как написать первую строчку кода, Саша и Сара, глядя на свою схему архитектуры и оценивая, что потребуется для каждого ее элемента, поймут, что им придется хранить большой объем данных:



- исходный код;
- тесты;
- файлы Dockerfiles;
- Markdown-файлы;
- задачи и пайплайны;
- схемы версий;
- ключи API;
- информацию о подключении.

Это очень много! (И это еще для довольно простой системы!) Но что общего у всех этих элементов? Все это данные. И если пойти чуть дальше, все они представляют собой *простой текст*.

Несмотря на то что каждый из этих элементов используется по-разному, все они представляют собой *простые текстовые данные*. И когда вы работаете над созданием и поддержкой программ, как это собираются делать Саша и Сара, вам нужно каким-то образом всеми ими управлять.

На помощь приходит *система контроля версий*. Эта система (также называемая *контролем исходного кода*) хранит данные и отслеживает их изменения. В ней хранятся все данные, необходимые продукту: исходный код, конфигурация, используемая для его запуска, вспомогательные данные, такие как документация и скрипты, — все данные, необходимые для определения и запуска программы, а также для взаимодействия с ней.

Словарик

Простые текстовые данные (plain text) — это данные в виде печатных (или человекочитаемых) символов. В контексте программного обеспечения простые текстовые данные часто противопоставляются *двоичным данным*, которые хранятся в виде последовательностей битов, не являющихся простым текстом. Проще говоря: простые текстовые данные — это удобочитаемые данные, а все остальное — двоичные данные. Систему контроля версий можно использовать для любых типов данных, но обычно она оптимизирована под простой текст, поэтому не очень хорошо справляется с двоичными данными. Это означает, что ее в принципе *можно* при желании использовать для хранения двоичных данных, но некоторые функции (например, отображение различий между изменениями) не будут работать или будут работать некорректно.

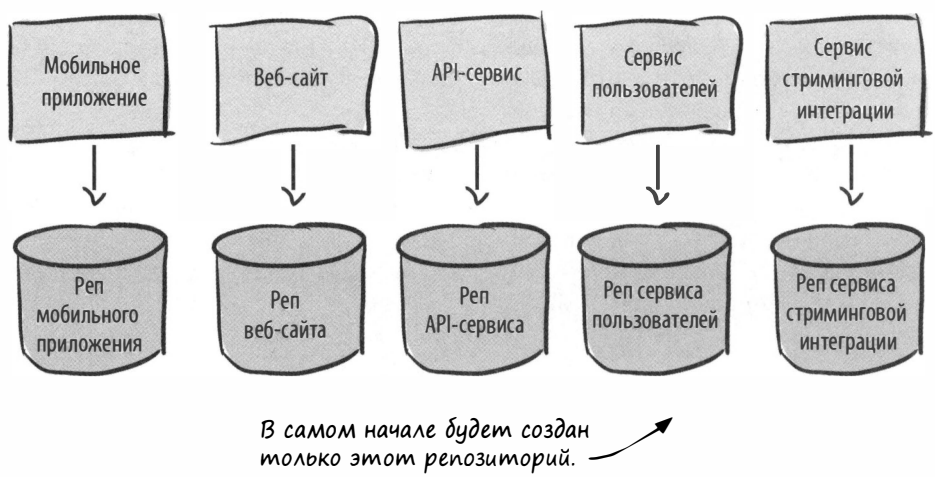
Репозитории и версии

Система контроля версий — это ПО для отслеживания изменений в простом тексте, где каждое изменение идентифицируется с помощью номера версии и называется *коммитом*, или *ревизией*. Система контроля версий продукта выполняет (по крайней мере) следующие две функции:

- Является централизованным местом для хранения всех данных, обычно называемым *репозиторием* (сокращенно — реп, или репа (repo)).
- Содержит историю всех изменений, причем каждое изменение (или набор изменений) приводит к созданию новой, однозначно идентифицируемой *версии*.

Конфигурация и исходный код, необходимые для проектов, часто хранятся в нескольких репозиториях. Один репозиторий для хранения всех данных используется достаточно редко и в этом случае называется *монорепозиторием* (или *монорепой*).

Саша и Сара решили использовать по одному репозиторию для каждого сервиса в своей архитектуре и определили, что прежде всего создадут репозиторий для сервиса пользователей.



Непрерывная доставка и контроль версий

Система контроля версий — это основа непрерывной доставки. Я сторонник того, чтобы относиться к непрерывной доставке как к *практике*, по умолчанию применяемой в программной разработке (в той или иной мере). Однако из этого правила есть одно исключение: если вы не используете систему контроля версий, вы не применяете непрерывную доставку.

Непрерывная доставка требует обязательного использования контроля версий.

Почему контроль версий так важен для непрерывной доставки? Вспомните условия непрерывной доставки:

- безопасное внесение изменений в ПО в любое время;
- максимально простая доставка ПО, буквально нажатием кнопки.

В главе 1 мы рассматривали, что требуется для достижения первого условия, — в частности, непрерывная интеграция, которую мы определили следующим образом:

> **Непрерывная интеграция — это процесс, при котором** *объединение изменений кода происходит постоянно* **и каждое изменение проверяется при внесении.**

Мы не пояснили, что в данном случае означает термин «*внесение изменений*» (*check-in*). На самом деле мы уже неявно предположили, что система контроля версий используется и «внесение» осуществляется в нее! Попробуем дать другое определение непрерывной интеграции, не предполагающее наличия контроля версий:

> ***Постоянный процесс внесения изменений в код,* когда каждое изменение проверяется *при добавлении его к уже имеющимся и проверенным изменениям.***

Из этого определения следует, что для непрерывной интеграции необходимы:

- способ внесения изменений;
- место для хранения (и добавления) изменений.

А как мы храним и вносим все изменения в код? Совершенно верно, с помощью системы контроля версий. Далее при обсуждении элементов, которые вам понадобятся при создании пайплайна непрерывной доставки, мы будем исходить из того, что все начинается с возможности отслеживать изменения в системе контроля версий.

ВАЖНО

Чтобы осуществлять непрерывную доставку, обязательно использовать систему контроля версий.

ВАЖНО

Разработка и поддержка программного обеспечения означает создание и редактирование большого объема данных, в частности простых текстовых данных. Используйте систему контроля версий для хранения и отслеживания истории исходного кода и конфигурации, то есть всех данных, которые необходимы для определения создаваемого продукта. Храните данные в одном или нескольких репозиториях, где каждое изменение однозначно идентифицируется с помощью версий.

Git и GitHub

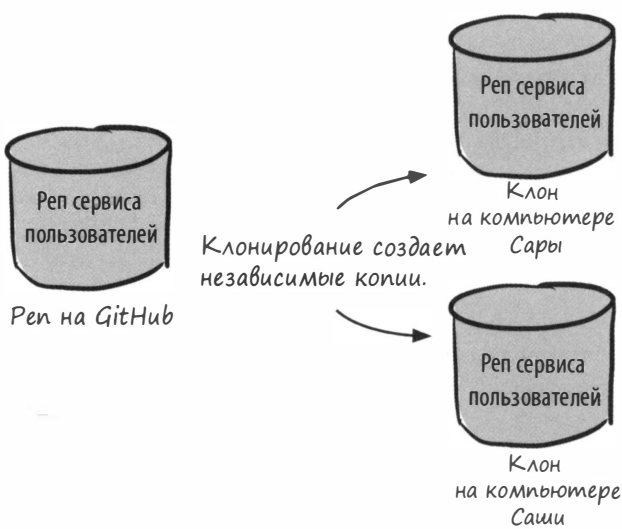
Сара и Саша собираются использовать Git для контроля версий. Следующий вопрос — где будет размещен их репозиторий и как они будут с ним взаимодействовать. Сара и Саша собираются использовать GitHub для размещения этого и последующих создаваемых ими репозиториев.

Git — это *распределенная система контроля версий*. Когда вы *клонируете* (копируете) репозиторий на свой компьютер, вы получаете полную копию всего репозитория, которую можно использовать независимо от удаленной версии. У нее даже история изменений отдельная.

Сара создает первый репозиторий проекта на GitHub, а затем клонирует его; таким образом на ее компьютере создается еще одна копия репозитория с теми же коммитами (сейчас их еще нет), но она может вносить в нее изменения независимо. Саша делает то же самое, теперь у них обоих есть клоны репозитория, над которыми они могут работать независимо и использовать их для *отправки* (пуша, push) изменений обратно в общий репозиторий на GitHub.

Какую систему контроля версий использовать? На момент написания книги широкой поддержкой и популярностью пользовался Git — отличный выбор!

Хотя для наших примеров мы используем GitHub, существуют и другие интересные варианты, со своими преимуществами и недостатками. Обзор таких систем представлен в приложении B.



Управление конфигурацией ПО и исходным кодом

Интересный факт! ПО для контроля версий является частью *управления конфигурацией программного обеспечения* (software configuration management, SCM) — процесса отслеживания изменений в конфигурации, используемой для сборки и запуска создаваемого продукта. В этом контексте под словом *конфигурация* подразумевается подробная информация обо всех данных в репозитории, включая исходный код, но существует еще и *управление конфигурацией* компьютера, которое восходит как минимум к 1970-м годам. Эта терминология вышла из употребления, и ей на смену пришли такие понятия, как «*инфраструктура как код*» и более позднее «*конфигурация как код*» (подробнее о них чуть далее), а аббревиатура *SCM* стала обозначать *управление исходным кодом* (source code management). В настоящее время SCM часто используется для обозначения системы контроля версий, а иногда и таких систем, как GitHub, которые предлагают управление версиями в сочетании с функциями отслеживания проблем и проверки кода.

Первый коммит — с багом!

Сара и Саша создали клоны репозитория сервиса пользователей и готовы к работе. В порыве вдохновения Сара начинает работать над исходным классом `User` в репозитории. Она рассчитывает, что репозиторий сможет хранить все фильмы, которые смотрел пользователь, и рейтинги, которые пользователь явно присваивал фильмам.

Созданный ею класс `User` хранит имя пользователя, и она добавляет метод `rate_movie`, который будет вызываться, когда пользователь захочет оценить фильм. Метод берет название фильма, который нужно оценить, и рейтинг (в процентах с плавающей точкой), который нужно поставить фильму, и пытается сохранить их в объекте `User`, но в коде ошибка: он использует объект `self.ratings`, но этот объект нигде не инициализирован.

```
class User:
  def __init__(self, name):
    self.name = name

  def rate_movie(self, movie, score):
    self.ratings[movie] = score
```

Здесь баг: словарь `self.ratings` *не инициализирован, поэтому попытка сохранить в нем ключ вызовет исключение!*

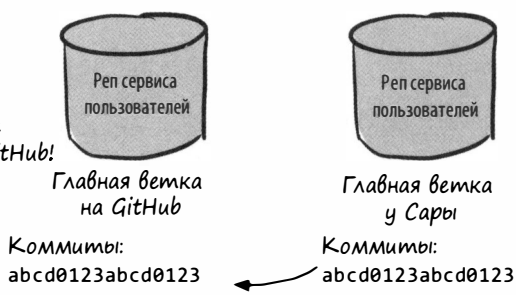
Сара внесла баг в этот код, однако она также написала юнит-тест, который отловит эту ошибку. Тест (`test_rate_movie`) пытается оценить фильм, а затем проверяет, был ли добавлен рейтинг:

```
def test_rate_movie(self):
  u = User("sarah")
  u.rate_movie("jurassic park", 0.9)
  self.assertEqual(u.ratings["jurassic park"], 0.9)
```

К сожалению, Сара забывает запустить тест перед тем, как коммитить новый код! Она добавляет эти изменения в свой локальный репозиторий, создавая новый коммит с идентификатором `abcd0123abcd0123`. Она коммитит его в главную ветку своего репозитория, а затем отправляет коммит обратно в главную ветку репозитория GitHub.

По умолчанию первая ветка, создаваемая в Git, называется *главной* (*main*). Данная ветка по умолчанию используется как *источник истины* (source of truth) — заслуживающая доверия версия кода, и все изменения в конечном итоге интегрируются в нее. Обсуждение других стратегий ветвления см. в главе 8.

Сара перемещает свою главную ветку обратно в репозиторий GitHub. Теперь баг находится в главной ветке репозитория GitHub!



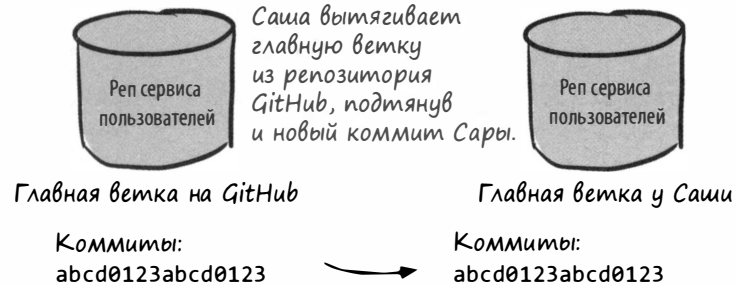
Эти идентификаторы приведены просто для примера; реальные идентификаторы коммитов Git — это хеш SHA-1 коммита.

Словарик

Внесение изменений в одну ветку из другой ветки часто называют *слиянием*. Изменения Сары из ее главной ветки *сливаются* с главной веткой GitHub.

Нарушение работы главной ветки

Вскоре после того, как Сара выложила свой новый код (и свой баг!), Саша *вытягивает* (pull) главную ветку с GitHub в свой локальный репозиторий, подтягивая туда новый коммит.



Саша в восторге от изменений Сары:

```
class User:
  def __init__(self, name):
    self.name = name

  def rate_movie(self, movie, score):
    self.ratings[movie] = score
```

Саша сразу же пробует применить их, но пытаясь использовать `rate_movie`, он наталкивается на баг и видит следующее сообщение об ошибке:

```
AttributeError: 'User' object has no attribute 'ratings'
```

«Мне казалось, я видел, что Сара добавила юнит-тест для этого метода, — недоумевает Саша. — Как он может не работать?»

```
  def test_rate_movie(self):
    u = User("sarah")
    u.rate_movie("jurassic park", 0.9)
    self.assertEqual(u.ratings["jurassic park"], 0.9)
```

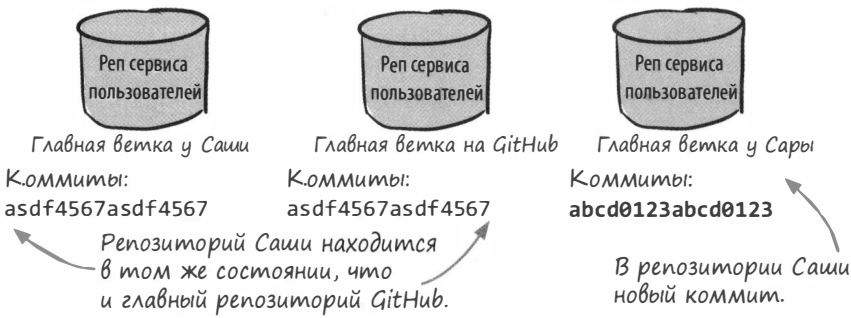
Саша запускает юнит-тест, и, увы, он тоже не проходит:

```
Traceback (most recent call last):
  File "test_user.py", line 21, in test_rate_movie
    u.rate_movie("jurassic park", 0.9)
  File "test_user.py", line 12, in rate_movie
    self.ratings[movie] = score
AttributeError: 'User' object has no attribute 'ratings'
```

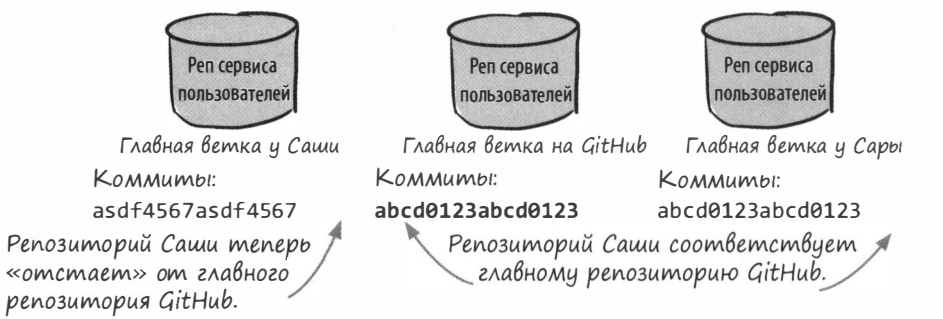
Саша понимает, что код в репозитории GitHub не работает.

Отправка (push) и вытягивание (pull) изменений

Посмотрим еще раз на описанные выше отправку и вытягивание изменений. После того как Сара внесла свои изменения и закоммитила их локально, репозитории стали выглядеть так:



Чтобы обновить репозиторий в GitHub с помощью своего коммита, Сара отправила коммит в удаленный репозиторий (то есть *осуществила его слияние*). Git просмотрит содержимое и историю удаленного репозитория, сравнит его с локальной копией Сары и *отправит* (загрузит) все недостающие коммиты в репозиторий. После этого оба репозитория будут иметь одинаковое содержимое:



Чтобы Саша мог получить изменения Сары, он должен *подтянуть* главную ветку с GitHub. Git просмотрит содержимое и историю репозитория GitHub, сравнит его с локальной копией Саши и подтянет все недостающие коммиты в репозиторий Саши:

Теперь во всех трех репозиториях одинаковые коммиты. Сара отправила новый коммит на GitHub, а Саша подтянул коммит оттуда.

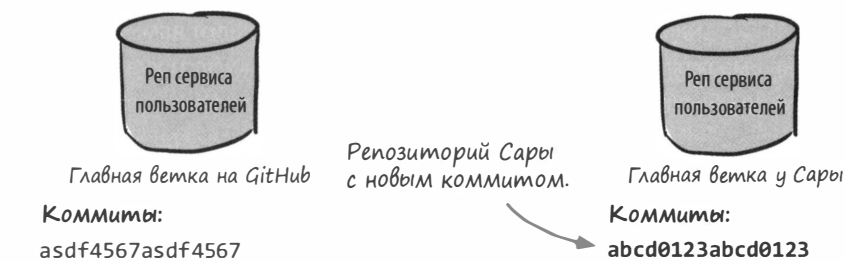
Запрос на вытягивание (pull request, пул-реквест)

Возможно, вы заметили, что Сара внесла свои изменения непосредственно в главную ветку репозитория на GitHub. Более безопасная стратегия — предусмотреть промежуточный этап, на котором изменения «предлагаются» перед добавлением. Это дает возможность проверить изменения с помощью кодревью и методов CI, прежде чем отправлять их.

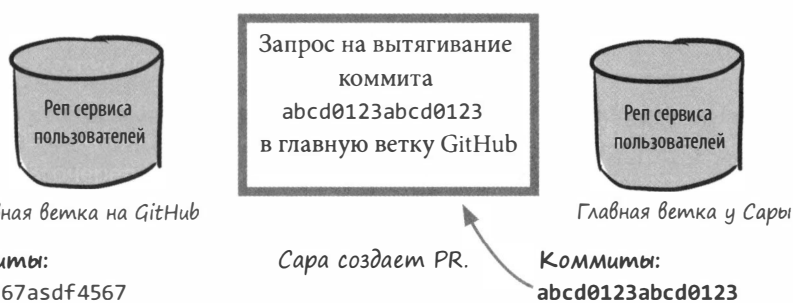
В этой главе мы рассмотрим инициирование непрерывной интеграции (CI) при отправке изменений в систему контроля версий, но в дальнейшем будем говорить о запуске CI *до* добавления изменений в главную ветку.

Возможность предлагать, просматривать и проверять изменения осуществляется с помощью *запросов на вытягивание,* или *пул-реквестов* (часто обозначаемых аббревиатурой *PR*, *pull request*), и в дальнейшем вы часто будете встречать термин на страницах книги. (Подробнее об этом термине и о том, как используется PR в различных удаленных системах контроля версий, см. в приложении B.)

Если бы Сара и Саша использовали PR, предыдущий процесс выглядел бы так:



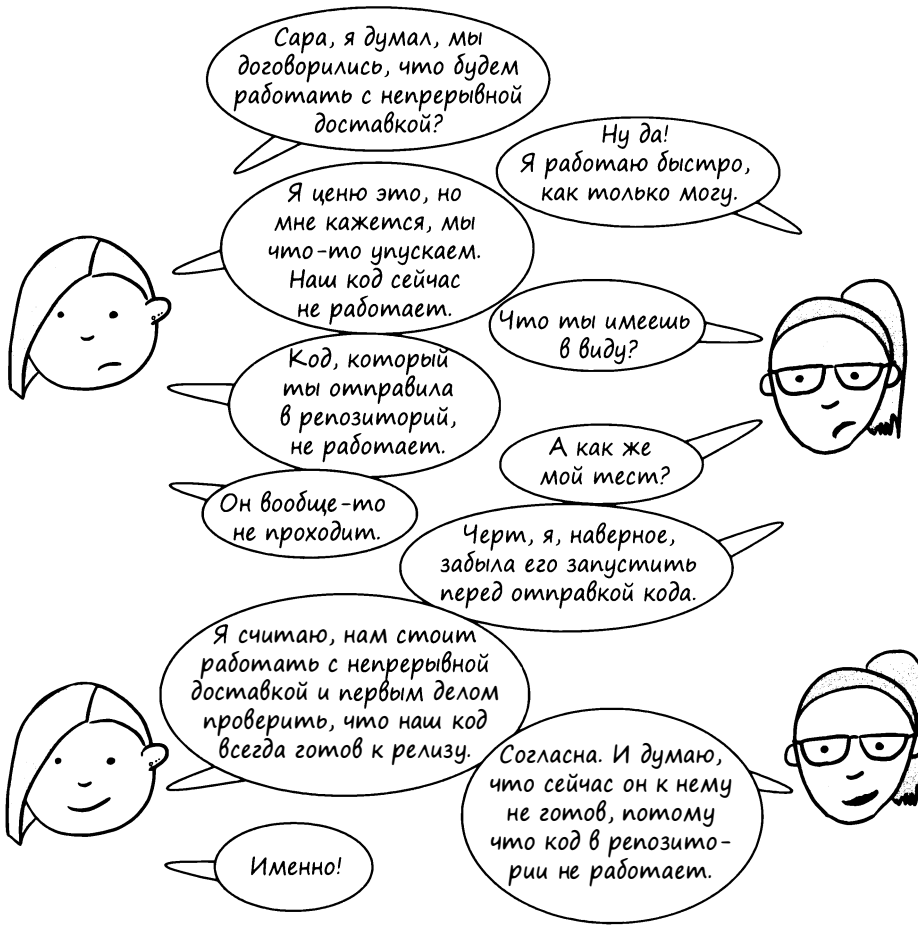
Вместо того чтобы отправлять свои изменения непосредственно в главную ветку GitHub, Сара открывает пул-реквест; она создает заявку с запросом на вытягивание ее изменений в главную ветку GitHub:



Это дает Саше возможность ознакомиться с изменениями и проверить их до того, как обновлять главную ветку GitHub.

Это действительно непрерывная доставка?

Саша немного расстроен, узнав, что код сервиса пользователей в репозитории GitHub не работает, и обращается с этой проблемой к Саре.



Поддерживайте возможность выпуска релизов

Сара и Саша поняли, что, позволяя коммитить нерабочий код в репозиторий сервиса пользователей на GitHub, они нарушают один из двух принципов CD. Вспомните условия непрерывной доставки:

- безопасное внесение изменений в ПО в любое время;
- максимально простая доставка ПО, буквально нажатием кнопки.

Доставку сервиса пользователей невозможно осуществить безопасно, пока не будет исправлена ошибка, сделанная Сарой. Это означает, что сервис не находится в состоянии, в котором доставка безопасна.

Сара может исправить ошибку и быстро отправить коммит с исправлением, но как быть уверенным, что это не повторится? В конце концов, Сара написала тест, который выявил проблему, но этого оказалось недостаточно, чтобы избежать проникновения бага в репозиторий.

Как бы Сара ни старалась, в будущем она опять может забыть запустить тесты перед коммитом, и Саша тоже. Ведь они, в конечном счете, всего лишь люди!

> ***Разве неработающий код не будет оставаться всегда?***
> Вы никогда не сможете отловить все баги до единого, поэтому в каком-то смысле неработающий код всегда будет попадать в систему контроля версий. Главное — всегда поддерживать ее в таком состоянии, чтобы можно было уверенно выпускать релизы; случайные баги — обычное дело, задача в том, чтобы до минимума снизить необходимость отката и риски, связанные с релизом или развертыванием. Подробнее о выпуске релизов читайте в главах 8 и 10.

Саше и Саре необходимо обеспечить выполнение тестов. Если нужно что-то гарантировать, лучше всего это автоматизировать (по возможности).

Если вы поручаете людям задачи, которые должны быть выполнены без ошибок, то готовьтесь к тому, что ошибки все же иногда будут возникать — и это совершенно нормально, потому что люди так устроены! Позвольте им быть специалистами в своем деле, а когда нужно гарантировать, что одно и то же действие будет выполняться каждый раз одинаково и без сбоев, применяйте автоматизацию.

ВАЖНО

Если вам необходимо гарантировать наступление какого-то события, используйте автоматизацию. Люди не машины, и им свойственно ошибаться и забывать что-то сделать. Не обвиняйте человека в забывчивости, лучше найдите способ избавить его от необходимости помнить о задаче.

Срабатывание при изменениях в системе контроля версий

Анализируя, почему репозиторий сервиса пользователей оказался в небезопасном состоянии, мы поймем, что Сара ошиблась не тогда, когда внесла баг в код, и даже не тогда, когда закоммитила его. Проблемы начались, когда она отправила неработающий код в удаленный репозиторий:

По-прежнему работаем с CD; ошибки случаются!

1. Сара пишет плохой код.

Пока все в порядке. И вот она забыла запустить тесты; такое бывает.

2. Сара коммитит код в своем репозитории.

Теперь невозможно выпустить безопасный релиз сервиса пользователей, и мы все дальше отходим от принципов CD.

3. Сара отправляет коммит в репозиторий GitHub.

Выпускать релиз не только небезопасно, но еще никто и не знает, что это небезопасно. Казалось бы, разумно выпустить релиз сейчас — но тут что-то действительно идет не так.

4. Саша пытается использовать новый код и находит баг.

Так чего же не хватает между шагами 3 и 4, что позволило бы Саре и Саше соблюсти принципы непрерывной доставки?

В главе 2 вы узнали важное правило, что делать при внесении критических изменений:

> **Не отправлять изменения, когда произошел сбой пайплайна.**

Какой еще пайплайн? У Саши и Сары нет никакого пайплайна и никакой автоматизации процессов. Чтобы найти ошибки, им приходится проводить ручное тестирование. И это как раз то, чего не хватает Саше и Саре, — не просто пайплайна для автоматизации ручных операций и обеспечения их надежности, но и *его настройки на срабатывание (triggering) при внесении изменений в удаленный репозиторий*:

> **Пайплайн должен срабатывать при изменениях в системе контроля версий.**

Если бы у Саши и Сары был пайплайн, запускающий юнит-тесты при каждой отправке изменений в репозиторий GitHub, Сара немедленно получила бы уведомление о проблеме, которая возникла по ее вине.

> ***Можно ли что-то сделать немного раньше, чтобы шаг 3 вообще не наступил?***
>
> Несомненно! Понять, в какой момент появляются проблемы, уже хорошее начало, но еще лучше — предотвратить их появление. Это можно сделать, запустив пайплайны до отправки коммитов в главную ветку удаленного репозитория. Подробнее об этом читайте в главе 7.

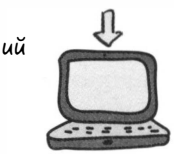
Запуск пайплайна сервиса пользователей

Саша и Сара создают пайплайн. Пока что в нем только одна задача — запускать юнит-тесты. Они настроили срабатывание веб-хука так, чтобы пайплайн автоматически запускался каждый раз, когда коммиты отправляются в репозиторий на GitHub, а в случае, если пайплайн не пройдет, им обоим на почту будет отправлено уведомление.

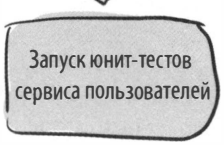
Теперь, если будут внесены критические изменения, Саша и Сара сразу узнают об этом. Они соглашаются принять политику «бросить все, чтобы заняться исправлением неполадок»:

Не отправлять изменения, когда произошел сбой пайплайна.

1. После внесения изменений GitHub запустит выполнение пайплайна.



2. Пайплайн содержит только одну задачу, запускающую юнит-тесты для сервиса пользователей.



3. Если произойдет сбой пайплайна (в данном случае — если юнит-тесты не пройдут), Саре и Саше будет отправлен email и они узнают о проблеме.

4. Если произойдет сбой пайплайна, будет небезопасно выпускать релиз кода, находящегося в репозитории GitHub. Сара и Саша соглашаются, что в этом случае не следует производить слияние других изменений, а приоритетом должно стать исправление кода и возвращение репозитория в состояние, пригодное для выпуска релиза.

Как происходит запуск пайплайна?

Мы не будем подробно разбирать особенности системы CD, которую выбрали Саша и Сара; некоторые из рассмотренных ими вариантов можно найти в приложениях в конце книги. Поскольку они уже используют GitHub, то могут легко и быстро запустить свой пайплайн через GitHub Actions.

ВАЖНО

Запускайте пайплайны при внесении изменений в систему контроля версий. Просто написать тесты недостаточно, их нужно регулярно запускать. Полагаться на то, что люди будут помнить о необходимости запускать тесты вручную, — значит провоцировать ошибки. Система контроля версий — это не только источник достоверной информации о состоянии продукта; это еще и отправная точка для автоматизации процесса CD, которую мы подробно изучим в этой книге.

Сборка сервиса пользователей

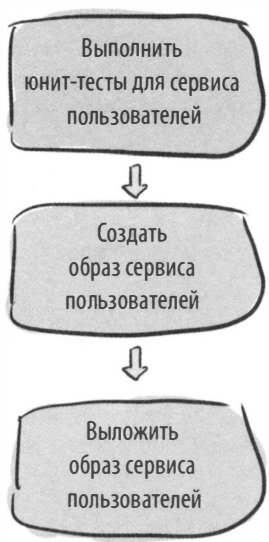
У Сары и Саши теперь есть пайплайн (небольшой), который немедленно отправит им уведомление, если что-то сломается. Но код сервиса пользователей не принесет им никакой пользы, если они ничего с ним не сделают! Пока что пайплайн помог им с первым условием CD:

Возможность безопасно вносить изменения в ПО в любое время.

Пайплайн и автоматизация его запуска помогут им и со вторым условием CD:

Доставка ПО должна быть такой же простой, как нажатие кнопки.

Им нужно добавить в пайплайн задачи сборки и публикации сервиса пользователей. Они решают упаковать сервис пользователей в образ контейнера и отправить его в реестр образов.



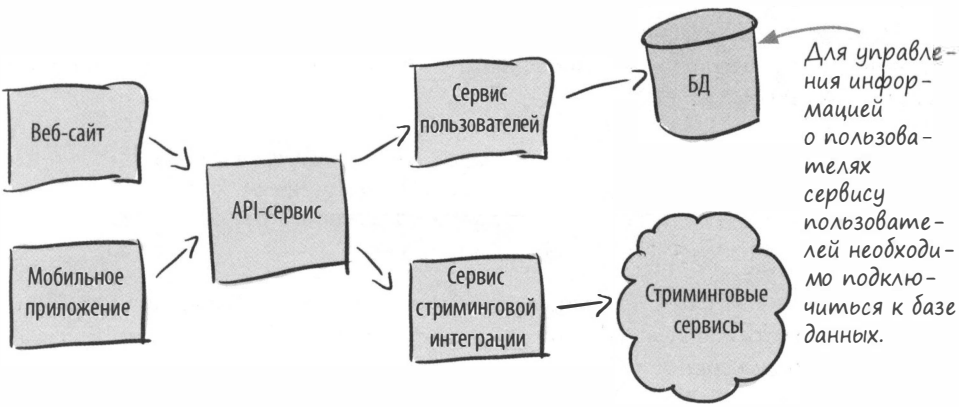
Добавив эти задачи в пайплайн, они делают процесс CD «простым, как нажатие кнопки» (или, в данном случае, еще проще, поскольку он будет запускаться автоматически с каждым внесением изменения в систему контроля версий).

Теперь при каждом коммите будут выполняться юнит-тесты, и если они пройдут успешно, то сервис пользователей будет упакован в контейнер и выложен в виде образа.

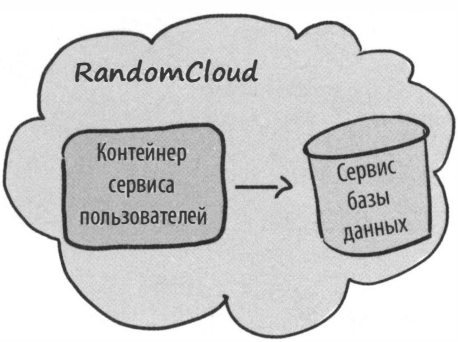
Сервис пользователей в облаке

Последний вопрос, на который Сара и Саша должны ответить для реализации сервиса пользователей, — как будет запускаться автоматически создаваемый образ. Они решают делать это с помощью популярного облачного провайдера RandomCloud.

RandomCloud предоставляет сервис для запуска контейнеров, поэтому задача легко решается, только надо учесть, что сервису пользователей понадобится доступ к базе данных, где хранится информация о пользователях и фильмах:



К счастью, как и большинство облачных решений, RandomCloud предоставляет сервис базы данных, который Сара и Саша могут использовать вместе с сервисом пользователей:



Поскольку пайплайн сервиса пользователей автоматически создает и публикует его образ, все, что Саше и Саре теперь остается сделать, — это настроить контейнер сервиса пользователей на использование базы данных на RandomCloud.

Подключение к базе данных RandomCloud

Чтобы запустить сервис пользователей на RandomCloud, Саше и Саре нужно настроить конфигурацию его контейнера так, чтобы он мог подключиться к сервису базы данных на RandomCloud. Чтобы это сделать, необходимы два условия:

- Возможность настраивать сервис пользователей, используя информацию, необходимую ему для подключения к базе данных.
- Возможность предоставить сервису пользователей при запуске конкретную конфигурацию, позволяющую работать с сервисом базы данных на RandomCloud.

Чтобы выполнить первое условие, Саша добавляет параметры командной строки, которые сервис пользователей использует для определения того, к какой базе данных подключаться:

```
./user_service.py \
  --db-host=10.10.10.10 \
  --db-username=some-user \
  --db-password=some-password \
  --db-name=watch-me-watch-users
```

Информация о подключении к базе данных предоставляется в виде аргументов командной строки.

Для выполнения второго условия сведения о сервисе базы данных RandomCloud можно передавать в виде конфигурации, которую RandomCloud использует для запуска контейнера сервиса пользователей:

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=some-user
  - --db-password=some-password
  - --db-name=watch-me-watch-users
```

Этот образ собирается и передается как часть пайплайна сервиса пользователей. Он содержит и запускает файл user_service.py.

Это те же аргументы, что и в предыдущем коде, которые теперь предоставляются как часть конфигурации RandomCloud.

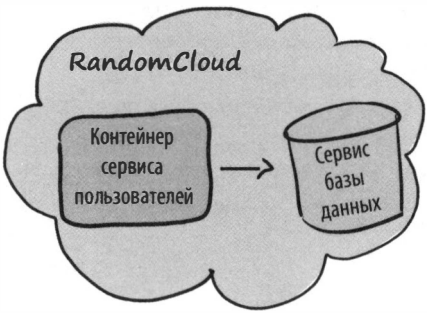
Развертывание самой последней версии образа имеет серьезные недостатки. О том, в чем они заключаются и что можно с этим сделать, читайте в главе 9.

Следует ли передавать пароли в виде простых текстовых данных?

Если кратко — нет. Саша и Сара вот-вот поймут, что им нужно хранить эти конфигурационные данные в системе контроля версий и уж точно не стоит коммитить туда пароль. Подробнее об этом чуть позже.

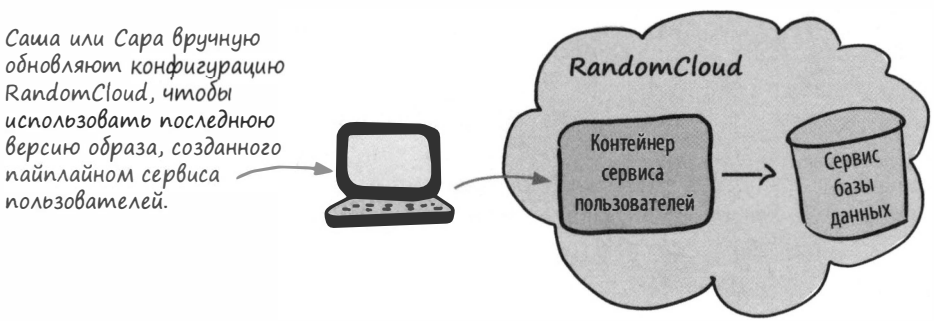
Управление сервисом пользователей

Сара и Саша готовы запустить сервис пользователей в виде контейнера с помощью популярного облачного провайдера RandomCloud.

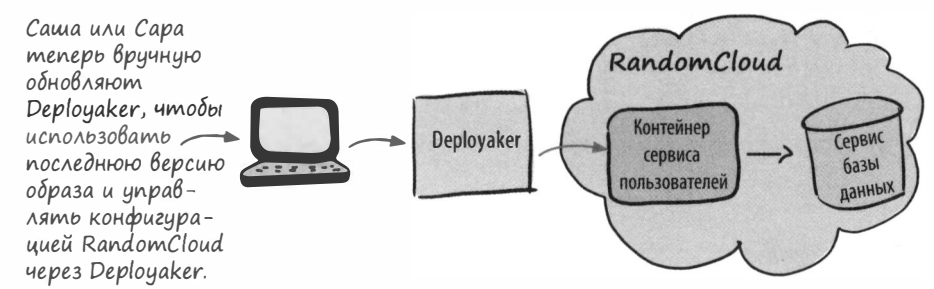


Подробнее об автоматизации развертывания читайте в главе 10!

Первые недели, каждый раз, когда они собираются выполнить запуск сервиса, они используют пользовательский интерфейс RandomCloud (RandomCloud UI) для обновления конфигурации контейнера до последней версии, иногда меняя и аргументы.



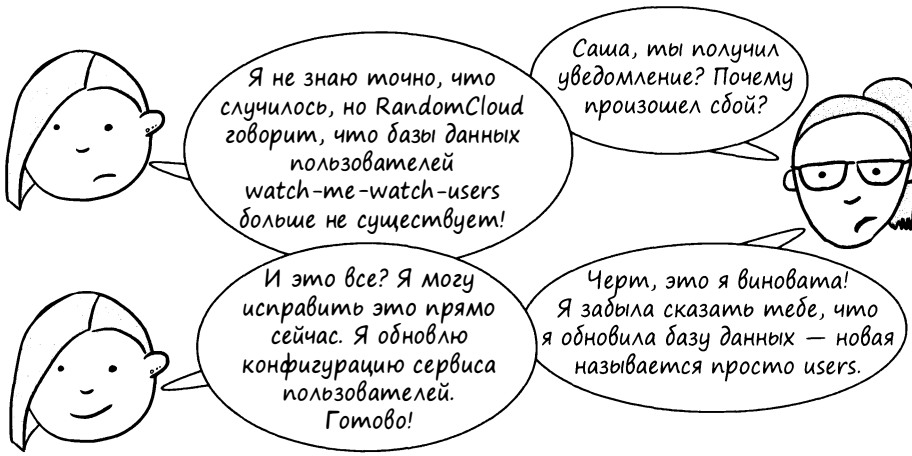
Вскоре Сара и Саша решают вложиться в инструментарий для развертывания и приобретают лицензию на Deployaker — инструмент, позволяющий легко управлять развертыванием сервиса пользователей (а позднее и других сервисов Watch Me Watch).



Сервис пользователей теперь запускается в контейнере на RandomCloud и управляется инструментом Deployaker. Deployaker постоянно мониторит состояние сервиса пользователей и следит, чтобы он всегда имел надлежащую конфигурацию.

Сбой в работе сервиса пользователей

Однажды в четверг Саше на телефон приходит уведомление от RandomCloud о том, что сервис пользователей не работает. Саша просматривает логи сервиса и обнаруживает, что он больше не может подключиться к сервису базы данных. Базы данных под названием `watch-me-watch-users` больше не существует!



Саша бросается исправлять настройки, но совершает роковую ошибку. Он совсем забывает, что теперь сервисом пользователей управляет Deployaker. И вместо того чтобы использовать Deployaker для обновления настроек, он вносит исправления напрямую через пользовательский интерфейс RandomCloud:

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=some-user
  - --db-password=some-password
  - --db-name=users
```

Саша обновляет конфигурацию, чтобы использовать правильную базу данных, но вносит изменения напрямую в RandomCloud, забыв о Deployaker.

Сервис исправлен, и уведомления от RandomCloud больше не приходят.

> ***Что такое Deployaker?***
>
> Deployaker — это вымышленный инструмент, аналог Spinnaker, реально существующего инструмента развертывания с открытым исходным кодом. Подробнее о развертывании см. главу 10.

Автоматизация всех переиграла

Саша поспешил внести исправления в конфигурацию RandomCloud, чтобы возобновить работу сервиса пользователей, но совершенно забыл о Deployaker. Крепкий ночной сон Сары был прерван внезапным уведомлением от RandomCloud. Сервис пользователей опять сломался!

Сара открывает интерфейс Deployaker и изучает конфигурацию, которую он использует для работы сервиса пользователей:


```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  – --db-host=10.10.10.10
  – --db-username=some-user
  – --db-password=some-password
  – --db-name=name=watch-me-watch-users
```

Эта конфигурация по-прежнему использует базу данных, которую Сара уже удалила!

Несмотря на то что Сара безумно устала, она понимает, что произошло. Саша исправил конфигурацию на RandomCloud, но не обновил ее в Deployaker. Deployaker периодически проверяет развернутый сервис пользователей, чтобы убедиться, что он развернут и сконфигурирован правильно. К сожалению, когда Deployaker проверял сервис той ночью, он обнаружил изменения, внесенные Сарой, которые не соответствовали тому, что, как он считал, верно. Deployaker решил эту проблему, перезаписав исправленную конфигурацию той конфигурацией, которую он хранил у себя, что снова вызвало сбой в работе! Сара вздыхает и вносит исправление в Deployaker:

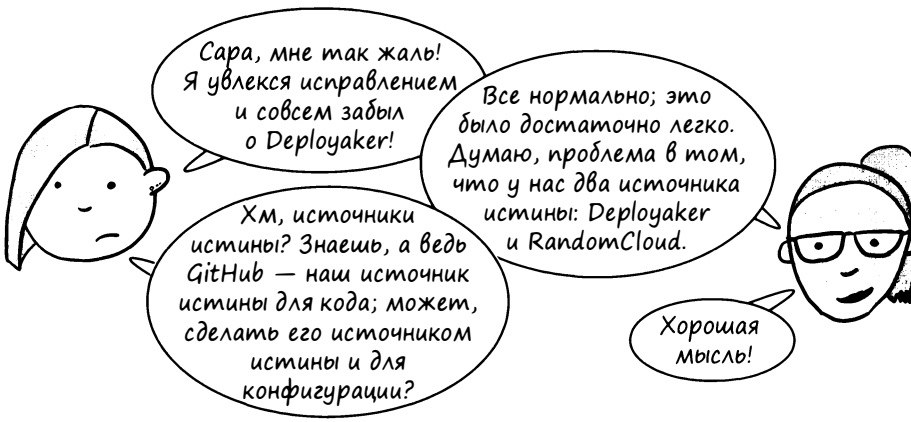
```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  – --db-host=10.10.10.10
  – --db-username=some-user
  – --db-password=some-password
  – --db-name=users
```

Теперь правильная конфигурация хранится в Deployaker, и Deployaker позаботится о том, чтобы сервис, запущенный в RandomCloud, использовал именно ее.

Уведомления прекратились, и Сара наконец-то смогла снова пойти спать.

В чем «источник истины»?

На следующее утро, за чашкой кофе, сонная Сара рассказала Саше, что произошло.



Конфигурация, о которой они говорят, — это настройки RandomCloud для контейнера сервиса пользователей, которые нужно было изменить, чтобы устранить сбой, произошедший накануне:

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  – --db-host=10.10.10.10
  – --db-username=some-user
  – --db-password=some-password
  – --db-name=users # или --db-name=watch-me-watch-users
```

У этой конфигурации два источника истины:

- конфигурация, которую использовал RandomCloud;
- конфигурация, хранящаяся в Deployaker, которую он будет брать для перезаписи любой используемой в RandomCloud, если обнаружит разницу.

Саша предположила, что стоит хранить эту конфигурацию в репозитории GitHub вместе с исходным кодом сервиса пользователей. Но не станет ли это в итоге третьим источником истины?

Последний недостающий элемент — это настройка Deployaker на использование конфигурации в репозитории GitHub в качестве источника истины.

Контроль версий и безопасность

Как правило, все данные, имеющие вид простого текста, должны находиться в системе контроля версий. Но что делать с конфиденциальными данными, такими как секретные коды и пароли? Эта информация не должна быть доступна каждому, у кого есть доступ к репозиторию (да это им обычно и не нужно). К тому же если добавить эту информацию в систему контроля версий, она будет храниться в истории репозитория неограниченное время!

Конфигурация сервиса пользователей Саши и Сары содержит конфиденциальные данные — имя пользователя и пароль для подключения к сервису базы данных:

user-service.yaml

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=some-user
  - --db-password=some-password
  - --db-name=users
```

Саше и Саре нужно, чтобы этот файл конфигурации был в системе контроля версий, но при этом они не хотят коммитить конфиденциальные данные.

Нужно закоммитить этот файл конфигурации в системе контроля версий; как это сделать, не передавая имя пользователя и пароль? Ответ — хранить эту информацию в другом месте, чтобы управление ею и заполнение осуществлялось без вашего участия. Большинство облачных сервисов предоставляют инструменты для хранения защищенной информации, и многие системы CD позволяют безопасно заполнять эти секретные данные, что означает достаточную степень доверия системе CD и предоставление ей доступа к ним.

Саша и Сара решают хранить имя пользователя и пароль в бакете хранилища RandomCloud и настраивают Deployaker так, чтобы он получал доступ к значениям в этом бакете и заполнял их во время развертывания:

user-service.yaml

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=randomCloud:watchMeWatch:userServiceDBUser
  - --db-password=randomCloud:watchMeWatch:userServiceDBPass
  - --db-name=users
```

Эти ключевые слова указывают программе Deployaker на необходимость получения реальных значений параметров из RandomCloud.

Принцип «конфигурация как код» для сервиса пользователей

Теперь, когда Саша и Сара настроили Deployaker, чтобы он мог получать конфиденциальные данные (имя пользователя и пароль для базы данных сервиса пользователей) из RandomCloud, им нужно закоммитить файл конфигурации для репозитория сервиса пользователей:

user-service.yaml

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=randomCloud:watchMeWatch:userServiceDBUser
  - --db-password=randomCloud:watchMeWatch:userServiceDBPass
  - --db-name=users
```

Они создают новый каталог в репозитории сервиса пользователей с именем config, где будут хранить этот файл конфигурации и помещать туда любые другие конфигурации, которые им понадобятся в процессе работы. Теперь структура репозитория сервиса пользователей выглядит так:

В этом новом каталоге будет храниться конфигурация сервиса пользователей, используемая Deployaker, а также любая другая конфигурация, которую понадобится добавить в будущем.

```
docs/
config/
  user-service.yaml
service/
test/
setup.py
LICENSE
README.md
requirements.txt
```

Весь исходный код находится в каталоге service.

Можно ли вместо этого хранить конфигурацию в отдельном репозитории?

Иногда это имеет смысл, особенно если вы работаете с несколькими сервисами и хотите управлять всеми их конфигурациями из одной точки. Однако хранение конфигурации рядом с кодом, который она настраивает, упрощает одновременное внесение изменений в них обоих. Начните с использования единого репозитория и переносите конфигурацию в отдельный репозиторий только в том случае, если позже вы поймете, что это действительно необходимо.

Жестко закодированные данные

user-service.yaml

```
apiVesion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=randomCloud:watchMeWatch:userServiceDBUser
  - --db-password=randomCloud:watchMeWatch:userServiceDBPass
  - --db-name=users
```

Даже при том, что Deployaker самостоятельно заполняет некоторые из этих значений, информация о подключении к базе данных, по сути, является жестко закодированной, и ее нельзя использовать в других средах.

Поскольку информация о подключении к базе данных жестко закодирована, ее нельзя использовать в других средах, например при развертывании тестовой среды или при локальной разработке. Это сводит на нет одно из преимуществ технологии «конфигурация как код» (config as code), которое заключается в том, что, отслеживая конфигурацию, которую вы используете при запуске продукта в системе контроля версий, вы можете применять ее затем при разработке и тестировании. Что же можно сделать с этими жестко закодированными значениями?

Чаще всего можно обеспечить предоставление разных значений во время выполнения (развертывания ПО), обычно одним из следующих способов:

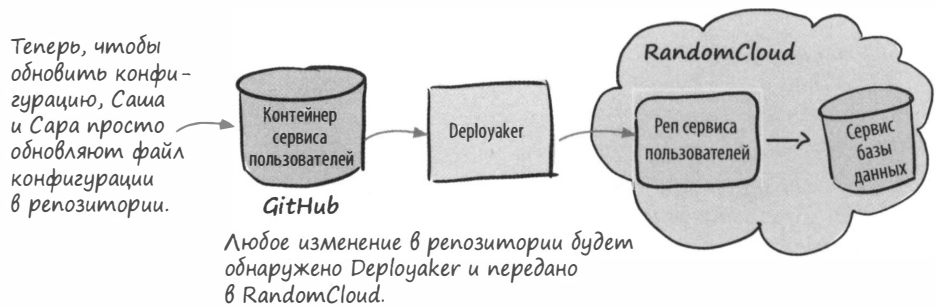
- Применить *шаблоны*: например, вместо жестко заданного `--db-host=10.10.10.10` использовать синтаксис шаблонов типа `--db-host={{ $db-host }}` и воспользоваться инструментом для заполнения значения `$db-host` в процессе развертывания.
- Применить *слои*: некоторые инструменты конфигурации позволяют задавать уровни, которые переопределяют друг друга, — например, внести в репозиторий жестко заданное значение `--db-host=10.10.10.10` для развертывания сервиса пользователей, а затем использовать инструменты для переопределения конкретных значений при запуске в другом месте (например, `--db-host=localhost:3306` или подобное при локальном запуске).

У обоих способов есть недостаток: конфигурация в системе контроля версий не полностью соответствует фактически выполняемой конфигурации. По этой причине иногда разработчики предпочитают добавлять в пайплайны этапы явной *гидратации* (полного заполнения конфигурации фактическими значениями для конкретной среды) и возврата гидратированной (hydrated) конфигурации в систему контроля версий.

Настройка Deployaker

Теперь, когда конфигурация сервиса пользователей зафиксирована на GitHub, ее больше не нужно передавать в Deployaker. Вместо этого Саша и Сара настраивают Deployaker на подключение к репозиторию сервиса пользователей на GitHub и указывают путь к файлу конфигурации этого сервиса: `user-service.yaml`.

Таким образом, Саре и Саше больше не нужно вносить изменения непосредственно в RandomCloud или Deployaker. Они коммитят изменения в репозиторий GitHub, а Deployaker забирает изменения оттуда и сбрасывает их в RandomCloud.



Стоит ли ожидать, что инструменты CD будут использовать конфигурацию, хранящуюся в системе контроля версий?

Разумеется! Многие инструменты позволяют напрямую указывать доступ к ним в файлах в системе управления версиями, или, по крайней мере, их можно настраивать программно, чтобы использовать другие инструменты для их обновления из конфигурации в системе контроля версий. На самом деле при выборе инструментов CD стоит обратить внимание на наличие такого функционала и избегать тех, которые можно настраивать только через пользовательский интерфейс.

Но что делать с конфигурацией, которая сообщает Deployaker, как найти конфигурацию сервиса в репозитории? Она тоже должна быть в системе контроля версий?

Хороший вопрос. В определенном смысле всему есть предел — *не все* можно поместить в систему контроля версий (например, конфиденциальные данные). Учитывая это, Саше и Саре было бы полезно написать хоть какую-нибудь документацию по настройке Deployaker и занести *ее* в систему контроля версий, чтобы они сами и новые члены команды понимали, как все работает, или чтобы в случае необходимости можно было снова настроить Deployaker. Кроме того, есть большая разница между настройкой Deployaker для подключения к нескольким Git-репозиториям и переносом и поддержкой на нем всей конфигурации сервиса Watch Me Watch.

Принцип «конфигурация как код»

Как конфигурация вписывается в принципы непрерывной доставки? Вспомним первое условие CD:

безопасный выпуск релизов ПО в любое время.

Многие разработчики под поставкой программного продукта подразумевают только поставку исходного кода. Но как мы увидели в начале главы, продукт включает в себя все виды текстовых данных, в том числе конфигурацию, используемую для его запуска.

Мы также рассмотрели процесс непрерывной интеграции (CI), чтобы понять, почему контроль версий имеет ключевое значение. CI — это:

Постоянный процесс внесения изменений в код,* когда каждое изменение проверяется *при добавлении его к уже имеющимся и проверенным изменениям.

> Обработка конфигурации как кода — совсем не новая идея! Возможно, вы помните из текста этой главы, что управление конфигурацией компьютера появилось как минимум в 1970-х годах. Иногда мы забываем открытия, которые уже сделаны, и вынуждены заново изобретать велосипед.

Чтобы действительно быть уверенным в том, что изменения в продукт будут внесены безопасно, необходимо собирать и проверять изменения *всех текстовых данных, из которых он состоит*, включая конфигурацию.

Практика, при которой с конфигурацией ПО обращаются так же, как с исходным кодом (то есть она хранится в системе контроля версий и проверяется посредством непрерывной интеграции), часто называется принципом «*конфигурация как код*» (config as code). Конфигурация как код — ключевой принцип CD, и реализуется он простым управлением версиями конфигурации в системе контроля версий с применением к ней, насколько это возможно, проверок, таких как линтинг, а также использованием этой конфигурации при создании тестовых сред.

В чем разница между инфраструктурой как кодом и конфигурацией как кодом?

Идея «инфраструктуры как кода» возникла первой, а «конфигурация как код» пришла ей на смену. Основной смысл принципа «*инфраструктура как код*» заключается в использовании кода/конфигурации (хранящейся в системе контроля версий) для описания инфраструктуры, на которой работает программа (например, спецификаций компьютера и конфигурации файервола). В то время как методика «*конфигурация как код*» предназначена для настройки работающего ПО, принцип «инфраструктура как код» больше связан с определением среды, в которой оно работает (и автоматизацией ее создания). Граница между этими двумя понятиями особенно размыта сегодня, когда мы в основном используем облачную инфраструктуру. Когда вы развертываете ПО в виде контейнера, вы определяете инфраструктуру или настраиваете ПО? Но основные принципы обеих методик одинаковы: рассматривайте все, что требуется для работы вашего продукта, как код: храните это в системе контроля версий и проверяйте.

Раскатка изменений в ПО и конфигурации

Сара и Саша начали работать в соответствии с принципом «конфигурация как код», сохраняя конфигурацию сервиса пользователей в Deployaker. Результаты они увидели почти сразу, через несколько недель, когда им потребовалось разделить данные, хранящиеся в одной базе данных, на две отдельные базы. Вместо одной огромной базы данных им нужна база данных пользователей и база данных фильмов. Для этого нужно внести два изменения:

1. Сервис пользователей раньше принимал только один аргумент для имени базы данных: `--db-name`; теперь он должен принимать два аргумента:

```
./user_service.py \
  --db-host=10.10.10.10 \
  --db-username=some-user \
  --db-password=some-password \
  --db-users-name=users \
  --db-movies-name=movies
```

Необходимо обновить сервис пользователей, чтобы он распознавал эти два новых аргумента.

2. Необходимо обновить конфигурацию сервиса пользователей, чтобы использовать два аргумента вместо одного текущего `--db-name`:

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=some-user
  - --db-password=some-password
  - --db-users-name=users
  - --db-movies-name=movies
```

Также необходимо обновить конфигурацию, чтобы она использовала новые аргументы.

Раньше, когда изменения вносились непосредственно в Deployaker, их приходилось раскатывать в два этапа:

1. Создать новый образ после внесения изменений в исходный код сервиса пользователей.
2. Новый образ становится несовместим с файлом конфигурации в Deployaker; развертывания выполнять невозможно, пока не будет обновлен Deployaker.

Но теперь, когда исходный код и конфигурация находятся в системе контроля версий, Саша и Сара могут сразу внести все изменения и раскатать их с помощью Deployaker!

ВАЖНО

Используйте инструменты, которые позволяют хранить конфигурации в системе контроля версий. Некоторые инструменты предусматривают возможность работы с их конфигурацией только через пользовательский интерфейс (например, через сайт или интерфейс командной строки (Command Line Interface, CLI)); это удобно, если требуется быстрая настройка и запуск, но в долгосрочной перспективе, чтобы реализовать непрерывную доставку, может понадобиться хранить конфигурацию в системе контроля версий. Избегайте инструментов, которые не позволяют этого делать.

ВАЖНО

Рассматривайте все текстовые данные своего программного продукта как код и храните их в системе контроля версий. Это создаст некоторые сложности, связанные с конфиденциальными данными и значениями, специфичными для среды, но дополнительный инструментарий, который вам понадобится, стоит затраченных на него усилий. Храня все в системе контроля версий, вы можете быть уверены, что система всегда находится в безопасном для выпуска релиза состоянии, причем все сопутствующие данные, а не только исходный код.

Заключение

Несмотря на то что работа над проектом Watch Me Watch только начинается, Сара и Саша быстро поняли, насколько важна система контроля версий для реализации непрерывной доставки. Они выяснили, что система контроля версий не просто пассивное хранилище, в нем разворачивается первый этап CD: именно здесь происходит объединение изменений кода, которые запускают проверку того, что продукт остается готовым к релизу.

Хотя изначально в системе контроля версий хранился только исходный код, Сара и Саша поняли, что очень полезно хранить там еще и конфигурацию и работать с ней как с кодом.

По мере роста компании они будут продолжать использовать систему контроля версий в качестве единого источника истины для своего продукта. С этого момента любая автоматизация процессов, начиная с автоматического выполнения юнит-тестов и заканчивая канареечным развертыванием (canary deployment), будет основываться на изменениях, внесенных в эту систему.

Итоги

- Чтобы работать с непрерывной доставкой, необходимо использовать систему контроля версий.
- Пайплайны CD должны запускаться при каждом внесении изменений в систему контроля версий.
- Система контроля версий — это источник истины о состоянии продукта, а также основа всей автоматизации процесса CD, описанного в этой книге.
- Применяйте принцип «конфигурация как код» и храните все текстовые данные, описывающие продукт (не только исходный код, но и конфигурацию), в системе контроля версий. Избегайте инструментов, которые не позволяют этого делать.

Далее...

В следующей главе мы рассмотрим, как использовать линтинг в пайплайнах CD, чтобы избежать распространенных ошибок и обеспечить соблюдение стандартов качества во всех унаследованных проектах, даже если над ними работает большое количество участников.

Эффективное использование линтинга | 4

В этой главе

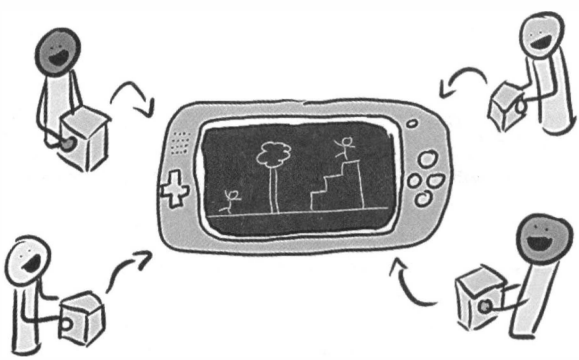
- ✓ Проблемы в коде, которые выявляет линтинг: баги, ошибки и проблемы стиля
- ✓ Полное отсутствие проблем — это идеально, но необходимо учитывать реальное состояние унаследованного кода
- ✓ Как проводить линтинг большого объема кода, используя итеративный подход
- ✓ Риски появления новых ошибок при решении проблем

Приступим наконец к созданию пайплайнов! Линтинг — ключевой компонент непрерывной интеграции пайплайна: он позволяет выявлять и помечать известные проблемы и нарушения стандартов написания кода, чтобы уменьшить количество багов в коде и облегчить его сопровождение.

Бекки и проект Super Game Console

Бекки недавно присоединилась к команде Super Game Console и очень этому рада! Игровая консоль Super Game Console очень популярна, она служит для запуска простых игр на Python. Главная ее особенность — это наличие огромной библиотеки игр на языке Python, в которую может внести свой вклад любой желающий.

В компании Super Game Console разработан процесс подачи заявок, который позволяет каждому, от любителя до профессионала, зарегистрироваться в качестве разработчика и прислать свои игры.



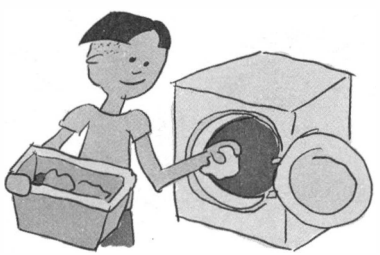
Но в этих играх *очень много* багов, и это настоящая проблема. Бекки и Рамон, который дольше работает в команде, тщетно пытаются разобраться с этим завалом. Бекки заметила несколько характерных моментов:

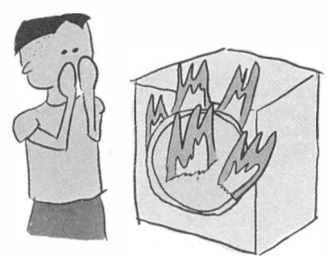
- Некоторые игры вообще не компилируются! А множество других багов возникает из-за простейших ошибок, таких как попытка использовать неинициализированные переменные.
- Большинство ошибок на самом деле не критичны, а просто сильно мешают работе Бекки (например, неиспользуемые переменные).
- Код каждой игры отличается от предыдущей. Такая несогласованность в стилях затрудняет отладку.

Линтинг спешит на помощь!

Анализируя характер проблем, вызвавших баги, которые они с Рамоном устраняли, Бекки понимает, что они очень напоминают те, которые обычно отлавливают линтеры.

Что вообще такое *линтинг*? Это процесс поиска линтов (lint)с помощью линтера (linter)! А что такое линт? Вы можете подумать, что это ворс[1], который накапливается в сушилке для белья.





Сами по себе отдельные ворсинки не вызывают проблем, но накопившись со временем, они могут помешать нормальной работе сушилки. В итоге, если долго не обращать на них внимания, ворсинок становится все больше, и они могут загореться от горячего воздуха в сушилке!

То же самое происходит с ошибками и неувязками в программах: они на первый взгляд кажутся незначительными, но со временем накапливаются. Как в случае с Бекки и Рамоном, код, который они просматривают, не согласован и полон мелких ошибок. Это не только приводит к появлению багов, но и мешает эффективно поддерживать код.

[1] Lint (*англ.*) — ворс. — *Примеч. пер.*

Вся правда о линтинге

Линтеры бывают разных видов и размеров. Поскольку они анализируют код и взаимодействуют с ним, они обычно ориентированы на конкретный язык (например, Pylint для Python). Некоторые линтеры применимы абсолютно ко всему, что можно делать на языке, а некоторые специфичны для конкретных задач и инструментов: например, линтеры для эффективной работы с библиотеками HTTP.

Мы остановимся на линтерах, которые применимы к языку в целом. Разные линтеры по-разному классифицируют возникающие проблемы, но все их можно отнести к одной из трех групп. Разберемся, что за проблемы обнаружила Бекки и к каким группам они относятся:

Баги: неправильно написанный код, который работает не так, как нужно!

- Некоторые игры вообще не компилируются! А множество других багов возникает из-за простейших ошибок, таких как попытка использовать неинициализированные переменные.

Ошибки: некорректное использование кода, которое не влияет на работу программы.

- Большинство ошибок на самом деле не критичны, а просто сильно мешают работе Бекки (например, неиспользуемые переменные).
- Код каждой игры отличается от предыдущей. Такая несогласованность в стилях затрудняет отладку.

Нарушения стиля: непоследовательные стилевые решения и «запахи» кода.

В чем разница между статическим анализом и линтингом?

Статический анализ позволяет проводить разбор кода без его выполнения. В следующей главе мы поговорим о тестах, которые представляют собой своего рода *динамический анализ*, поскольку они требуют выполнения кода. *Линтинг* — это разновидность статического анализа; термин *статический анализ* в общем случае включает в себя множество различных способов анализа кода. В контексте непрерывной доставки бóльшая часть статического анализа, который мы будем обсуждать, выполняется с помощью линтеров. В остальном различие не так важно. Название *линтер* происходит от одноименного инструмента, созданного компанией Bell Labs в 1978 году. В большинстве случаев, особенно в рамках CD, термины *статический анализ* и *линтинг* можно использовать как взаимозаменяемые, однако некоторые формы статического анализа выходят за рамки возможностей линтеров.

Сказ о Pylint и множестве ошибок

Поскольку все игры для Super Game Console написаны на Python, Бекки и Рамон решили, что лучше всего начать с использования инструмента Pylint. Вот структура кода Super Game Console:

```
console/
docs/
games/
test/
setup.py
LICENSE
README.md
requirements.txt
```

В каталоге games хранятся все игры, присланные разработчиками.

В папке games тысячи игр. Бекки не терпится узнать, что Pylint сможет поведать о них ей и Рамону. Бекки с замиранием сердца вводит команду и нажимает Enter...

```
$ pylint games
```

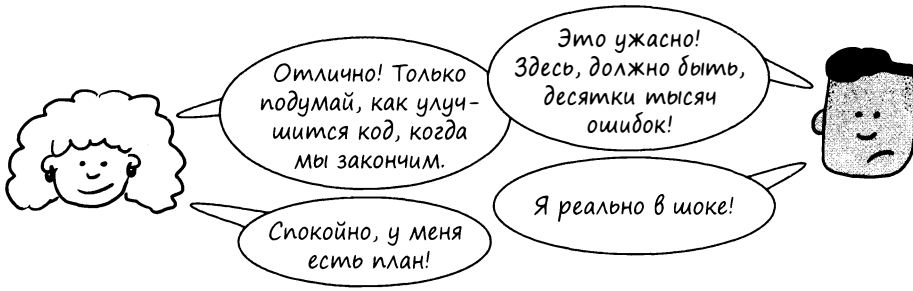
И в награду на экран выводится множество предупреждений и сообщений об ошибках! Вот лишь малая часть того, что она увидела:

```
games/bridge.py:40:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
games/bridge.py:41:0: W0311: Bad indentation. Found 4 spaces, expected 8 (bad-indentation)
games/bridge.py:46:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
games/bridge.py:1:0: C0114: Missing module docstring (missing-module-docstring)
games/bridge.py:3:0: C0116: Missing function or method docstring (missing-function-docstring)
games/bridge.py:13:15: E0601: Using variable 'board' before assignment (used-before-assignment)
games/bridge.py:8:2: W0612: Unused variable 'cards' (unused-variable)
games/bridge.py:23:0: C0103: Argument name «x» doesn't conform to snake_case naming style (invalid-name)
games/bridge.py:23:0: C0116: Missing function or method docstring (missing-function-docstring)
games/bridge.py:26:0: C0115: Missing class docstring (missing-class-docstring)
games/bridge.py:30:2: C0116: Missing function or method docstring (missing-function-docstring)
games/bridge.py:30:2: R0201: Method could be a function (no-self-use)
games/bridge.py:26:0: R0903: Too few public methods (1/2) (too-few-public-methods)
games/snakes.py:30:4: C0103: Method name «do_POST» doesn't conform to snake_case naming style (invalid-name)
games/snakes.py:30:4: C0116: Missing function or method docstring (missing-function-docstring)
games/snakes.py:39:4: C0103: Constant name «httpd» doesn't conform to UPPER_CASE naming style (invalid-name)
games/snakes.py:2:0: W0611: Unused import logging (unused-import)
games/snakes.py:3:0: W0611: Unused argv imported from sys (unused-import)
```

А как обстоят дела с другими языками и линтерами?

Бекки и Рамон используют Python и Pylint, но принципы этого инструмента применимы к любому языку или линтеру. Все хорошие линтеры должны предоставлять такую же гибкость конфигурации, которую мы продемонстрируем на примере Pylint, и должны обнаруживать такое же разнообразие проблем.

Унаследованный код: системный подход



При первом запуске линтера количество найденных им ошибок может вас ошеломить. (Скоро мы поговорим о том, что делать, если вам не нужно работать со всем объемом кода.)

К счастью, Бекки уже работала с линтингом унаследованного кода и разработала системный подход, который позволит им с Рамоном ускорить процесс и рационально распределить время:

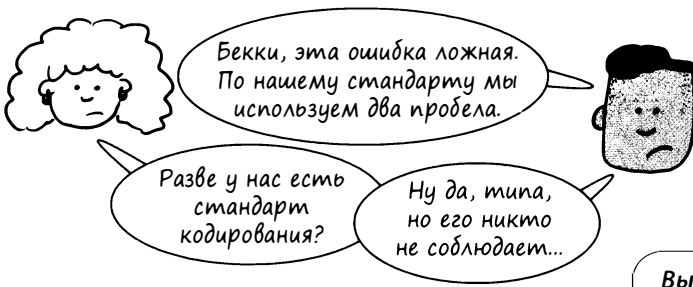
1. Прежде чем что-то делать, необходимо настроить инструменты линтинга. Опции, которые Pylint применяет по умолчанию, могут не подойти для Super Game Console.
2. Необходимо определить исходный уровень ошибок и отслеживать его. Бекки и Рамону не обязательно исправлять каждую ошибку; если им достаточно убедиться, что количество ошибок со временем уменьшается, то этот результат и нужно считать удовлетворительным.
3. Получив результаты, в следующий раз, когда разработчик пришлет новую игру, Бекки и Рамон могут снова провести измерения и остановить процесс регистрации игры, если в ней появились новые ошибки. Таким образом, количество ошибок никогда не будет увеличиваться.
4. На данном этапе Бекки и Рамон добились того, что ситуация не ухудшится; с учетом этого они могут приступить к решению имеющихся проблем. Бекки знает, что не все выявленные линтером проблемы одинаковы, поэтому они с Рамоном будут действовать по принципу «разделяй и властвуй», чтобы максимально эффективно использовать драгоценное время.

Бекки знает, что им не нужно исправлять все ошибки: просто предотвратив появление новых, они уже сделали половину дела. И действительно, не всегда можно и нужно исправлять все ошибки.

Шаг 1. Настройка линтера в соответствии со стандартами написания кода

Рамон изучает часть ошибок, которые выдает Pylint, и замечает, что линтер реагирует на то, что отступ должен быть в четыре пробела вместо двух:

```
bridge.py:2:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
```



Часто стандарты кодирования не автоматизированы, поэтому Бекки не удивлена. Но к счастью, стандарты кодирования (которые все игнорируют) содержат большую часть информации, нужной Бекки и Рамону, а именно:

- Отступ табуляцией или пробелами? Если пробелами, то сколько?
- Как именовать переменные — в стиле **`snake_case`** или **`camelCase`**?
- Существует ли максимальная длина строки? И какова она?

> ***Для соблюдения стандартов кодирования необходима автоматизация***
>
> В отсутствие автоматизации разработчикам приходится помнить о соблюдении стандартов, а ревьюерам — не забыть их проверить. Люди есть люди: мы можем что-то упустить! Но машины — нет: линтеры позволяют автоматизировать применение стандартов кодирования, чтобы никому не приходилось думать об их выполнении.

> ***Вы уже умеете настраивать линтер?***
>
> Тогда можете пропустить эту страницу! Прочитайте ее, если вы никогда раньше не настраивали инструменты линтинга.

> ***На что обращать внимание***
>
> При выборе инструментов для линтинга обращайте внимание на возможность их настройки. Не весь унаследованный код и не все команды разработчиков одинаковы, поэтому важно иметь возможность настраивать линтер. Линтеры должны работать на вас, а не наоборот!

Ответы на эти вопросы можно загрузить в Pylint в виде параметров конфигурации в файле, который обычно называется .pylintrc.

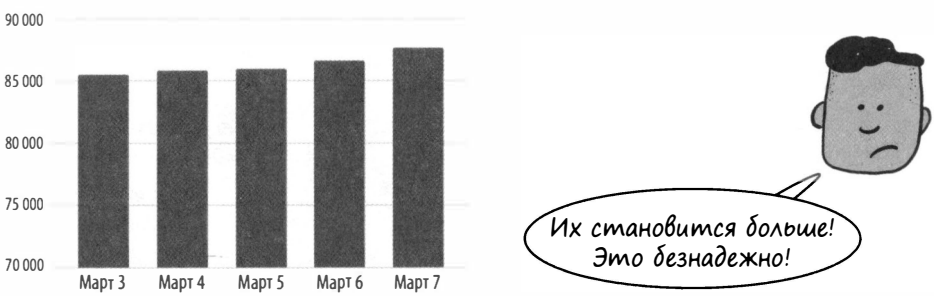
В существующем стандарте Бекки не нашла всего, что ей было нужно, поэтому им с Рамоном иногда приходилось самим принимать решения. Они привлекли к обсуждению остальных членов команды Super Game Console, но не смогли договориться по ряду пунктов; в конце концов Бекки и Рамон просто приняли волевое решение. В случае сомнений они обращались к идиомам языка Python, что чаще всего означало придерживаться настроек Pylint по умолчанию.

Шаг 2. Определение базового уровня

После настройки Pylint в соответствии с существующим и новым стандартом кодирования ошибок стало немного меньше, но их по-прежнему десятки тысяч.

Бекки знает, что даже если они с Рамоном оставят код как есть и просто будут указывать количество ошибок при его изменении, это мотивирует команду делать меньше ошибок. А на следующем этапе они будут использовать эти данные, чтобы остановить рост количества ошибок.

Бекки пишет скрипт, который запускает Pylint и подсчитывает количество ошибок, о которых он сообщает. Она создает пайплайн, который запускается каждую ночь, и публикует эти данные в хранилище BLOB-объектов. Через неделю она собирает данные и строит график, показывающий динамику изменения количества ошибок:



Количество ошибок продолжает расти, потому что пока Бекки и Рамон работают над этой проблемой, разработчики с энтузиазмом шлют новые игры и обновления на Super Game Console. Каждая новая игра и новое обновление содержат новые ошибки.

Нужно ли создавать скрипт самостоятельно?

Да, в данном случае, скорее всего, придется. Бекки пришлось самой написать инструмент, измеряющий исходное количество ошибок и отслеживающий изменение этого количества с течением времени. Если вам нужен такой же, скорее всего, вам придется создать его самостоятельно. Это будет зависеть от используемого языка и доступных инструментов. Вы также можете подписаться на сервисы, которые будут отслеживать для вас эту информацию. В большинстве систем CD такая функциональность не предусмотрена, поскольку она зависит от языка и специфики конкретной предметной области.

Шаг 3. Принудительная проверка при поступлении данных

Рамон заметил, что по мере поступления новых данных количество ошибок, которые находит Pylint, увеличивается, но у Бекки есть решение: блокировать данные, которые увеличивают количество ошибок. Это означает введение нового правила для каждого пул-реквеста (PR):

Каждый PR должен уменьшать количество ошибок линтинга или оставлять его на прежнем уровне.

Бекки создает соответствующий скрипт для пайплайна, который Super Game Console запускает для всех PR:

```
# при запуске пайплайна он будет передавать в качестве аргументов
# пути к файлам, которые изменились в PR
paths_to_changes = get_arguments()

# запустите линтинг для файлов, которые изменились, чтобы посмотреть,
# сколько ошибок обнаружено
problems = run_lint(paths_to_changes)

# Бекки создала пайплайн, который запускается каждую ночь и записывает
# количество обнаруженных ошибок в хранилище BLOB-объектов;
# скрипт линтинга загружает эти данные
known_problems = get_known_problems(paths_to_changes)

# сравнение количества ошибок, найденных в измененном коде,
# с количеством ошибок, найденных прошлой ночью
if len(problems) > len(known_problems):
  # не выполнять слияние для данного PR, если оно
  # увеличивает количество ошибок линтинга
  fail("number of lint issues increased from {} to {}".format(
    len(known_problems), len(problems)))
```

Следующим шагом Бекки добавляет этот скрипт в существующий пайплайн, который запускается для каждого PR.

Что еще за PR?

PR означает *пул-реквест*, или *запрос на вытягивание* (*pull request*), способ предложить и просмотреть изменения в коде. Подробнее см. главу 3.

Разве не нужно отслеживать что-то еще, кроме количества ошибок?

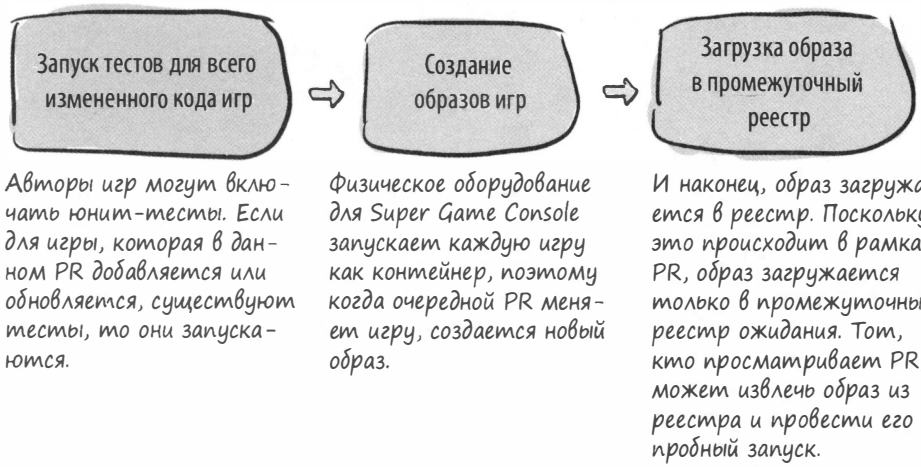
Действительно, при простом сравнении количества ошибок что-то можно упустить; например, в изменении может не быть прежней ошибки, но при этом появится новая. Однако в данной ситуации действительно важна только общая тенденция, а не конкретные проблемы.

Новая или небольшая кодовая база

Чуть ниже мы поговорим об этой теме подробнее, но если вы работаете с небольшой или совсем новой кодовой базой, вы можете пропустить определение базового уровня и сразу подчистить все найденные ошибки. Тогда вместо того, чтобы добавлять в пайплайн проверку того, что количество ошибок не увеличивается, добавьте проверку, которая не пройдет, если линтинг обнаружит новые ошибки.

Добавление принудительной проверки в пайплайн

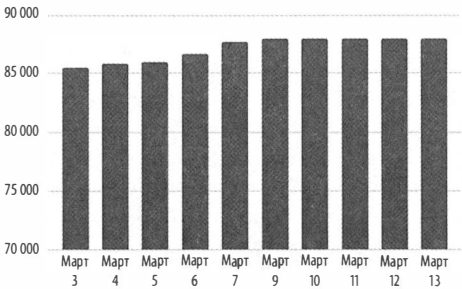
Бекки нужно, чтобы новая проверка выполнялась каждый раз, когда разработчик отправляет новую игру или обновление для существующей игры. Сервис Super Game Console добавляет новые игры через пул-реквест (PR) к своему GitHub-репозиторию. Компания уже предоставляет разработчикам возможность включать свои тесты в игры, и они запускают эти тесты при каждом PR. Вот как выглядит пайплайн до изменений Бекки:



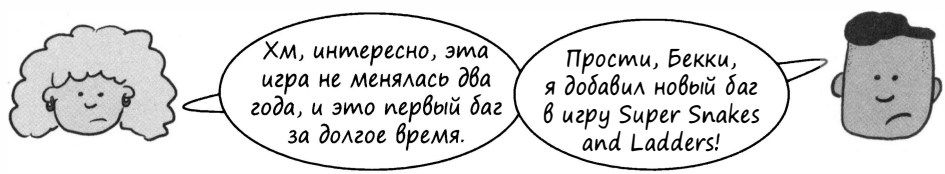
Бекки хочет добавить новую проверку в пайплайн, который Super Game Console запускает для каждого PR.

Теперь каждый раз, когда разработчик будет открывать PR для добавления или изменения игры на Super Game Console, будет запускаться скрипт Бекки. Если открытый PR увеличит количество ошибок линтинга, пайплайн остановится. И прежде чем он продолжит создание образов, разработчик должен будет исправить ошибки.

Шаг 4. «Разделяй и властвуй»



Бекки и Рамон остановили развитие проблемы. Теперь напряжение снято, и они могут спокойно заняться исправлением существующих ошибок, будучи уверенными, что новых не прибавится. Пришло время исправлять ошибки! Но у Рамона сразу же появляется проблема...



Внесение *любых* изменений, в том числе тех, которые исправляют ошибки, найденные при линтинге, чревато появлением новых ошибок. Так почему же мы все-таки проводим линтинг? Потому что его польза перевешивает риск! И он имеет смысл, только если это действительно так. Проанализируем преимущества и риски при исправлении ошибок, выявленных при линтинге:

Преимущества	Риски
Линтинг отлавливает баги	Внесение изменений может привести к появлению новых багов
Линтинг помогает устранить несущественные ошибки	Исправление ошибок, найденных при линтинге, отнимает время
Последовательный код легче поддерживать	

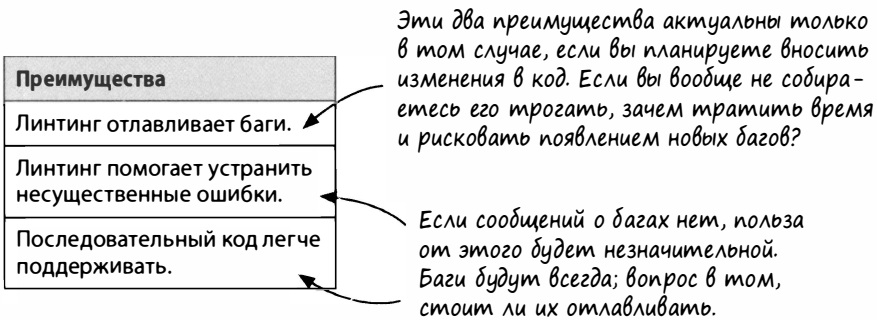
Из таблицы можно сделать несколько интересных выводов. Первое преимущество — это обнаружение багов, и его нужно сопоставить с первым риском — появлением новых.

Рамон внес новый баг в игру, в которой раньше их не находили. Стоило ли идти на риск добавления нового бага в игру, которая, судя по всему, отлично работала? Возможно, нет!

Два других преимущества актуальны, только если требуется изменять код. Если код не нужно менять ни сейчас, ни в будущем, не имеет значения, сколько в нем мелких ошибок и насколько он непоследователен.

Рамон занимался обновлением игры, в которую два года не вносили изменения. Стоило ли тратить время и рисковать появлением новых багов в игре, которая долго не обновлялась? Скорее всего, нет! Нужно найти способ изолировать эти игры, чтобы не тратить на них время.

Изолирование: не все нужно исправлять



Бекки и Рамон просматривают все игры в библиотеке и выявляют те, которые меняются меньше всего. Всем им больше года, и разработчики перестали их обновлять. Коллеги также изучают количество ошибок, о которых сообщают пользователи этих игр. Они выбирают игры, которые не менялись больше года и для которых нет сообщений об ошибках, и перемещают их в отдельную папку. Их код теперь выглядит так:

```
.pylintrc
console/
docs/
games/
  frozen/
  ...
test/
setup.py
LICENSE
README.md
requirements.txt
```

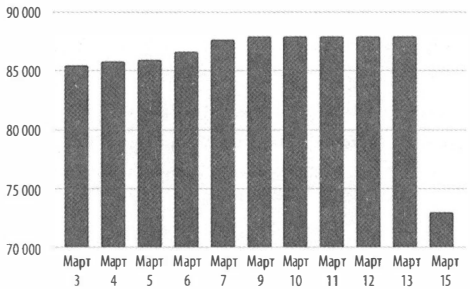
Файл конфигурации для Pylint, созданный Бекки и Рамоном на шаге 1.

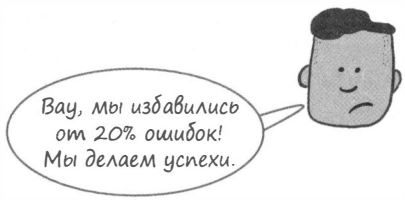
Эти игры не обновлялись больше года, и в них нет известных багов. Изменений в них не ожидается, поэтому их можно исключить из проверки линтером.

Бекки и Рамон обновляют .pylintrc, исключая из анализа игры, помещенные в каталог frozen/:

```
[MASTER]
ignore=games/frozen
```

Принудительное изолирование



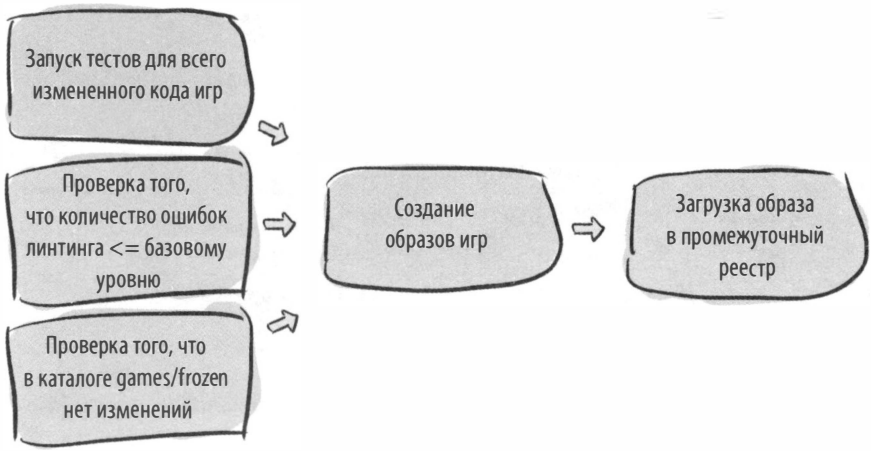


А для большей безопасности Бекки создает новый скрипт, который следит, чтобы никто не вносил изменения в игры из каталога frozen/:

```
# при запуске пайплайна он будет передавать в качестве
# аргументов пути к файлам, которые были изменены в PR
paths_to_changes = get_arguments()

# вместо жесткого кодирования этого скрипта
# для отслеживания изменений в games/frozen
# загрузим игнорируемые каталоги из
# .pylintrc, чтобы проверка стала универсальной.
ignored_dirs = get_ignored_dirs_from_Pylintrc()

# проверка на наличие изменений по всем путям,
# находящимся в игнорируемых каталогах
ignored_paths_with_changes = get_common_paths(
  paths_to_changes, ignored_dirs)

if len(ignored_paths_with_changes) > 0:
  # не выполнять слияние для PR, если он
  # предполагает изменения в игнорируемых каталогах.
  fail("для {} не выполняется линтинг, "
   "поэтому изменения не допускаются".format(
   ignored_paths_with_changes))
```

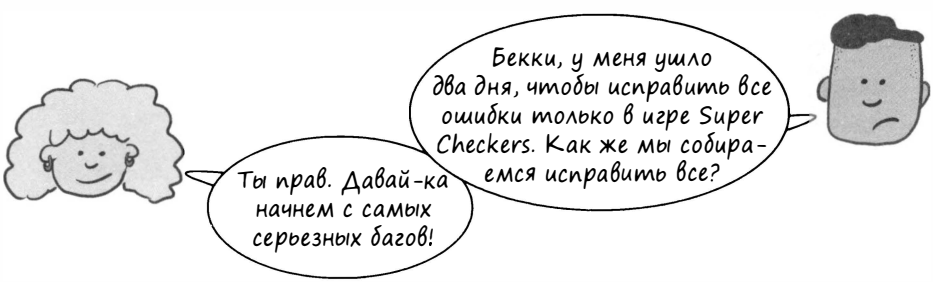
> ***Но что, если потребуется изменить «замороженную» игру?***
>
> Сообщение об ошибке должно содержать рекомендации о том, что следует делать при необходимости изменить игру. Разработчик, желающий внести изменения, должен переместить игру из папки frozen в рабочую среду и обработать все ошибки, которые, несомненно, будут найдены при линтинге.

Затем она добавляет его в пайплайн, который работает с PR:



Не все проблемы одинаковы

Итак, неужели *теперь* можно приступить к исправлению ошибок? Рамон с головой погружается в работу, но уже через два дня его силы на исходе:

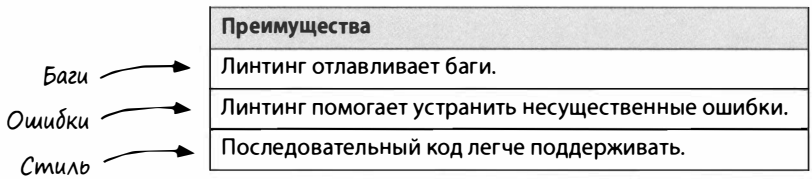


Бекки и Рамон хотят сначала исправить самое важное. Проанализируем еще раз преимущества и риски исправления ошибок линтинга, чтобы понять, как это сделать:

Преимущества	Риски
Линтинг отлавливает баги	Внесение изменений может привести к появлению новых ошибок
Линтинг помогает устранить несущественные ошибки	Исправление ошибок, выявленных при линтинге, отнимает время
Последовательный код легче поддерживать	

Рамон сталкивается со вторым типом рисков: у него уходит слишком много времени на исправление всех ошибок. А у Бекки есть встречное предложение: сначала исправить самые серьезные ошибки. Таким образом, они потратят время максимально эффективно, и им не потребуется исправлять абсолютно все.

Так какие же проблемы нужно решить в первую очередь? Преимущества линтинга касаются всех видов ошибок, которые он находит:



Какие проблемы выявляет линтинг

Проблемы, которые находят линтеры, можно разделить на три вида: баги, ошибки и стиль.

Баги, обнаруживаемые с помощью линтинга, — это типичные случаи неправильного использования кода, из-за чего он работает неверно. Например:

- неинициализированные переменные;
- несоответствие форматов переменных.

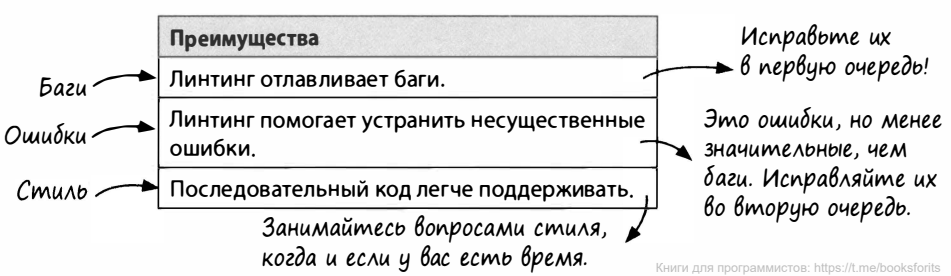
Ошибки, выявляемые линтингом, — это типичные ошибки в коде, которые не влияют на работу кода, однако могут вызывать проблемы с производительностью или затрудняют поддержку кода. Например:

- неиспользуемые переменные;
- псевдонимы переменных.

И наконец, проблемы *стиля* — это непоследовательное применение стилевых решений и запахи кода. Например:

- длинные сигнатуры функций;
- непоследовательный порядок импорта.

Наверное, было бы неплохо исправить все эти проблемы, однако если бы у вас было время только на один их вид, какой бы вы выбрали? Наверное, баги? Логично, ведь они влияют на работу продуктов! Вот как выглядит иерархия:



Сначала баги, потом стиль

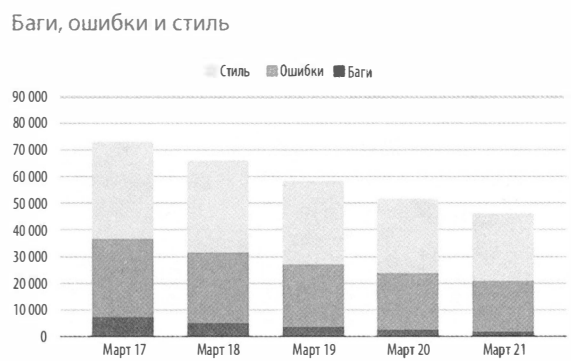
Бекки рекомендует Рамону исправлять проблемы, найденные линтером, систематически. Тогда, если им придется переключиться на другой проект, они будут знать, что не зря потратили время на устранение проблем. Возможно, они даже решат ограничить свои усилия по времени: посмотреть, сколько ошибок они смогут исправить за две недели, а затем двигаться дальше.

Как определить, к какому виду относится проблема? Многие инструменты линтинга классифицируют найденные проблемы. Посмотрим еще раз на некоторые из них, обнаруженные с помощью Pylint:

```
games/bridge.py:46:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
games/bridge.py:1:0: C0114: Missing module docstring (missing-module-docstring))
games/bridge.py:13:15: E0601: Using variable 'board' before assignment (used-before-assignment)
games/bridge.py:8:2: W0612: Unused variable 'cards' (unused-variable)
games/bridge.py:30:2: R0201: Method could be a function (no-self-use)
games/bridge.py:26:0: R0903: Too few public methods (1/2) (too-few-public-methods)
games/snakes.py:30:4: C0103: Method name "do_POST" doesn't conform to snake_case naming style (invalid-name)
```

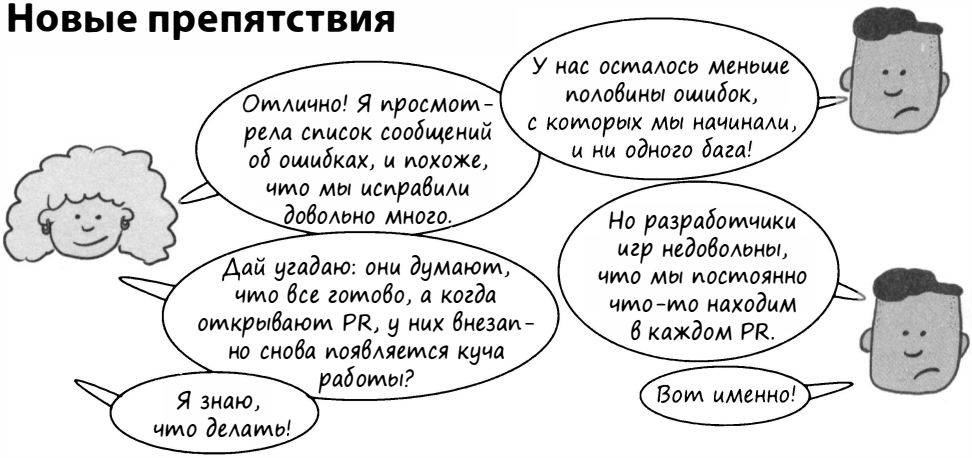
Каждой проблеме присвоена буква и номер. Pylint распознает четыре категории проблем: E — *ошибка (error)*, то есть то, что мы называем *багами*; W — *предупреждение (warning)* — то, что мы называем *ошибками*. Две последние категории, C — *соглашение (convention)* и R — *рефакторинг (refactor)*, мы относим к *стилю*.

Рамон создает скрипт и отслеживает количество проблем каждого вида на протяжении всей следующей недели:



Общее количество проблем остается довольно высоким, однако количество багов — самых важных проблем, выявляемых линтингом, — неуклонно снижается!

Новые препятствия



Бывает обидно, когда кажется, что все готово, а на самом деле приходится преодолевать новые трудности. Решение довольно простое: включите линтеры в процесс разработки. Как это сделать и как облегчить задачу разработчикам? Просто выполните следующее:

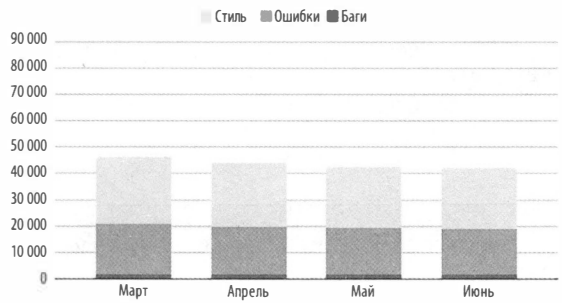
1. Разместите файлы конфигурации линтинга в репозитории с кодом. Бекки и Рамон внесли код .pylintrc, который они используют, прямо в репозиторий Super Game Console. Теперь разработчики могут применять конфигурацию, которая используется пайплайном CD, и избежать сюрпризов.
2. Запускайте линтер во время работы. Его можно запустить и вручную, но проще всего это сделать с помощью интегрированной среды разработки (integrated development environment, IDE). Большинство IDE и даже такие редакторы, как Vim, позволяют интегрировать линтеры и запускать их в процессе работы. Таким образом, если вы допустите ошибку, вы сразу же об этом узнаете.

Бекки и Рамон рассылают уведомление всем разработчикам, с которыми они сотрудничают, с рекомендацией включить линтинг в свою IDE. Кроме того, они добавляют для них напоминалку, появляющуюся, когда линтинг в PR не проходит: о том, что разработчики могут включить у себя эту функцию.

> *А как насчет форматтеров?*
>
> В некоторых языках довольно легко избежать многих ошибок стиля, используя инструменты, называемые *форматтерами*, которые автоматически форматируют код во время работы. Они могут взять на себя такие функции, как обеспечение правильного порядка импорта и согласованности интервалов. Форматтер может избавить вас от многих неприятностей! Обязательно интегрируйте инструмент форматирования в свою IDE. Запустите форматтер в пайплайне и сравните полученный результат с поставленным кодом.

Унаследованный и идеальный код

Бекки и Рамон не могут исправить все ошибки, потому что большой объем кода уже существовал до того, как они начали использовать линтинг. Это значит, что они должны продолжать отслеживать базовый уровень ошибок и проверять, чтобы количество ошибок не увеличивалось, или же им придется постоянно подстраивать конфигурацию Pylint, чтобы он игнорировал ошибки, с которыми они решили смириться.



А как выглядит идеал? Если бы Бекки и Рамон могли проводить линтинг в течение неограниченного количества времени, какого результата они хотели бы достичь? Если вам посчастливилось работать над совершенно новой или относительно небольшой кодовой базой, идеал должен быть таким:

При анализе кода линтер не выдает сообщений об ошибках.

Разумна ли эта цель? Да! И даже если вы никогда ее не достигнете, мечтать о полете к звездам и посадке на Луну не так уж плохо.

Если вы работаете с новой или небольшой кодовой базой, вам не обязательно делать все то, что делали Бекки и Рамон. Заметьте, что на шагах 2 и 3 Бекки и Рамон уделяют много времени измерению и отслеживанию базового уровня. Вместо этого лучше исправьте все ошибки. Вы по-прежнему можете применять очередность их исправления, описанную на шаге 4, так что если вас по какой-то причине прервут, вы все равно сначала разберетесь с самыми важными из них. Но ваша глобальная цель — добиться полного отсутствия ошибок.

Вы можете применить проверку, аналогичную той, которую Бекки и Рамон добавили на шаге 3, но вместо того, чтобы сравнивать количество ошибок линтинга с базовым уровнем, задайте условие его равенства нулю!

Заключение

В проекте Super Game Console накопилось огромное количество багов и других проблем, а отсутствие единого стиля написания кода затрудняло поддержку игр. Несмотря на то что объем кодовой базы был очень большим, Бекки смогла добавить линтинг в процессы таким образом, что он принес ощутимую пользу. Она сделала это, подойдя к проблеме итеративно.

После переопределения стандартов кодирования Бекки вместе с Рамоном измерила количество ошибок, выявляемых при линтинге, и добавила в пайплайн PR-тесты, проверяющие, что оно не увеличивается. Когда Бекки и Рамон начали исправлять ошибки, они поняли, что не все проблемы одинаково важны, поэтому основные усилия они приложили к исправлению кода, который, скорее всего, будет меняться, и стали решать проблемы в порядке их важности.

Итоги

- Линтинг выявляет баги и помогает поддерживать согласованность и удобство сопровождения кода.
- Идеальный результат — когда инструменты линтинга не выявляют ошибок. Если унаследованного кода очень много, можно ограничиться условием, чтобы ошибок не становилось больше.
- Изменение кода всегда сопряжено с риском добавления новых ошибок, поэтому важно быть осторожным и тщательно обдумывать, стоит ли вносить изменения. Если код меняется часто и/или содержит много известных багов, то, вероятно, сто́ит. В противном случае можно изолировать код и оставить его в покое.
- Линтинг обычно выявляет три вида проблем, и не все они одинаково важны. Баги почти всегда следует исправлять. Ошибки могут привести к появлению багов и усложнить поддержку кода, но они не так важны, как баги. Наконец, исправление проблем со стилем упрощает работу с кодом, но они не столь важны, как баги и ошибки.

Далее...

В следующей главе мы рассмотрим, как работать с нестабильными наборами тестов. Мы подробно разберем, почему тесты становятся нестабильными и какой сигнал от них нам на самом деле необходим.

Работа с нестабильными тестами | 5

В этой главе

- ✓ Почему тесты чрезвычайно важны для CD
- ✓ Пошаговый план перехода от сбоев нестабильных (noisy) тестов к полезным и информативным сигналам
- ✓ Что делает тесты нестабильными
- ✓ Сбои при тестировании — это баги
- ✓ Что такое ненадежные (flaky) тесты и почему они вредны
- ✓ Как правильно проводить повторные тесты

Непрерывную доставку практически невозможно реализовать без тестов! Для многих тесты являются синонимом, по крайней мере, непрерывной интеграции, но со временем некоторые наборы тестов теряют актуальность. В этой главе мы рассмотрим, как работать с нестабильными наборами тестов.

Непрерывная доставка и тесты

Как тесты вписываются в непрерывную доставку? Из главы 1 мы знаем два условия CD:

- существует возможность выпускать релизы ПО безопасно и в любое время;
- выпуск можно делать максимально просто, буквально одним нажатием кнопки.

Как узнать, что изменения можно вносить безопасно? Вы должны быть уверены, что ваш код делает то, что нужно. Это можно гарантировать, только протестировав его. Тесты подтвердят, что код делает именно то, что вы задумали.

Словарик

Набор тестов (test suite) — это группа тестов. Часто это несколько тестов, которые проверяют конкретную часть программного продукта.

Эта книга не научит вас писать тесты; этой теме посвящено множество другой замечательной литературы. Я буду исходить из того, что вы не только знаете, как писать тесты, но и что большинство современных программных проектов включают хотя бы *несколько* тестов (если в вашем проекте это не так, как можно скорее добавьте в него тесты).

В главе 3 говорилось о важности постоянной проверки каждого изменения. *Чрезвычайно важно запускать тесты не просто регулярно, а при каждом изменении.* Это хорошо работает, когда проект новый и в нем всего несколько тестов, но по мере развития проекта наборы тестов увеличиваются и со временем становятся медленными и менее надежными. В этой главе я покажу, как поддерживать тесты с течением времени, чтобы они оставались эффективными и обеспечивали постоянную готовность кода к релизу.

Как QA вписывается в CD?

При всем внимании к автоматизации тестирования может возникнуть вопрос: значит ли, что если работать с CD, то контроль качества (quality assurance, QA) не нужен? Нет, не значит! Важно позволить людям делать то, что у них получается лучше всего: исследовать и мыслить нестандартно. Автоматизируйте то, что можно, но автоматические тесты всегда будут делать только то, что вы им скажете. Если вы хотите обнаружить новые проблемы, о которых вы даже не догадываетесь, вам понадобятся люди, выполняющие QA!

Сбой в работе сервиса Ice Cream for All

Очень успешная компания по доставке мороженого Ice Cream for All — одна из тех, кто уделяет внимание тестам. Уникальность ее коммерческого предложения в том, что она связывает покупателей напрямую с поставщиками в их районе, чтобы можно было заказать свое любимое мороженое и получить его с доставкой на дом за несколько минут!

Компания Ice Cream for All связывает пользователей с тысячами поставщиков мороженого. Для этого сервис Ice Cream должен иметь возможность подключаться к уникальному API каждого поставщика.

Словарик

Ретроспективный отчет, который иногда называют «*разбор полетов*» (или *postmortem*), — это возможность проанализировать процесс, когда что-то идет не так, и решить, что можно улучшить на будущее.

4 июля[1] — день максимальной нагрузки для Ice Cream for All. Каждый год 4 июля компания получает больше всего заказов на мороженое. Но в этом году именно в часы максимальной нагрузки в работе произошел серьезный сбой! Сервис Ice Cream не работал больше часа.

Команда сервиса составила ретроспективный отчет, чтобы понять, что пошло не так, и исправить это на будущее, а в комментариях прошла интересная дискуссия:

Ретроспективный отчет: сбой сервиса Ice Cream 4 июля

Результаты:
80% запросов сервиса Ice Cream с кодом ошибки 500 с 19:00 UTC 4 июля до 20:13 UTC.

Первопричина:
PR № 20034 создал регрессию (ранее исправленную ошибку в проблеме № 9877) в классе Ice Cream API Adapter.

Продолжительность: 73 минуты

Решение: Пиюш вернул изменения из № 20034 и вручную собрал и отправил новый образ для сервиса Ice Cream.

Процент затронутых сервисов: 93% запросов к сервису Ice Cream завершились с ошибкой.

Обнаружение: дежурный инженер (Пиюш) получил оповещение об обнаружении нарушения SLO.[2]

Ниши
Я немного запуталась. Если мы уже исправляли эту ошибку, почему она возникла снова? Разве у нас не было тестов?

Пиюш
У нас есть тесты для этого случая, и похоже, что они провалились на № 20034.

Ниши
Что? Почему же мы все-таки объединили № 20034?

Пит
Этот тест постоянно проваливался, так что мы не поняли, что в этот раз он действительно обнаружил ошибку. :(

[1] Национальный праздник, День независимости США. — *Примеч. ред.*

[2] SLO (Service level objectives) — набор целевых, «желаемых» значений SLI, выход за пределы которых может привести к нарушению SLA конкретного сервиса или компонента. SLI (Service level indicators) — набор ключевых метрик, по которым можно определить жизненный статус сервиса, его производительность и удовлетворенность конечных пользователей работой сервиса. SLA (Service level agreement) — так называемое соглашение об уровне доступности сервиса. — *Примеч. пер.*

Сигнал и шум

У компании Ice Cream for All проблема с нестабильными тестами. Они так часто проваливаются, что инженеры просто игнорируют сообщения об ошибках. И это привело к реальным проблемам: игнорирование нестабильного теста стоило компании прибыли в самый напряженный день в году!

Что делать с нестабильными тестами? Прежде всего понять суть проблемы. *Что означает «нестабильный тест»?*

Термин «*нестабильный*» (или «*зашумленный*», noisy) происходит от понятия *соотношение сигнал/шум* (signal-to-noise ratio), которое описывает отношение полезной информации (*сигнала*) к помехам, которые делают ее менее четкой (*шуму*).

Волнистая линия — это сигнал, а беспорядочная неровная линия — это шум, мешающий его воспринимать.

Что является сигналом для тестов? Какую информацию мы ищем? Это интересный вопрос, потому что можно подумать, что сигнал — это когда тест прошел. Но может быть и наоборот, сигналом являются случаи провала тестов.

Ответ: и то и другое! *Сигнал — это информация, а шум — это все, что отвлекает нас от нее.*

Когда тесты проходят, вы получаете информацию: вы знаете, что система ведет себя так, как вы ожидаете (как определено тестами). Когда тесты не проходят, вы тоже получаете информацию. И даже еще сложнее. Из следующей таблицы видно, что и провалы, и успехи могут быть сигналами, а могут быть и шумом.

Тесты	Успех	Провал
Сигнал	Проходит и должен проходить (то есть ловит ошибки, которые должен был поймать)	Сбои сообщают новую информацию
Шум	Проходит, но не должен (то есть тест ошибочный)	Сбои не сообщают никакой новой информации

«Нестабильный успех»

Эта информация может полностью изменить ваши представления, особенно если вы привыкли считать, что если тесты проходят, то это сигнал, а если проваливаются — то шум. Может, это и так, но как вы только что увидели, все немного сложнее:

Сигнал — это информация, а шум — это все, что отвлекает нас от нее.

- Успех — это сигнал, если только он не скрывает информацию.
- Провал — это сигнал, если он сообщает новую информацию, и шум, если не сообщает.

Когда успешный тест скрывает информацию? Например, в случае, когда тест проходит, но на самом деле не должен проходить, это так называемый *нестабильный успех* (noisy success). Например, в классе `Orders` метод, недавно добавленный в код Ice Cream For All, должен был возвращать самый последний заказ, и для него был добавлен следующий тест:

```
def test_get_most_recent(self):
  orders = Orders()
  orders.add(datetime.date(2020, 9, 4), "swirl cone")
  orders.add(datetime.date(2020, 9, 7), "cherry glazed")
  orders.add(datetime.date(2020, 9, 10), "rainbow sprinkle")

  most_recent = orders.get_most_recent()
  self.assertEqual(most_recent, "rainbow sprinkle")
```

Тест проходит, но оказывается, что метод `get_most_recent` просто возвращает последний заказ в базовом словаре:

```
class Orders:
  def __init__(self):
    self.orders = collections.defaultdict(list)

  def add(self, date, order):
    self.orders[date].append(order)

  def get_most_recent(self):
    most_recent_key = list(self.orders)[-1]
    return self.orders[most_recent_key][0]
```

В этом методе несколько ошибок, в том числе не учитывается случай, когда заказы не добавляются, но что еще более важно, когда заказы добавляются не по порядку.

Метод `get_most_recent` вообще не обращает внимания, когда были сделаны заказы. Он просто предполагает, что последний ключ в словаре соответствует самому последнему заказу. И так как тест добавляет самый свежий заказ последним (а начиная с Python 3.6 словарь упорядочен по порядку внесения элементов), то тест проходит.

Но поскольку базовая функциональность нарушена, тест вообще не должен проходить. Как я уже говорила, это то, что мы называем *нестабильным успехом*: прохождение этого теста скрывает информацию о том, что базовая функциональность не работает должным образом.

Как провалы тестов становятся шумом

Вы только что видели, как успешный тест может стать шумом. А как насчет провалов? Всегда ли неудачные тесты — это шум? Или всегда сигнал? Ни то ни другое! Ответ в том, что провалы являются сигналом, когда они дают новую информацию, и шумом, когда не дают. Вспомните:

Сигнал — это информация, а шум — это все, что отвлекает нас от нее.

- Успешные тесты — это сигналы, если только они не маскируют информацию.
- Сбои — это сигналы, если они сообщают новую информацию, и шум, если не сообщают.

Когда тест проваливается с самого начала, мы получаем новую информацию о том, что между ожидаемым и фактическим поведением теста существует несоответствие. Это и есть сигнал.

Тот же самый сигнал может стать шумом, если проигнорировать сбой. В следующий раз, когда произойдет этот же сбой, мы получим информацию, которую уже знаем: мы уже знаем, что раньше тест не прошел, поэтому новый сбой не сообщает новой информации. *Игнорируя неудачный тест, мы превратили сбой в шум.*

Это особенно часто случается, если трудно определить причину сбоя. Если сбой происходит не всегда (скажем, тест проходит успешно при запуске в рамках автоматизации CI, но не проходит локально), то вероятность того, что его проигнорируют, гораздо выше, что создает шум.

Ваш ход: определите, сигнал или шум

Посмотрите на следующие ситуации, происходящие в компании Ice Cream for All, и определите, «шум» это или «сигнал»:

1. Работая над новой функцией выбора любимых вкусов мороженого, Пит создает PR для своих изменений. Один из тестов пользовательского интерфейса не прошел, потому что Пит случайно переместил кнопку «Заказать» с ожидаемого места на другое.
2. Выполняя интеграционные тесты на своей машине, Пиюш видит, что тест не проходит: `TestOrderCancelledWhenPaymentRejected`. Он изучает результат теста, сам тест и тестируемый код и не понимает, почему тест не проходит. Когда он повторно запускает тест, все работает.
3. Хотя тест `TestOrderCancelledWhenPaymentRejected` у Пиюша однажды не прошел, он не может воспроизвести проблему и поэтому сливает свои изменения с общим кодом. Позже он вносит еще одно изменение и видит, что его PR не проходит тот же тест. Он повторно запускает его, и тест проходит, поэтому он снова игнорирует провал и сливает изменения.
4. Ниши проводит рефакторинг кода отображения истории заказов. При этом она замечает, что логика в одном из тестов неверна: тест `TestPaginationLastPage` ожидает, что сгенерированная страница будет содержать три элемента, но на самом деле она должна включать только два. Логика разбивки на страницы содержит баг.

Ответы

1. Сигнал. Провал теста пользовательского интерфейса сообщил Питу новую информацию: он переместил кнопку «Заказать».
2. Сигнал. Пиюш не понимал, что послужило причиной провала этого теста, но что-то вызвало провал, и это позволило получить новую информацию.
3. Шум. Из своего опыта с этим тестом Пиюш подозревал, что с ним может быть что-то не так. Увидев, что тест снова провалился, он решил, что информация, которую он получил раньше, была достоверной, но допустив этот провал и слияние, он создал шум.
4. Шум. Тест, который обнаружила Ниши, должен был всегда проваливаться; то, что он прошел, скрыло информацию.

От шума к сигналу

Системы оповещения полезны, только если на них обращают внимание. Если они слишком нестабильные, люди перестают их замечать и могут пропустить сигнал.

Пример — автосигнализация: если вы живете в районе, где припарковано много машин, и слышите, как срабатывает сигнализация, броситесь ли вы к окну с телефоном наготове на экстренный случай? Наверное, это зависит от того, как часто подобное происходит; если вы никогда не слышали сигнал, то скорее всего, вы отреагируете. Но если вы слышите его каждые несколько дней, скорее всего, вы подумаете: «А, кто-то в кого-то врезался. Надеюсь, сигналка скоро замолчит».

Что вы сделаете, если живете или работаете в многоквартирном доме и у вас сработала пожарная сигнализация? Вы, скорее всего, воспримете это всерьез и нехотя выйдете из здания. А если это повторится на следующий день? Вы, вероятно, все равно покинете здание, потому что эти сигналы тревоги *громкие*, но начнете сомневаться, что это действительно чрезвычайная ситуация, а еще через день точно решите, что это ложная тревога.

Чем дольше поступает нестабильный сигнал, тем легче его игнорировать и тем менее он эффективен.

Цель — зеленый режим

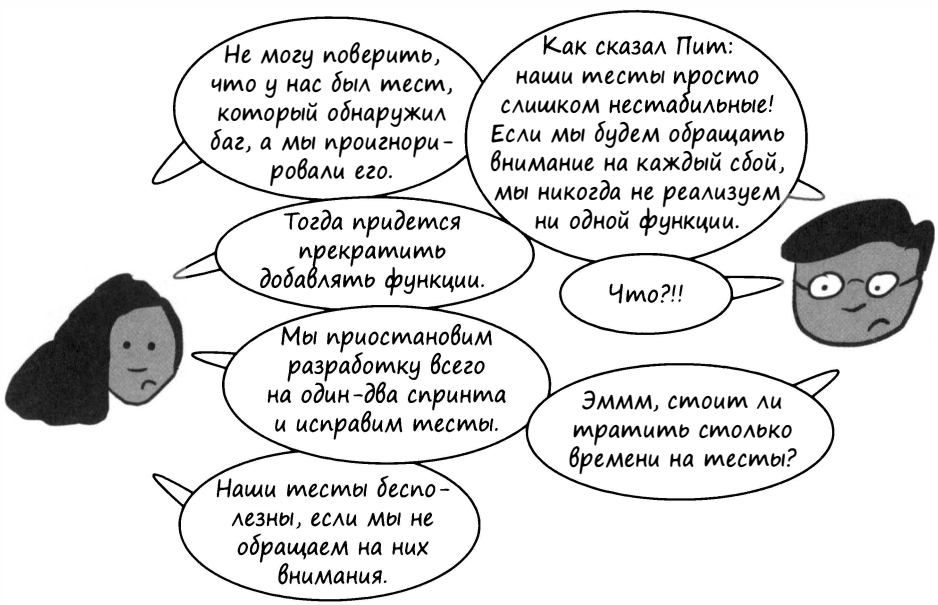
Словарик

Успешные тесты часто визуализируются зеленым цветом, а неудачные — красным. *Зеленый режим* означает, что все тесты пройдены!

Чем дольше тесты остаются нестабильными, тем легче их игнорировать (даже если они сообщают информацию) и тем меньше их эффективность. Если ничего с ними не делать, они перестанут приносить пользу. Люди перестают воспринимать ложные провалы и просто их игнорируют.

Именно это случилось в компании Ice Cream for All: ее разработчики настолько привыкли игнорировать тесты, что пропустили несколько серьезных проблем. Тесты выявили эти проблемы, но компания все-таки потеряла деньги.

Что же делать? Решение — как можно быстрее войти в *зеленый режим*, то есть в состояние, когда все тесты постоянно проходят, так что любое изменение этого состояния (сбой) станет реальным сигналом, который необходимо будет изучить.

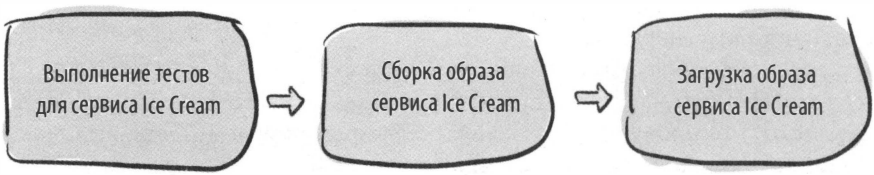


Ниши совершенно права: мы создаем и поддерживаем тесты не ради самих тестов; мы делаем это, потому что считаем, что они приносят пользу, и бóльшую часть этой пользы составляют сигналы, которые они нам сообщают. Поэтому она принимает трудное решение: приостановить разработку новых функций, пока все тесты не будут исправлены.

Тесты приносят и другую пользу. Например, юнит-тесты, или модульные тесты, помогают улучшить качество кода. Эта тема выходит за рамки данной книги. Чтобы узнать больше, поищите книгу о модульном тестировании.

Опять сбой!

Команда делает то, что просит Ниши: приостанавливает разработку функций на две недели и в это время занимается только исправлением тестов. Через три недели тесты в пайплайне стабильно проходят. Они вошли в зеленый режим!



Команда уверена, что можно добавлять новые функции, и на третьей неделе возвращается к обычной работе. В конце недели должен состояться выпуск очередного релиза, и планируется устроить небольшую вечеринку по этому поводу. Но в 3 часа ночи Ниши разбудил сигнал о том, что произошел очередной сбой.

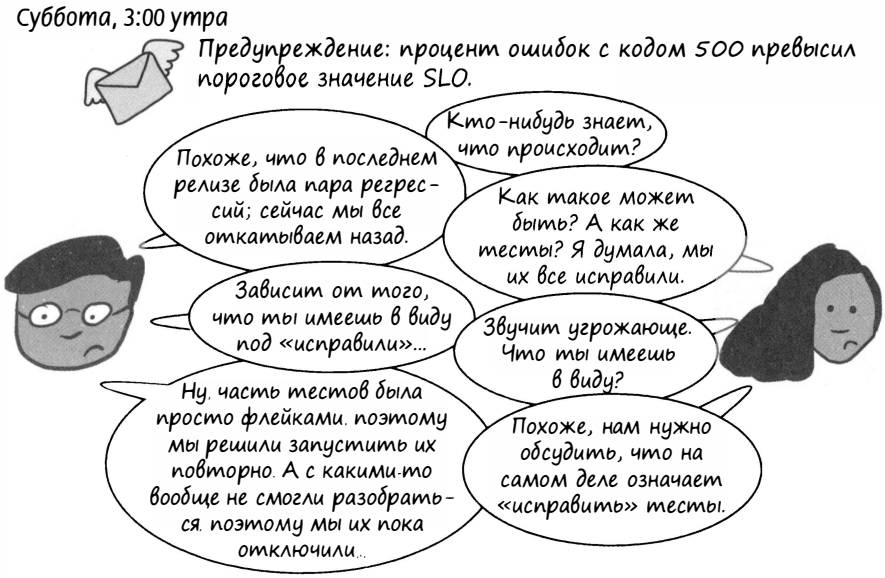
Повод для размышления

Правильно ли поступила Ниши? Она решила приостановить разработку функций на две недели, чтобы исправить тесты, а в итоге снова произошел сбой. Оглядываясь на это решение, легко сделать вывод, что (несомненно, затратное) замораживание разработки функций того не стоило и принесло больше вреда, чем пользы.

Ниши пришлось принять решение, с которым часто сталкиваются многие из нас: сохранить статус-кво (для Ice Cream for All это означает нестабильные тесты, которые неожиданно вызывают сбои) или принять меры. «Принять меры» означает попробовать что-то сделать и что-то изменить. Каждый раз, когда вы меняете привычный порядок действий, вы рискуете: изменения могут пойти как на пользу, так и во вред. Нередко происходит и то и другое! И зачастую после изменений наступает период адаптации, в течение которого все становится только хуже, хотя в конечном итоге ситуация изменится к лучшему.

Как бы вы поступили на месте Ниши? Считаете ли вы, что она права? Что бы вы сделали по-другому на ее месте, если бы вообще стали что-то делать?

Успешно прошедшие тесты могут оказаться нестабильными



Команда была довольна своими тестами, потому что они проходили успешно. К сожалению, шум никуда не делся, его только изменили. Теперь успешные тесты стали шумом.

Изменение шума на сигнал — правильное решение, а переход от часто проваливающихся тестов в зеленый режим — хорошо для начала, поскольку это помогает бороться с ослаблением внимания.

Но простого перехода к зеленому режиму недостаточно: тесты, которые проходят, все еще могут быть нестабильными и скрывать серьезные проблемы.

Словарик

Более подробно объясню чуть позже, но вкратце: *флейк* (flake) — это тест, который иногда проходит, а иногда нет.

Команда решила проблему ослабления внимания, но не смогла исправить тесты.

ВАЖНО

Ниши сделала правильный выбор, но одного зеленого режима недостаточно. Необходимо, чтобы тесты оставались в устойчивом состоянии («пройден») и любое изменение в этом состоянии («сбой») являлось реальным сигналом, который необходимо проанализировать. Работайте с нестабильными тестами так:

1. Как можно быстрее войдите в зеленый режим.
2. Действительно *исправляйте* каждый проваленный тест; простое их игнорирование только добавляет шума.

Исправление ошибок в тестах

Возможно, вы удивитесь, узнав, что понять, исправили вы тест или нет, довольно сложно. Здесь мы возвращаемся к вопросу о том, что в тестах является сигналом, а что — шумом.

Люди часто думают, что исправить тест означает сделать проваленный тест успешно пройденным. Но этого недостаточно!

Технически исправление теста означает, что тест из состояния шума перешел в состояние сигнала. То есть существуют тесты, которые проходят, но которые нужно исправить. Подробнее об этом чуть позже; сейчас поговорим об исправлении тестов, которые не проходят.

Каждый раз, когда тест не проходит, это означает, что произошло одно из двух событий (или оба сразу):

1. Тест написан неправильно (система не должна вести себя так, как ожидает тест).
2. В системе есть ошибка (с тестом все в порядке, а система ведет себя неправильно).

Интересно, что обычно мы пишем тесты, имея в виду вторую ситуацию, но когда тесты проваливаются (особенно если не сразу понятно почему), мы склонны считать, что имеет место первая ситуация (проблема в самих тестах).

Именно это чаще всего происходит, когда разработчики говорят, что их тесты нестабильны: тесты проваливаются, и они не могут понять почему, поэтому делают поспешный вывод, что с тестами что-то не так.

Но у обеих ситуаций есть нечто общее:

Если тест не проходит, значит, существует несоответствие между тем, какое поведение системы ожидает тест, и тем, как на самом деле ведет себя система.

Независимо от того, что необходимо исправить — тест или систему, это несоответствие должно быть проанализировано. Именно в этот момент жизненного цикла теста вероятность появления шума наибольшая. Провал теста сообщил информацию, в частности, о том, что между тестами и системой существует несоответствие. Если ее проигнорировать, то каждый новый сбой не даст ничего нового. Вместо этого он будет повторять то, что уже известно: несоответствие существует. *Именно так провалы тестов становятся шумом.*

Другой способ внести шум — ошибочно принять ситуацию 2 за ситуацию 1. Часто проще изменить тест, чем выяснить, почему система ведет себя так, как ведет. Если вы изменяете тест, не понимая толком поведения системы, вы создаете нестабильный успешный тест. Каждый раз, когда этот тест проходит, он скрывает информацию, а именно тот факт, что между тестом и системой существует несоответствие, которое так и не было до конца изучено.

Рассматривайте каждый провал теста как баг и исследуйте его полностью.

Виды провала тестов: flaky-тесты

Если усложнить историю о сигнале и шуме в тестах, то мы придем к самому печально известному виду сбоев в тестировании: flaky-тестам. Тесты могут проваливаться двумя способами:

- *Последовательно* (консистентно) — каждый раз, когда тест запускается, он завершается ошибкой.
- *Непоследовательно* (неконсистентно) — иногда тест проходит успешно, а иногда проваливается, и неясно, при каких условиях.

Тесты, которые терпят неудачу время от времени, обычно называют *флейками*, или *flaky-тестами*[1] (то есть нестабильными, ненадежными). Когда такие тесты проваливаются, это называется «*ненадежностью*»: точно так же как нельзя полагаться на ненадежного друга в общих планах, нельзя рассчитывать, что эти тесты будут стабильно проходить или проваливаться.

С консистентными тестами гораздо проще работать, чем с flaky-тестами, и вероятность их исправления (хочется надеяться, таким образом, чтобы снизить уровень шума) гораздо выше. Чаще всего наборы тестов оказываются нестабильными именно из-за того, что в них присутствуют flaky-тесты. И может быть, из-за этого, а может, просто потому, что так проще, flaky-тестам не уделяют столько же внимания, как последовательным сбоям:

- Flaky-тесты делают наборы тестов нестабильными.
- Flaky-тесты, скорее всего, будут игнорироваться и рассматриваться как несущественные сбои.

Это довольно иронично, потому что мы видели, что чем нестабильнее набор тестов, тем от него меньше пользы. А какие тесты могут сделать весь набор нестабильным? Flaky-тесты, которые мы, скорее всего, проигнорируем.

Обрабатывайте flaky-тесты как любой другой вид сбоев тестов — как баги.

Как и в любом другом случае провала теста, flaky-тесты представляют собой несоответствие между реальным и ожидаемым поведением системы. Разница лишь в том, что в этом несоответствии есть неопределенность.

[1] Flake — осколок, клочок. Flaky — буквально «распадающийся на хлопья». — *Примеч. пер.*

Реагирование на сбои

Что пошло не так в подходе компании Ice Cream For All к процессу тестирования? Команда изначально мыслила в верном направлении:

> **Если тесты проваливаются, необходимо остановить процесс: не продолжать, пока тесты не будут исправлены.**

Если в коде есть тесты, которые не проходят, важно как можно быстрее перейти в зеленый режим, не выполнять слияния с главной веткой, пока эти сбои не будут исправлены. А если неудачные тесты возникли в самой ветке, не проводить слияние с этой веткой до тех пор, пока не будут исправлены ошибки. Но вопрос в том, *как устранять эти ошибки*. Вариантов несколько:

- *Действительно все исправить* — в конечном счете необходимо понять, почему тест провалился, и либо исправить выявленную ошибку, либо обновить некорректный тест.
- *Удалить тест* — это случается редко, но в ходе анализа может выясниться, что тест бесполезен и что его провал не повод принимать меры. В таком случае нет смысла оставлять тест и продолжать его поддерживать.
- *Отключить тест* — это крайняя мера, и если применять ее, то только временно. Отключение теста означает, что сигнал будет скрыт. Любые отключенные тесты необходимо проверить как можно быстрее и либо исправить (см. вариант 1), либо удалить.
- *Повторить тест* — это еще одна крайняя мера, которая также скрывает сигнал. Это распространенный способ борьбы с flaky-тестами. В основе этого способа лежит стремление добиться того, чтобы все тесты проходили, но это в корне неверно: тесты должны предоставлять информацию. Если тест иногда не проходит и вы скрываете это, повторяя попытку, тем самым вы скрываете информацию и повышаете нестабильность. Повторные попытки иногда целесообразны, но редко на уровне самого теста.

Приемлемым является только вариант 1 и, в редких случаях, вариант 2. Варианты 3 и 4 — это меры, которые следует принимать только на некоторое время, если вообще стоит, потому что они добавляют к сигналу шум, скрывая сбои.

Код или тест: что менять, исправляя тесты?

Компания Ice Cream for All откатила последний релиз и снова приостановила разработку функций, пока команда изучает тесты, которые они «исправляли» ранее.

Просматривая часть исправлений, для которых было проведено слияние, Ниши замечает тревожную закономерность: многие из этих «исправлений» изменяют только тесты и лишь некоторые — собственно тестируемый код. Ниши понимает, что это антипаттерн. Например, этот тест был ненадежным, поэтому он был обновлен и время ожидания условий его успешного выполнения увеличено:

```
def test_submit_order(self):
  orders = _generate_orders(5)
  submit_orders(orders)
  events = get_events(PROCESSED)
  self.assertEqual(len(events), 5)
```

```
def test_submit_order(self):
  orders = _generate_orders(5)
  submit_orders(orders)

  # wait for all the orders to be processed
  done = lambda: len(get_events(PROCESSED)) == 5
  wait_for_condition(TIMEOUT_SECONDS, done)

  events = get_events(PROCESSED)
  self.assertEqual(len(events), 5
```

Изначально тест был написан исходя из предположения, что заказ будет считаться подтвержденным сразу после его отправки. И код, вызывающий **`submit_orders`**, тоже был создан с учетом этого предположения. Но этот тест оказался ненадежным из-за возникновения состояния гонки данных в **`submit_orders`**!

Вместо того чтобы исправить эту проблему в функции `submit_orders`, кто-то обновил тест, что скрыло баг и добавило «нестабильный» успешный тест в набор тестов.

На самом деле баг просто скрыли.

Когда вы работаете с тестом, который дает сбой, то прежде чем вносить изменения, выясните, нет ли проблемы в самом тестируемом коде. То есть если код ведет себя подобным образом, когда используется вне тестов, должен ли он так себя вести? Если да, то имеет смысл исправить тест. Но если нет, то исправление не должно касаться теста: его необходимо внести в код.

Это означает, что нужно мысленно переключиться с «давайте исправим тест» (чтобы он успешно проходил) на «давайте поймем, чем фактическое поведение кода отличается от ожидаемого, и исправим то, что нужно».

Рассматривайте каждый провал теста как баг и исследуйте его со всех сторон.

Ниши просит разработчика, обновившего тест, провести дальнейшее исследование. Найдя источник состояния гонки, они смогут исправить основную ошибку, и тест вообще не нужно будет менять.

Опасности повторного тестирования

Повторный запуск всего теста обычно не слишком хорошая идея, потому что все, что вызвало сбой, будет скрыто. Посмотрите на тест ниже, это один из тестов для интеграции сервиса Ice Cream с Mr. Freezie:

```
# Тест не должен проваливаться только потому,
# что сетевое подключение MrFreezie ненадежно.
@retry(retries=3)
def test_process_order(self):
  order = _generate_mr_freezie_order()
  mrf = MrFreezie()
  mrf.connect()
  mrf.process_order(order)
  _assert_order_updated(order)
```

Во время приостановки разработки Пит решил, что этот тест следует повторить. Его доводы были вполне логичными: известно, что сетевое подключение к серверам Mr. Freezie ненадежно, поэтому тест иногда дает сбои, так как не может успешно установить соединение, и сразу же проходит при повторной попытке.

Но проблема в том, что Пит повторяет весь тест. Поэтому если тест не пройдет по другой причине, он все равно будет повторен. Именно это и произошло. Оказалось, что в способе передачи заказов на Mr. Freezie была ошибка, из-за которой общая сумма заказа иногда оказывалась неверной. Когда это происходило в реальной системе, пользователям выставлялись неверные счета, что приводило к возникновению ошибки с кодом 500 и сбою в работе.

Что Пит должен был сделать на самом деле? Помните, что сбои в тестировании представляют собой несоответствие:

> **Если тест не проходит, значит, существует несоответствие между тем, какое поведение системы ожидает тест, и тем, как на самом деле ведет себя система.**

Пит должен задать себе вопрос, который необходимо задавать каждый раз, изучая провал теста:

> **Где на самом деле заключено желаемое поведение — в тесте или в системе?**

Разумным шагом в стратегии Пита было бы изменить логику повторных тестов, то есть чтобы они касались только сетевого подключения:

```
def test_process_order_better(self):
  order = _generate_mr_freezie_order()
  mrf = MrFreezie()

  # Тест не должен проваливаться только потому,
  # что сетевое подключение MrFreezie ненадежно.
  def connect():
    mrf.connect()
  retry_network_errors(connect, retries=3)

  mrf.process_order(order)
  _assert_order_updated(order)
```

Пересмотр повторного тестирования

Пит доработал свое решение и стал перезапускать только ту часть теста, которая, по его мнению, иногда могла не срабатывать. В процессе анализа кода Пиюш идет еще дальше:

Пиюш

Спасибо за исправление, Пит! Так намного лучше :)
Просто интересно: код, который фактически вызывает `MrFreezie.Connect()`, мы тоже запускаем повторно? Я думаю, что если подключение такое ненадежное, пользователи столкнутся с той же проблемой.

Пит

Хороший вопрос. Ты прав: если не удается вызвать функцию `Connect()` для любой из интеграций, мы просто сразу прекращаем работу. Я обновлю код сервиса Ice Cream, чтобы он был более устойчив к сетевым ошибкам.

На самом деле повторный запуск теста скрывал два бага: помимо ошибки в способе передачи заказов на Mr. Freezie, существовал более крупный баг, который заключался в том, что код сервисов Ice Cream не был устойчив к сбоям в сети (вы же не хотите, чтобы ваш заказ на мороженое не был выполнен только из-за того, что в сети возникли временные неполадки, правда?).

Компании Ice Cream for All повезло, что разработчики так быстро обнаружили проблемы, возникшие из-за повторного запуска тестов. Если бы сбоя не произошло, их, возможно, никогда бы и не заметили и для исправления ненадежных тестов продолжали использовать перезапуск. Только представьте возможный масштаб проблемы: сколько багов остались бы скрытыми через несколько лет применения этой стратегии?

Перезапуск flaky-тестов, чтобы они успешно проходили, вносит шум: шум тестов, которые прошли, но не должны были.

Специфика программных проектов такова, что в них добавляется все больше сложности, и это ведет к тому, что небольшие возникающие недочеты будут разрастаться по мере развития проекта. Если немного сбавить темп и переосмыслить такие промежуточные меры, как перезапуск тестов, это окупится в долгосрочной перспективе!

Ваш ход: исправление flaky-теста

Пиюш пытается разобраться с еще одним нестабильным тестом в сервисе Ice Cream. Этот тест не проходит, но случается это реже чем раз в неделю, хотя тесты выполняются не менее сотни раз в день, и эту ситуацию очень трудно воспроизвести локально:

```
def test_add_to_cart(self):
   cart = _generate_cart()
   items = _genterate_items(5)
   for item in items:
     cart.add_item(item)
   self.assertEqual(len(cart.get_items()), 5)
```

Иногда это не срабатывает.

Корзина работает с базой данных. Каждый раз, когда товар добавляется в корзину, база данных обновляется, а когда товары считываются из корзины, они считываются из базы данных.

1. Предположим, что когда тесты не проходят, количество товаров в корзине равно 4, а не 5. Как вы думаете, что может пойти не так?
2. А если проблема в том, что количество товаров 6, а не 5, что тогда может пойти не так?
3. Если Пиюш решит эту проблему, повторно запустив тест, какие ошибки он рискует скрыть?
4. Допустим, Пиюш замечает эту проблему, когда пытается исправить критически важный для выпуска продукта вопрос. Что ему сделать, чтобы убедиться, что он не добавит еще больше шума, при этом не заблокировав критически важное исправление?

Ответы

1. Если количество прочитанных элементов меньше, чем записанное, возможно, где-то возникает состояние гонки. Необходимо ввести синхронизацию, чтобы прочитанные данные действительно соответствовали записанным.
2. Если число больше, возможно, существует фундаментальный дефект в способе записи элементов в базу данных.
3. Любой из описанных выше сценариев указывает на недостатки в логике работы корзины, которые могут привести к потере заказов и неправильному выставлению счетов на оплату.
4. В этом сценарии целесообразно добавить временный перезапуск для разблокировки этой работы, если проблема с `test_add_to_cart` впоследствии будет расцениваться как баг, а логика перезапусков будет быстро удалена.

Зачем же перезапускать тесты?

Учитывая то, что мы только что узнали, вы можете удивиться, зачем вообще повторять проваленные тесты. Если это так плохо, почему так много разработчиков делают это и почему так много тестовых фреймворков это поддерживают? На то есть несколько причин:

- Для использования той или иной логики повторения тестов часто есть веские основания: например, Пит был прав, когда планировал повторить попытку сетевого подключения, если оно не сработало. Но вместо того чтобы проверить, что логика перезапуска теста размещена в нужном месте, ему проще было повторить весь тест.
- Если вы правильно настроили пайплайны, проваленный тест блокирует и замедляет всю работу. Логично стараться возобновить ее как можно быстрее и с минимальными усилиями, и в таких ситуациях перезапуск тестов может быть уместен, но при условии, что это только временная мера.
- Приятно что-то починить, а еще приятнее — сделать это с помощью технологии; перезапуск тестов позволяет получить немедленное удовлетворение.
- И основное: люди часто думают, что главное — чтобы тесты проходили, но это заблуждение. Нельзя запускать тесты только ради того, чтобы они проходили. Тесты важны потому, что они сообщают информацию (сигнал). Когда мы скрываем сбои, не решая проблему, мы снижаем значимость набора тестов, добавляя в него шум.

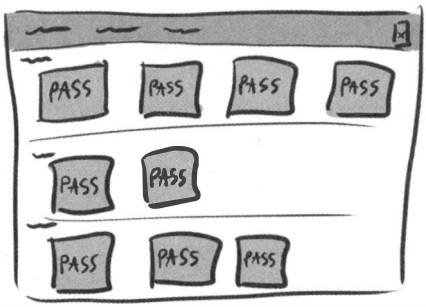
> ***Нет ничего более постоянного, чем временное***
>
> Будьте осторожны всякий раз, когда вносите временное исправление. Такие решения отодвигают решение основной проблемы, и вы не успеете оглянуться, как пройдет два года, и временное исправление станет постоянным.

Поэтому если у вас возникнет соблазн повторить тест, остановитесь и подумайте, можно ли выяснить, что на самом деле вызвало проблему. Перезапускать тесты целесообразно в следующих случаях:

- перезапуск применяется только к недетерминированным и неконтролируемым элементам (например, к интеграции с другими работающими системами);
- повторяемая операция изолирована (например, в случае Пита достаточно повторить только вызов `Connect()` вместо всего теста).

Добившись «зеленого» статуса, поддерживайте его

Казалось, что бы ни делала компания Ice Cream for All, что-то все время шло не так. Однако разработчики двигались в верном направлении; они получили несколько полезных уроков, которые им нужно усвоить, и надеюсь, мы тоже сможем извлечь пользу из их ошибок!



В любом проекте в первую очередь важно привести набор тестов в «зеленый» режим и поддерживать его в таком состоянии. Если у вас много тестов, которые проваливаются (независимо от того, постоянно ли они не проходят или это флейки), имеет смысл принять радикальные меры, чтобы они стали сообщать полезный сигнал:

- Приостановка разработки для исправления наборов тестов, несомненно, со временем окупится. Однако если добро на это вам не дают (в конце концов, это дорогое удовольствие), не отчаивайтесь, просто прийти в «зеленый» режим будет сложнее.
- Отключение и перезапуск проблемных тестов, несмотря на то что в долгосрочной перспективе такой подход неэффективен, может помочь войти в «зеленый» режим (то есть получать информативные сигналы), если правильно расставить приоритеты при последующем исследовании этих тестов!

Помните о балансе: как бы вы ни старались и как бы хорошо ни поддерживали свои тесты, баги всегда будут возникать. Вопрос в том, во сколько они обходятся.

Если вы разрабатываете критически важную технологию в области здравоохранения, стоимость этих ошибок огромна, и целесообразно потратить время на тщательное устранение всех возможных багов. Но если вы работаете над интернет-магазином мороженого, то, безусловно, имеете бóльшую свободу. (Ни в коем случае не преуменьшаю ценность мороженого, оно очень вкусное!)

Достигнув «зеленого» режима, постарайтесь в нем остаться. Обрабатывайте каждый сбой как баг, но и не относитесь к ним серьезнее, чем это необходимо.

Ладно, бросьте: многие тесты ненадежны, но не все они ведут к сбоям, не так ли?

Возможно, злосчастной компании Ice Cream for All просто исключительно не повезло, что у нее случилось несколько сбоев, которых можно было избежать, если бы обнаружившие их тесты не были проигнорированы. Однако исключать, что подобное произойдет снова, нельзя. Обычно такие ситуации совсем не очевидны, и главное, никогда не знаешь наверняка, когда они возникнут. Еще более серьезная проблема заключается в том, что такое небрежное отношение к тестам со временем нивелирует их значимость. Представьте себе пожарную сигнализацию, которая иногда срабатывает при пожаре, а иногда вхолостую (или, что еще хуже, не срабатывает при пожаре!), и сигнализацию, которая *всегда* означает пожар. Какая из них представляет бóльшую ценность?

Билд-инженер и разработчик

В зависимости от того, какая у вас роль в команде, вы, возможно, читаете эту главу с ужасом, думая: «Но ведь я не могу изменять тесты!». Довольно часто роли в команде распределяются таким образом, что за состояние тестов отвечают совсем не те люди, которые их пишут или разрабатывают функции. Например, это может быть *билд-инженер*, *инженер по повышению производительности* или аналогичные роли: эти специалисты взаимодействуют с командой разработчиков функций и оказывают ей поддержку.

Если вы именно такой специалист, у вас может возникнуть соблазн положиться на решения, которые не требуют внимания или усилий разработчиков, занимающихся созданием функций. Далеко не в последнюю очередь поэтому так часто используют автоматизацию (например, повторные тесты) вместо того, чтобы устранять ошибки напрямую в самих тестах.

Но если эволюция сферы разработки и научила нас чему-то, так это тому, что слишком жесткое разграничение ответственности между исполнителями является антипаттерном. Просто взгляните на DevOps — это попытка разрушить барьеры между разработчиками и командой эксплуатации. Точно так же если мы попытаемся провести жесткую грань между процессами сборки и разработки функций, мы с удивлением поймем, что нам это не удалось.

Правда непрерывной доставки в целом и тестирования в частности заключается в том, что без усилий со стороны самих разработчиков функций они не будут эффективными. Попытка исключить фактор разработчиков со временем приведет к ухудшению качества и эффективности набора тестов.

Итак, если вы билд-инженер, что вам делать? У вас есть три варианта:

- Применять возможности автоматизации, такие как повторные тесты, и смириться с тем, что это со временем приведет к ухудшению качества тестов.
- Взять на себя обязанности разработчика функций и научиться вносить необходимые исправления (в тесты *и* в код, который вы тестируете).
- Заручиться поддержкой со стороны разработчиков функций и тесно сотрудничать с ними для устранения любых сбоев в тестировании (например, открывая баги для отслеживания неудачных тестов в системе и доверяя разработчикам их оперативное устранение).

Заключение

Тестирование — это сердце непрерывной доставки. Без тестирования вы не будете знать, безопасны ли изменения, которые вы собираетесь непрерывно интегрировать. Но печальная правда в том, что если не поддерживать наборы тестов, их эффективность с течением времени снижается. В частности, это часто происходит из-за непонимания того, что такое нестабильные тесты, — но это решаемо!

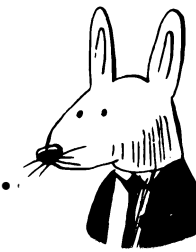
Итоги

- Тесты имеют решающее значение для непрерывной доставки.
- Как проваленные, так и прошедшие тесты могут добавлять шум; нестабильные тесты — это любые тесты, которые скрывают информацию, которую должен предоставлять набор тестов.
- Лучший способ восстановить эффективность нестабильного набора тестов — как можно быстрее войти в «зеленый» режим (в котором все тесты успешно проходят).
- Работайте с проваленными тестами как с багами и помните, что часто необходимо исправлять код, а не сам тест. В любом случае сбой представляет собой несоответствие между реальным поведением системы и поведением, ожидаемым в тесте, и этот сбой заслуживает тщательного анализа.
- Повторное выполнение всего теста требуется редко, и делать это следует с осторожностью.

Далее...

В следующей главе мы продолжим рассматривать проблемы, которые возникают в наборах тестов по мере их расширения, а именно их тенденцию замедляться, часто до такой степени, что это тормозит разработку функций.

Ускорение медленных наборов тестов | 6



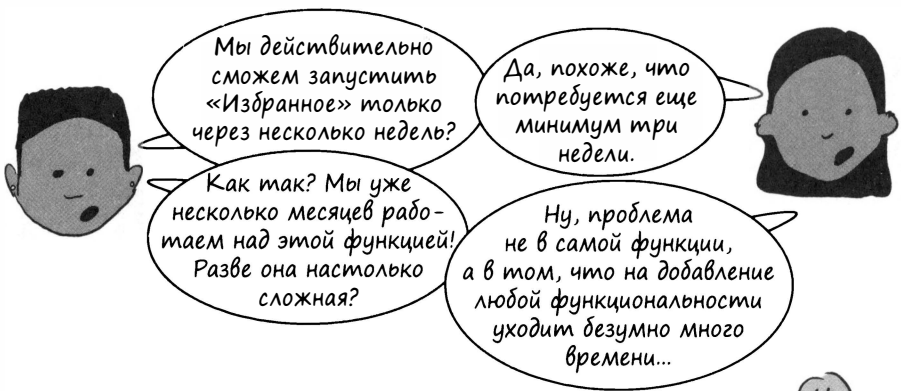
В этой главе

- ✓ Ускорение работы медленных тестовых наборов за счет того, что вначале запускаются более быстрые тесты
- ✓ Использование «пирамиды тестов», чтобы определить оптимальное соотношение между интеграционными, системными и юнит-тестами
- ✓ Измерение тестового покрытия кода для достижения и поддержания необходимого соотношения
- ✓ Как получать более быстрый сигнал от медленных тестов за счет параллельного выполнения и шардинга
- ✓ Когда параллельное выполнение и шардинг целесообразны и как их использовать

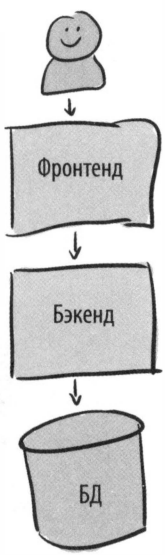
В предыдущей главе вы узнали, что делать с тестами, которые не предоставляют полезный сигнал, но как быть, если тесты банально тормозят? Неважно, насколько качественный сигнал дают тесты; если на его получение уходит слишком много времени, это замедляет всю разработку! Посмотрим, что можно сделать даже с самыми безнадежно медленными наборами.

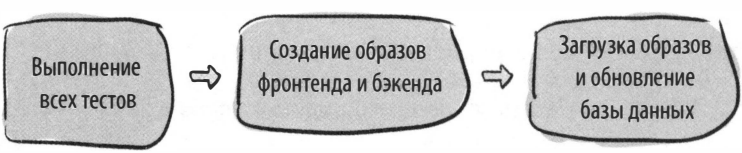
Веб-сайт Dog Picture

Помните сайт Cat Picture из главы 2? У его крупнейшего конкурента, сайта Dog Picture, трудности со скоростью разработки. Джаду, менеджера этого программного продукта, волнует то, что на разработку даже простейших пользовательских функций уходят месяцы.



Чтобы понять, почему разработка веб-сайта Dog Picture идет так медленно, кратко рассмотрим его архитектуру и пайплайн. Можно заметить, что архитектура Dog Picture немного проще, чем некоторые другие архитектуры, которые мы рассматривали: разработчики разделили фронтенд и бэкенд, но не пошли дальше и не переместили ни одно из хранилищ в облако.





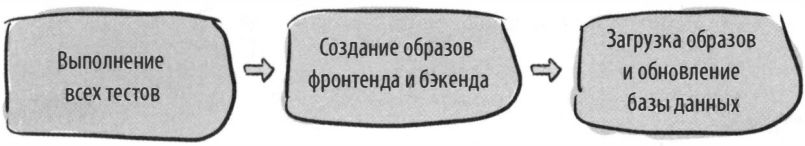
Почему же у них возникли проблемы при такой простой архитектуре?

Перенос в облако решит все проблемы?

Между сайтами Dog Picture и Cat Picture есть одна большая разница — последний использует облачное хранилище. Решает ли оно проблему? Нет, не в этом случае! Скорее наоборот, это усложнит процесс тестирования, поскольку меньше компонентов будет находиться под контролем разработчика. (Прочие преимущества могут перевесить эти недостатки, но это уже тема другой книги!)

Когда простое слишком просто

Пайплайн, который используется на сайте Dog Picture, кажется простым и логичным. На первый взгляд он такой же, как и пайплайны, которые мы рассматривали до сих пор. Но у него есть важное отличие.



Это *единственный* пайплайн, который используется для Dog Picture. Разработчики применяют его и для тестирования, и для сборки и загрузки обоих образов: фронтенда и бэкенда. Других пайплайнов у них нет. Еще в главе 2 мы познакомились с архитектурой и дизайном пайплайна крупнейшего конкурента Dog Picture — сайта Cat Picture.

Сайт Cat Picture использует отдельный пайплайн для каждого из своих сервисов:

Dog Picture вместо этого решил использовать один пайплайн для всей системы, что неплохо для начала, однако при этом компания не сможет развиваться. В частности, задача, запускающая тесты, запускает все тесты сразу. Dog Picture значительно отстает от своего ближайшего конкурента по сложности структуры пайплайна!

Новый разработчик пытается отправить код

Посмотрим, что происходит при попытке отправить код на сайт Dog Picture и как структура пайплайна, особенно в части тестирования, влияет на скорость разработки. Шридхар, новичок в команде Dog Picture, работает над новой функцией «Избранное», о которой спрашивала Джада. Фактически он уже написал код, который, по его мнению, необходим для этой функции, а также несколько тестов. Что дальше?

Вторник

14:00: Шридхар вносит изменения в код.

15:14: Другой разработчик вносит изменения.

15:30: Третий разработчик вносит еще больше изменений.

23:00: Система CD начинает ночной прогон пайплайна.

Среда

01:42: тесты не прошли.

Система CD отправляет электронное письмо Шридхару и двум другим разработчикам, которые внесли изменения, о том, что пайплайн остановлен.

16:02: Потратив весь день на попытки отладить систему, Шридхар и два других разработчика отменяют свои изменения, чтобы восстановить работу пайплайна. Шридхар еще раз попытается отладить возникшие проблемы и, возможно, завтра снова выложит изменения.

Проблемы сайта Dog Picture отличаются от тех, которые мы рассматривали в предыдущей главе: набор тестов всегда выдает «зеленый» результат, но тесты запускаются только раз в день вечером, а утром разработчикам приходится разбираться, кто что повредил. И как мы видели в главе 2, это действительно очень замедляет работу!

Тесты и непрерывная доставка

Самое время задать интересный вопрос: а работает ли вообще команда сайта Dog Picture с непрерывной доставкой? В какой-то мере ответ всегда положительный, поскольку компания придерживается некоторых ее принципов, включая автоматизацию развертывания и *непрерывное тестирование*, но вернемся еще раз к тому, что вы узнали в главе 1. Вы реализуете CD, когда:

- можете безопасно вносить изменения в свой продукт в любое время;
- доставка этого продукта максимально проста, буквально нажатием кнопки.

Подумаем о первом условии: можно ли вносить изменения на сайт Dog Picture безопасно и в любое время? Шридхар произвел слияние своих изменений за несколько часов до того, как ночью автоматизированная система заметила, что тесты не проходят. А если бы команда Dog Picture захотела выполнить развертывание во второй половине этого дня, было бы это безопасно?

Нет! Определенно нет! Потому что их тесты запускаются только ночью.

- Разработчикам всегда нужно ждать по крайней мере до следующего дня после внесения изменения, чтобы провести развертывание продукта.
- Их продукт находится в состоянии готовности к релизу только сразу после прохождения тестов, пока не внесены другие изменения (допустим, тесты проходят ночью, а кто-то вносит изменение в восемь утра: это немедленно возвращает продукт в состояние, когда неизвестно, можно ли безопасно выпустить релиз).

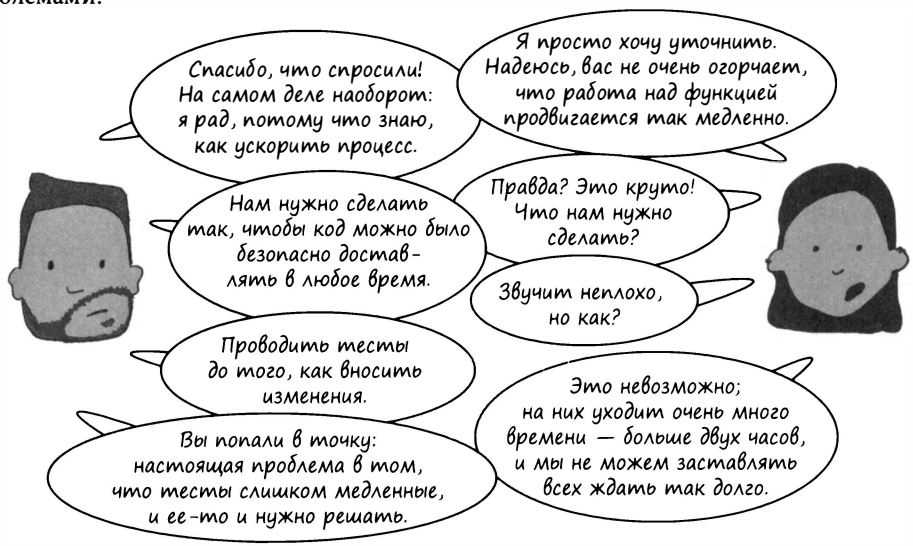
Итак, можно сказать, что сайт Dog Picture не соответствует первому условию непрерывной доставки.

Словарик

Непрерывное тестирование — это выполнение тестов как часть пайплайна CD. Это не столько отдельная методика, сколько признание того, что тесты должны выполняться постоянно. Просто иметь тесты недостаточно: можно их иметь, но никогда не запускать или автоматизировать, но запускать лишь время от времени.

Диагноз: слишком медленно

К счастью, Шридхар — опытный разработчик и уже сталкивался с подобными проблемами!



Его руководитель настроен скептически, но Шридхар уверен в себе, а Джада, менеджер продукта, в восторге от идеи попытаться ускорить разработку. Шридхар изучает среднее время выполнения тестов за последние несколько недель: 2 часа 35 минут. Он формулирует следующие цели:

- Запускать тесты при каждом изменении до его отправки.
- Выполнять весь набор тестов в среднем за 30 минут или меньше.
- Выполнять интеграционные и юнит-тесты меньше чем за 5 минут.
- Выполнять юнит-тесты меньше чем за 1 минуту.

> ***В первую очередь займитесь тем, что доставляет самые большие неудобства***
>
> Лучше всего об этом сказали Джез Хамбл и Дэвид Фарли в своей книге *Continuous Delivery*[1]: «Если операция болезненная, выполняйте ее чаще и в первую очередь». Если задача сложная или занимает много времени, мы инстинктивно откладываем ее на потом, но лучше делать наоборот! Если вы изолируете себя от проблемы, у вас будет меньше мотивации ее решить. Поэтому лучший способ справиться с проблемой — заниматься ею чаще!

Целевые показатели, к которым следует стремиться, зависят от проекта, но в большинстве случаев они должны примерно совпадать с теми, какие определил Шридхар.

[1] Хамбл Дж., Фарли Д. «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ».

Пирамида тестов

Вы, наверное, заметили, что цели, поставленные Шридхаром, отличаются в зависимости от типа теста:

- Выполнять весь набор тестов в среднем за 30 минут или меньше.
- Выполнять интеграционные и юнит-тесты меньше чем за 5 минут.
- Выполнять юнит-тесты меньше чем за 1 минуту.

Я не буду подробно останавливаться на различиях между этими видами тестов. О них вы можете узнать из книг, посвященных тестированию.

Что это за тесты? Шридхар имеет в виду пирамиду тестов, то есть общую визуализацию типов тестов, которые необходимы в большинстве программных проектов, и их примерное соотношение.

В основе пирамиды лежит тот факт, что подавляющее большинство тестов в наборе являются юнит-тестами, интеграционных тестов значительно меньше и, наконец, сквозных тестов совсем немного. При постановке целей тестирования для сайта Dog Picture Шридхар руководствовался этой пирамидой:

Сервисные тесты, тесты пользовательского интерфейса (UI-тесты), сквозные тесты, интеграционне тесты и...

Если вы уже сталкивались с пирамидами тестов, вы могли использовать другую терминологию. Термины здесь не так важны, как то, что существуют различные типы тестов и что тесты в нижней части пирамиды меньше всего связаны между собой, а тесты в верхней части — наоборот. *Связанность* (coupling) здесь означает растущую взаимозависимость между тестируемыми компонентами, что обычно приводит к усложнению тестов, требующих больше времени на выполнение.

Сначала быстрые тесты

Шридхар использует подход на основе пирамиды тестов в том числе потому, что знает, что один из способов получить быструю обратную связь — начать группировать и выполнять тесты в зависимости от их типа. Пол М. Дюваль в книге «Continuous Integration»[1] предлагает следующее:

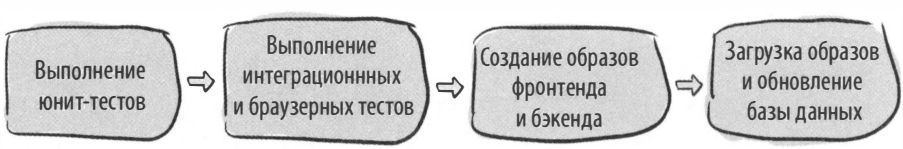
Выполняйте сначала самые быстрые тесты.

Сейчас на сайте Dog Picture все тесты выполняются одновременно, но когда Шридхар выделяет в коде юнит-тесты и запускает их отдельно, оказывается, что они уже выполняются менее чем за минуту. То есть он уже достиг своей первой цели!

Если он сможет сделать так, чтобы все разработчики сайта Dog Picture запускали только юнит-тесты, то они смогут быстро получать оперативный отклик на свои изменения. Они смогут запускать эти тесты локально для своих изменений перед тем, как вносить эти изменения в общий код. Осталось только найти способ упростить изолированный запуск тестов. Для этого есть несколько вариантов:

- Договориться о местоположении тестов — это самый простой способ: например, всегда хранить юнит-тесты рядом с кодом, который они тестируют, а интеграционные и системные тесты — в соответствующих папках. То есть для юнит-тестирования запускать тесты в папках с кодом (или в папке с названием unit); для интеграционного — тесты в папке integration tests, и т. д.
- Многие языки позволяют указывать тип теста, например, с помощью флага сборки в Go (можно изолировать интеграционные тесты, требуя запускать их с флагом сборки, таким как `integration`) или с помощью декоратора, отмечающего тесты разных типов, если используется пакет pytest в Python.

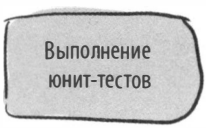
К счастью, команда Dog Picture уже частично соблюдала договоренность о расположении тестов: тесты браузера находились в папке tests/browser, а юнит-тесты — рядом с кодом. Интеграционные были смешаны с юнит-тестами, поэтому Шридхар переместил их в папку tests/integration, а затем обновил пайплайн, чтобы он выглядел следующим образом:



[1] Дюваль Пол М. и др. «Непрерывная интеграция: улучшение качества программного обеспечения и снижение риска».

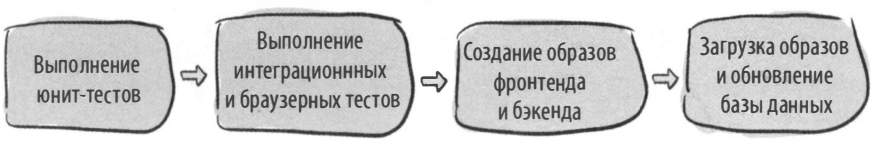
Два пайплайна

До сих пор разработчикам приходилось ждать ночного запуска пайплайна, чтобы проверить свои изменения, поскольку его выполнение занимало очень много времени. Однако новая задача «Выполнение юнит-тестов», которую создал Шридхар, выполняется менее чем за минуту, поэтому ее можно запускать для каждого изменения, даже до слияния. Шридхар обновляет автоматизацию сайта Dog Picture так, чтобы при каждом изменении перед слиянием запускался следующий пайплайн, содержащий только одну задачу:



Этот (крошечный) пайплайн запускается для каждого изменения перед слиянием.

На сайте Dog Picture теперь два пайплайна: описанный выше — который запускается при каждом изменении, и более длинный и медленный — который запускается ночью:



Этот пайплайн запускается каждую ночь.

Два пайплайна — это хорошо или плохо? Наша цель — всегда *смещаться влево* и получать как можно больше информации как можно раньше (подробнее об этом в следующей главе), так что эта ситуация не идеальна. Но создав отдельный, более быстрый пайплайн, который может работать с каждым изменением, Шридхар смог улучшить положение: раньше разработчики вообще не получали обратной связи о своих изменениях до их слияния. Теперь у них будет хоть какая-то информация. В зависимости от потребностей проекта в нем может быть один пайплайн или несколько. Подробнее об этом читайте в главе 13.

ВАЖНО

Если ваш набор тестов медленный, в первую очередь обеспечьте автоматический запуск и выполнение самых быстрых тестов. Даже если выполнение всех тестов по-прежнему будет занимать много времени, это позволит получать определенную часть сигнала несколько быстрее.

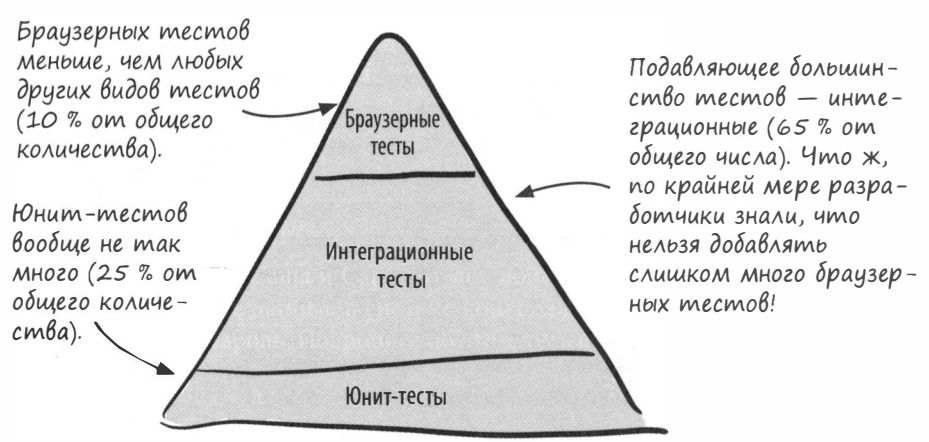
Соблюдение баланса

Шридхар немного исправил ситуацию, но его изменение практически не повлияло на интеграционные и браузерные тесты. Они остались такими же медленными, как и раньше, а разработчикам по-прежнему приходится ждать следующего утра, чтобы узнать результаты сделанных изменений.

На следующем шаге Шридхар возвращается к пирамиде тестирования. В последний раз он думал об относительной скорости каждого блока тестов. Но теперь он собирается рассмотреть относительное распределение тестов.

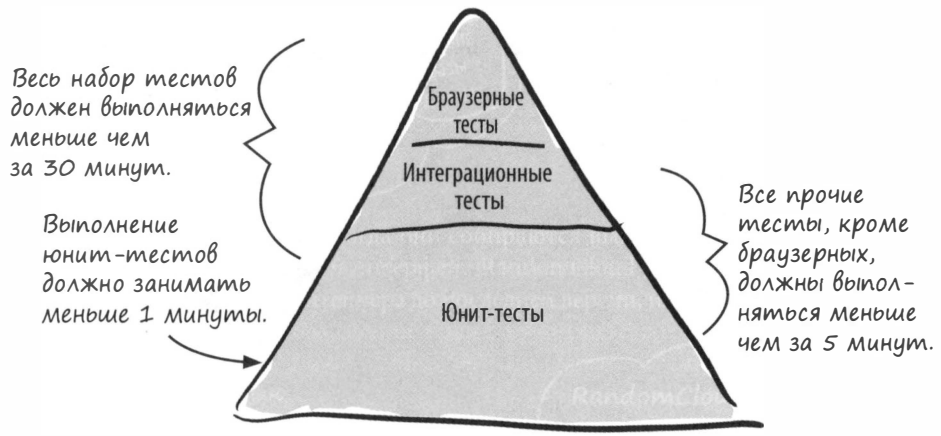
Пирамида, кроме прочего, дает рекомендации, какое количество тестов каждого типа необходимо. Почему? Потому что по мере продвижения по пирамиде вверх тесты становятся медленнее. (А еще их становится сложнее поддерживать, но это тема другой книги!)

Шридхар подсчитал количество тестов в наборе для веб-сайта Dog Picture и сравнил его пирамиду с идеальной. Для сайта Dog Picture она выглядит примерно так:



Меняем пирамиду тестов

Почему Шридхар обращает внимание на соотношение типов тестов в пирамиде? Потому что он знает, что оно не является постоянным. Его можно менять, и это изменение может ускорить работу всего набора тестов. Вспомним цели, которые Шридхар задал относительно времени выполнения тестов:



Шридхар хочет сократить время выполнения интеграционных и юнит-тестов до 5 минут и меньше. Сейчас интеграционные тесты составляют 65 % от общего числа тестов. Остальное — это 10 % браузерных тестов и 25 % юнит-тестов. Учитывая, что интеграционные тесты выполняются медленнее, чем юнит-тесты, представьте, какая разница во времени получится, если изменить их соотношение (при том же общем количестве тестов) — если бы интеграционные тесты составляли только 20 % от общего числа тестов, а юнит-тесты тесты, напротив, 70 %. Если примерно 2/3 существующих (медленных) интеграционных тестов заменить на (более быстрые) юнит-тесты, это немедленно повлияет на общее время выполнения всего набора.

Поставив перед собой конечную цель — скорректировать соотношение типов тестов, чтобы ускорить работу тестов в целом, Шридхар формулирует ряд новых задач:

- Увеличить долю юнит-тестов с 25 до 70 %.
- Уменьшить процент интеграционных тестов с 65 до 20 %.
- Оставить долю браузерных тестов на уровне 10 %.

Безопасная корректировка тестов

Шридхар хочет изменить соотношение между интеграционными и юнит-тестами. Он собирается сделать следующее:

- увеличить процент юнит-тестов с 25 до 70 %;
- уменьшить долю интеграционных тестов с 65 до 20 %.

Ему нужно увеличить количество юнит-тестов, одновременно уменьшив число интеграционных тестов. Как это сделать безопасно и с чего вообще начать?

При измерении покрытия кода запускаются и сами юнит-тесты, поэтому иногда эти задачи объединяют в одну. (Если задача выполнения юнит-тестов завершится неудачей, то задача измерения покрытия, скорее всего, тоже!)

Шридхар обращает внимание на то, что пайплайн сайта Dog Picture не предусматривает измерения тестового покрытия кода. Пайплайн запускает тесты, затем выполняет сборку и развертывание ПО, но при этом ни разу не измеряет покрытие кода, которое обеспечивают тесты. Таким образом, первым делом необходимо добавить в пайплайн функцию измерения тестового покрытия параллельно с запуском тестов:

Поскольку задача измерения тестового покрытия выполняется так же быстро, как и задача юнит-тестирования, Шридхар может добавить ее в пайплайн, который запускается до объединения изменений.

Постойте! А где же линтинг? Я читал главу 4 и знаю, что линтинг тоже важен. Разве Шридхар не должен добавить линтинг?

Я полностью согласна, и это, вероятно, будет следующим шагом Шридхара, когда он разберется с тестами, однако он не может решать одновременно несколько задач! В главе 2 мы рассмотрели все элементы, которые должен содержать пайплайн CD, включая линтинг.

Покрытие тестами

Шридхар считает, что для того, чтобы провести безопасную корректировку соотношения интеграционных и юнит-тестов, необходимо начать измерять покрытие тестами. Что такое измерение тестового покрытия и почему оно важно?

Покрытие тестами (test coverage) — это оценка того, насколько эффективно тесты используют тестируемый код. В частности, отчеты о тестовом покрытии покажут, строка за строкой, какой проверяемый код используется тестами, а какой нет. Например, на сайте Dog Picture есть такой юнит-тест для проверки логики поиска по тегам:

```
def test_search_by_tag(self):
  search = _new_search()
  results = search.by_tags(["fluffy"])
  self.assertDogResultsEqual(results, "fluffy", [Dog("sheldon")])
```

Этот тест проверяет метод `by_tags` на объекте `Search`, который выглядит так:

```
def by_tags(self, tags):
  try:
    query = build_query_from_tags(tags)
  except EmptyQuery:
    raise InvalidSearch()
  result = self._db.query(query)
  return result
```

Программа измерения тестового покрытия запустит тест `test_search_by_tag`, проверит, какие строки кода в `by_tags` выполняются, и сгенерирует отчет о проценте покрытых строк. Покрытие для `by_tags` тестом `test_search_by_tag` выглядит следующим образом: строки, которые выполняются тестом, выделены светлым, а строки, которые не выполняются, — темным:

```
def by_tags(self, tags):
  try:
    query = build_query_from_tags(tags)
  except EmptyQuery:
    raise InvalidSearch()
  result = self._db.query(query)
  return result
```

Логично, что этот тест не проверяет условия возникновения соответствующих ошибок; это лучше оставить для другого теста. Однако в данном случае `test_search_by_tag` — единственный юнит-тест для `by_tags`. Следовательно, эти строки вообще не покрываются никаким тестом. Таким образом, для этого метода тестовое покрытие составляет три строки из пяти, или 60 %.

Критерии покрытия

В предыдущем примере использовался критерий покрытия, получивший название *покрытие операторов* (statement coverage), который проверяет каждый оператор на предмет того, был ли он хотя бы раз выполнен. Можно использовать и другие, более точные критерии, например *покрытие условий* (condition coverage). Если оператор `if` содержит несколько условий, при использовании покрытия операторов будет считаться, что оператор покрыт, если к нему было хотя бы одно обращение, а покрытие условий потребует полного анализа каждого условия. В этой главе для начала я остановлюсь на покрытии оператора.

Как обеспечить тестовое покрытие

Важно помнить, что пока Шридхар вносит эти изменения, другие разработчики продолжают работать и отправлять функции. Коллеги присылают новые функции (и исправления багов), а иногда (надеюсь, всегда или почти всегда!) и тесты. Это значит, что даже при том, что Шридхар отслеживает покрытие кода тестами, оно может снижаться!

Но к счастью, Шридхар знает не только как предотвратить этот процесс, но и как увеличить количество юнит-тестов.

Прежде чем двигаться дальше, Шридхар собирается обновить задачу измерения покрытия таким образом, чтобы она останавливала пайплайн, если покрытие снижается. После добавления этого изменения он может быть уверен, что тестовое покрытие в коде как минимум не снизится, а в идеале даже увеличится.

Звучит знакомо? Этот подход похож на тот, который Бекки использовала в главе 4, когда работала с линтингом. У измерения количества ошибок линтинга и измерения тестового покрытия много общего.

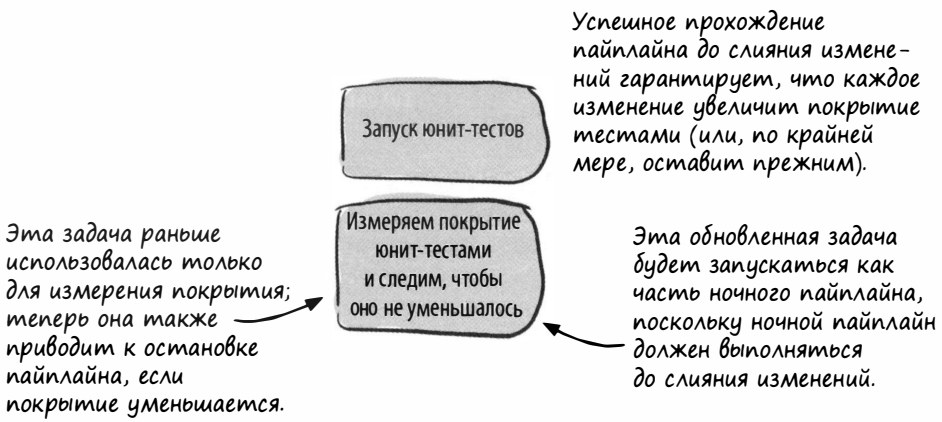
Помимо решения общей проблемы, это еще и отличный способ разделить нагрузку, чтобы Шридхару не приходилось выполнять всю работу одному! Он обновляет задачу, запускающую измерение тестового покрытия, чтобы выполнялся этот скрипт:

Вы можете заметить, что это вариация скрипта для линтинга, который Бекки написала в главе 4.

```
# при запуске пайплайна он будет передавать в качестве аргументов
# пути к файлам, которые изменились в PR
paths_to_changes = get_arguments()

# измерение покрытия кода для файлов, которые были изменены
coverage = measure_coverage(paths_to_changes)

# измерение покрытия для файлов до внесения изменений; это может
# производиться путем извлечения значений из какого-либо хранилища
# или же просто повторным измерением покрытия
# для тех же файлов в главной ветке проекта (то есть до внесения изменений)
prev_coverage = get_previous_coverage(paths_to_changes)

# сравнить покрытие с учетом изменений и предыдущее значение покрытия
if coverage < prev_coverage:
  # слияния изменений не должно произойти, если оно уменьшит покрытие
  fail('coverage reduced from {} to {}'.format(prev_coverage, coverage))
```

Тестовое покрытие в пайплайне

Добавив этот скрипт в пайплайн перед слиянием веток, Шридхар упорядочил существующую проблему покрытия: ребята добавляли юнит-тесты как попало, без разбора. С помощью автоматизации измерения покрытия и блокировки изменений PR, снижающих покрытие, разработчики смогут более обоснованно решать, что покрывать тестами, а что нет. После того как Шридхар обновил задачу измерения покрытия юнит-тестами так, чтобы обеспечить соблюдение требований к тестовому покрытию, часть пайплайна до слияния стала выглядеть следующим образом:



Это очень незначительное изменение по сравнению с предыдущей итерацией, но теперь Шридхар может продолжить с уверенностью, что функции и исправления багов, добавляемые во время его работы, либо увеличат покрытие, либо, как минимум, оставят без изменений.

Нужно ли создавать эту функцию самостоятельно?

Зависит от обстоятельств! Для большинства языков программирования доступны различные инструменты измерения покрытия и даже хранения и составления отчетов. Многие все равно предпочитают писать свои собственные инструменты, потому что их не очень сложно реализовать и они предоставляют бóльшую свободу управления их работой. Проанализируйте доступные инструменты и решите, стоит ли этим заниматься.

Перемещение тестов по пирамиде с учетом покрытия

На данном этапе количество юнит-тестов, скорее всего, начнет неуклонно расти даже без дальнейшего вмешательства, потому что Шридхар сделал обязательным условием добавление юнит-тестов одновременно с изменениями, которые вносят разработчики.

Будет ли этого достаточно для достижения его целей? Вспомним их:

- увеличить процент юнит-тестов с 25 до 70 %;
- уменьшить долю интеграционных тестов с 65 до 20 %.

Со временем эти соотношения, вероятно, будут меняться в нужном направлении, но недостаточно быстро, чтобы произошли кардинальные изменения, к которым стремится Шридхар. Ему наверняка придется написать дополнительные юнит-тесты и, возможно, удалить существующие интеграционные. Как определить, какие добавить, а какие убрать?

Шридхар просматривает отчеты о покрытии кода, находит код с наименьшим процентом покрытия и обращает внимание на то, какие строки не покрыты. К примеру, он анализирует покрытие функции `by_tags`, рассмотренной чуть выше.

```
def by_tags(self, tags):
  try:
    query = build_query_from_tags(tags)
  except EmptyQuery:
    raise InvalidSearch()
  result = self._db.query(query)
  return result
```

Ошибка, связанная с пустым запросом, не покрыта юнит-тестами. Из этого Шридхар делает вывод, что здесь нужно добавить юнит-тест. Кроме того, если найдется интеграционный тест, который охватывает ту же логику, его в принципе можно будет удалить. Поэтому он просматривает интеграционные тесты и находит тест под названием `test_invalid_queries`. Этот тест создает экземпляр работающей серверной службы (что, собственно, и делают все интеграционные тесты), затем

> ***Нужно ли измерять покрытие для интеграционных и сквозных тестов?***
>
> Чтобы получить полное представление о покрытии тестами, у вас может возникнуть желание измерить покрытие для интеграционных и сквозных тестов. Иногда это можно сделать; обычно для этого требуется добавить в систему дополнительную отладочную информацию, которую можно использовать для измерения покрытия кода во время выполнения этих высокоуровневых тестов. Вам может показаться это полезным; однако, как правило, необходимый инструмент придется создавать самостоятельно, и он может дать вам ложное чувство уверенности. Лучше всего всегда обеспечивать высокое покрытие юнит-тестами, поскольку эта метрика важна сама по себе, а если отслеживать только общее покрытие для набора тестов, ее можно упустить.

формирует ошибочные запросы и проверяет, что они не проходят. Анализируя этот тест, Шридхар понимает, что он может покрыть все случаи проверки некорректных запросов с помощью юнит-тестов. Он пишет ряд тестов, которые выполняются менее чем за секунду, и теперь может удалить `test_invalid_queries`, выполнение которого занимало около 20 секунд, а возможно, и больше. И он по-прежнему может быть уверен, что набор тестов выявит те же ошибки, что и до внесения изменений.

Как двигаться вниз по пирамиде?

Чтобы продолжать увеличивать долю юнит-тестов, Шридхар применяет к набору тестов следующий паттерн:

1. Он ищет пробелы в покрытии юнит-тестами (строки кода, которые не покрыты). В целях эффективности он начинает с пакетов и файлов, в которых доля юнит-тестов наименьшая.
2. Для обнаруженного неохваченного кода он добавляет юнит-тесты.
3. Он просматривает более медленные тесты (в данном случае интеграционные) в поисках тех, которые включают проверку логики, покрываемую юнит-тестами, и обновляет или удаляет их.

Таким образом, он может как значительно увеличить количество юнит-тестов, так и уменьшить количество интеграционных (то есть увеличить количество быстрых тестов и сократить количество медленных). Наконец, он проверяет интеграционные тесты на предмет дублирующего покрытия; для каждого интеграционного теста он задает следующие вопросы:

- Охвачен ли этот случай юнит-тестами?
- Почему этот тест может не пройти, если юнит-тесты прошли успешно?

Если этот случай уже покрыт юнит-тестами и нет ничего, что могло бы привести к провалу интеграционного теста, когда юнит-тесты прошли успешно, то интеграционный тест можно безопасно удалить.

Но если я это сделаю, не потеряю ли я данные? Разве интеграционные тесты не лучше юнит-тестов? Я видел эти мемы типа: юнит-тестов недостаточно.

Вы правы! Вопрос в том, сколько интеграционных тестов необходимо. Интеграционные тесты служат для проверки того, что все отдельные блоки кода корректно связаны между собой. Если вы протестируете блоки кода, а затем проверите правильность их соединения, вы обеспечите практически полное покрытие. Это своего рода компромисс между затратами и выгодой: стоит ли тратить ресурсы на проведение и поддержку интеграционных тестов, которые охватывают ту же область, что и юнит-тесты, в расчете на то, что они смогут выявить специфический случай, который вы упустили? Ответ зависит от того, над чем вы работаете. Если на карту поставлены жизни людей, ответ может быть утвердительным; важно найти правильный компромисс для создаваемого продукта.

Ваш ход: определите недостающие тесты

Шридхар обнаружил, что класс `Search` в целом имеет очень низкое покрытие, и работает над отчетами, чтобы его увеличить. Он изучает покрытие для метода `from_favorited_search` и видит следующее:

```
def from_favorited_search(self, favorite):
  try:
    cached_result = self._cache.get_result(favorite.query())
  except CacheError:
    cached_result = None
  if cached_result is None:
    result = self._db.query(favorite.query())
  else:
    result = cached_result.result()
  return result
```

Он ищет интеграционные тесты, которые проверяют логику поиска по избранному (favorited search), и находит их:

```
test_favorited_search_many_results
test_favorited_search_no_results
test_favorited_search_cache_connection_error
test_favorited_search_many_results_cached
test_favorited_search_no_results_cached
```

Какие из этих интеграционных тестов Шридхар должен удалить? Какие юнит-тесты он может добавить?

Ответы

Это классический сценарий, в котором всю тяжелую работу выполняют интеграционные тесты. Юнит-тесты охватывают только один путь — без кэшированных результатов и без ошибок, а интеграционные тесты стремятся покрыть все. Шридхар собирается изменить эту ситуацию: вместо того чтобы проводить юнит-тесты для одного пути, а все остальные случаи обрабатывать с помощью интеграционных тестов, он заменяет все интеграционные тесты на `test_favorited_search` и добавляет юнит-тесты для всех вариантов, которые прежде проверялись интеграционными тестами.

Унаследованные тесты и FUD

Изменять или даже удалять тесты, которые существуют уже очень давно, страшновато! В таких случаях мы часто сталкиваемся с явлением *FUD* (fear, uncertainty and doubt): страх, неуверенность и сомнения.

Если поддаться FUD, можно решить, что вносить изменения в существующие наборы тестов опасно: тестов слишком много, сложно сказать, что они тестируют, и возникает страх удалить тест, на котором, как окажется потом, все держалось.

Если вы обнаружили, что рассуждаете подобным образом, стоит задуматься о том, что же такое FUD на самом деле и откуда он берется. В конечном счете, все дело в букве *F* — страх. Это страх, что вы можете сделать что-то не так или ухудшить положение, и именно он удерживает вас от внесения изменений.

В таком случае вспомните, для чего мы проводим тесты: их назначение — поддержать нас, дать уверенность в том, что можно безопасно вносить изменения, которые будут делать именно то, что необходимо. FUD — это полная противоположность тому, что должны делать тесты. Они придают нам уверенность, а FUD ее отнимает.

Не позволяйте FUD сковывать вас! Когда вы слышите, как FUD шепчет вам, что вносить изменения слишком опасно, противопоставляйте ему сухие факты. Вспомните, что такое тесты: это формальное описание того, как, по мнению их создателя, должна вести себя система, — ни больше и ни меньше. По сути, они даже не являются системой! Не поддавайтесь страху; сделайте глубокий вдох и спросите себя: понимаю ли я, что делает этот тест? Если нет, найдите время, чтобы разобраться и понять. Если вы все понимаете, считайте, что вы тот, кто ответствен за внесение изменений. Если вы не внесете их, возможно, этого не сделает никто, и чувство FUD, которое остальные испытывают по отношению к тестам, со временем будет только расти.

В общем, если работать под влиянием страха и поддаваться FUD, вы не попробуете новое. Тем самым вы не сможете совершенствоваться, а если вы не исправите свои тесты, то со временем они неизбежно станут только хуже.

Просто скажи FUD: нет!

ВАЖНО

При работе с медленными наборами тестов оцените их через призму пирамиды тестирования — это поможет понять, где именно что-то идет не так. Если у вашей пирамиды слишком перегружена вершина (распространенная проблема), оцените уровень тестового покрытия, чтобы быстро оптимизировать соотношение тестов и определить, какие из них можно заменить на более быстрые и простые в обслуживании юнит-тесты.

Параллельный запуск тестов

После напряженной работы над интеграционными и юнит-тестами Шридхар добился (по его мнению) максимально возможного на данный момент результата. Он достиг поставленных целей по улучшению соотношения тестов:

- увеличил долю юнит-тестов с 25 до 72 % (цель — 70 %); и
- уменьшил долю интеграционных тестов с 65 до 21 % (цель — 20 %).

Юнит-тесты по-прежнему выполняются меньше чем за минуту, однако даже при достигнутых результатах на выполнение интеграционных тестов уходит около 35 минут. Главной целью Шридхара было добиться того, чтобы интеграционные и юнит-тесты в совокупности выполнялись менее чем за 5 минут. Несмотря на то что он сократил общее время (сэкономив более часа), тесты все еще выполняются медленнее, чем ему хотелось бы. Он хотел бы проводить их перед слиянием, и показатель 35 минут делает это почти реальным, но у него есть еще один козырь, который позволит ему существенно сократить и это время.

Он собирается запускать интеграционные тесты параллельно. В большинстве наборов по умолчанию тесты запускаются по очереди. Вот пример интеграционных тестов, оставшихся после того, как Шридхар сократил их количество, и среднее время их выполнения:

1. **`test_search_query`** (20 секунд)
2. **`test_view_latest_dog_pics`** (10 секунд)
3. **`test_log_in`** (20 секунд)
4. **`test_unauthorized_edit`** (10 секунд)
5. **`test_picture_upload`** (30 секунд)

Поочередное выполнение этих тестов занимает в среднем 90 секунд (20 + 10 + 20 + 10 + 30 = 90). Шридхар обновляет интеграционные тесты, чтобы они запускались параллельно и в максимально возможном объеме. В большинстве случаев это означает одновременный запуск по одному тесту на каждое ядро процессора. На восьмиядерном компьютере пять тестов могут легко выполняться параллельно, а это означает, что выполнение всех тестов займет столько времени, сколько длится самый долгий тест: только 30 секунд, а не все 90.

> Если один тест выполняется намного дольше других, вы станете его заложником; например, если один тест сам по себе занимает 30 минут, распараллеливание не поможет, потребуется исправление самого теста.

После такой чистки на сайте Dog Picture осталось 116 интеграционных тестов. Их поочередное выполнение занимает около 35 минут с учетом того, что на каждый в среднем уходит по 18 секунд. Параллельный запуск на восьмиядерной машине позволяет одновременно выполнять восемь тестов, и весь набор будет завершен примерно за одну восьмую часть времени, или примерно 4,5 минуты. Запустив интеграционные тесты параллельно, Шридхар наконец достиг своей цели — сократить время выполнения интеграционных и юнит-тестов до 5 минут.

Когда можно выполнять тесты параллельно?

Любые ли тесты подходят для параллельного запуска? Не совсем. Чтобы тесты можно было запускать параллельно, они должны соответствовать следующим критериям:

- не зависеть друг от друга;
- выполняться в любом порядке;
- не мешать друг другу (например, используя общую память).

Старайтесь писать полностью независимые тесты, которые не мешают друг другу. Если вы пишете правильные тесты, у вас не возникнет проблем с их параллельным выполнением.

Вероятно, самое сложное условие — чтобы тесты не мешали друг другу. Это легко может произойти случайно, особенно при тестировании кода, использующего глобальное хранилище. Как следует потрудившись, вы сможете найти способ исправить тесты так, чтобы они были полностью изолированы друг от друга, что, скорее всего, поможет повысить и общее качество кода: обеспечить слабую связанность (coupling) и сильную связность (cohesion).

> ***Параллельный запуск юнит-тестов — это запах кода***
>
> Помните, что юнит-тесты предназначены для тестирования изолированной функциональности и должны быть *быстрыми*, порядка нескольких секунд или меньше. Если ваши юнит-тесты выполняются несколько минут или дольше и вы подумываете о том, чтобы ускорить их с помощью параллельного запуска, скорее всего, они делают слишком много и на самом деле это интеграционные или системные тесты; велика вероятность, что у вас вообще нет юнит-тестов.

Обновив набор тестов для сайта Dog Picture так, чтобы они работали параллельно, Шридхар обнаружил несколько мешающих друг другу тестов, которые следует исправить, но после изменения он добился выполнения и интеграционных, и юнит-тестов менее чем за 5 минут.

> ***Нужно ли создавать функциональность параллельных тестов самому?***
>
> Скорее всего, нет! Это настолько частый способ оптимизации тестов, что возможность запускать тесты параллельно предусмотрена для большинства языков программирования — либо из самого пакета, либо с помощью стандартных библиотек. Например, в Python можно запускать тесты параллельно с помощью такой библиотеки, как testtools, или расширения популярной библиотеки pytest. В Go эта функциональность реализуется в возможности пометить тест как пригодный для параллельного запуска при его написании с помощью команды `t.Parallel()`. Поищите соответствующую информацию для своего языка в документации по параллельному или одновременному выполнению тестов.

Обновление пайплайнов

Теперь, когда Шридхар достиг своей цели — выполнять интеграционные и юнит-тесты меньше чем за 5 минут, — он может добавить интеграционные тесты в пайплайн до слияния. Тогда разработчики будут получать обратную связь как по интеграционным, так и по юнит-тестам прежде, чем произойдет слияние их изменений.

Для этого ему придется внести правки в набор задач в пайплайне сайта Dog Picture, поскольку одна из задач до сих пор одновременно выполняет запуск и интеграционных, и браузерных тестов.

К счастью, тесты уже полностью готовы к этому изменению. Как вы помните, браузерные тесты уже находятся в отдельной папке с названием tests/browser. Когда Шридхар обновлял пайплайн, чтобы вначале запускались юнит-тесты, он отделил интеграционные тесты и поместил их в папку tests/integration. Это упрощает выполнение последнего шага раздельного запуска интеграционных и браузерных тестов.

Почему бы не выполнять все тесты параллельно?

Шридхар предположил, что если юнит-тесты, выполняемые за секунды, не сработают, то нет смысла запускать интеграционные тесты, потому что они, скорее всего, тоже не пройдут, однако оба варианта подходят. Подробнее см. главу 13 о реализации пайплайнов.

А затем Шридхар добавляет задачу запуска интеграционных тестов в пайплайн перед слиянием веток. Пайплайн быстро остановится, если возникнут проблемы с юнит-тестами, и вся проверка займет менее 5 минут.

Все еще слишком медленно!

Как следует поработав над интеграционными и юнит-тестами, Шридхар добился (по его мнению) максимально возможного на данный момент результата. Он достиг поставленных целей в части соотношения тестов:

- увеличил долю юнит-тестов с 25 до 72 % (цель — 70 %) и
- уменьшил долю интеграционных тестов с 65 до 21 % (цель — 20 %).

Завершена ли работа? Шридхар оценивает свои глобальные цели:

- *Запускать тесты при каждом изменении до его отправки.* Цель почти достигнута: так выполняются интеграционные и юнит-тесты, но не браузерные.
- *Выполнять весь набор тестов в среднем за 30 минут или меньше.* Шридхар сократил время выполнения интеграционных тестов: раньше они занимали 35 минут, а теперь — около 5. Раньше весь набор тестов выполнялся за 2 часа 35 минут, а теперь — чуть больше чем за 2 часа. Это значительное улучшение, но Шридхар все еще не достиг поставленной цели.
- *Выполнять интеграционные и юнит-тесты меньше чем за 5 минут.* Сделано!
- *Выполнять юнит-тесты меньше чем за 1 минуту.* Сделано!

Весь набор тестов выполняется в среднем за 2 часа 5 минут:

- *юнит-тесты* — менее 1 минуты;
- *интеграционные тесты* — около 5 минут;
- *браузерные тесты* — остальные 2 часа.

Последняя проблема — это браузерные тесты. Все это время они были самыми медленными в наборе. При среднем времени выполнения 2 часа, неважно, насколько оптимизирована остальная часть набора, если ничего не сделать с браузерными тестами, тесты все равно будут занимать больше 2 часов.

Может ли Шридхар поступить как раньше и убрать браузерные тесты, заменив их интеграционными и юнит-тестами? Это вариант, но изучив набор браузерных тестов, Шридхар не нашел ни одного подходящего, чтобы удалить! Тесты уже очень четко структурированы и хорошо продуманы, а браузерных тестов всего 10 % от общего количества (это примерно 50 тестов), что является хорошим показателем.

Шардинг тестов, он же параллельный режим ++

Шридхар застрял с браузерными тестами в их нынешнем виде, когда на их выполнение уходит около 2 часов. Означает ли это, что ему придется распрощаться с целью выполнять весь набор тестов для каждого изменения меньше чем за 30 минут?

К счастью, нет! Потому что у него в запасе остался еще один хитрый прием: *шардинг*. Шардинг (или шардирование) — это метод, который очень похож на параллельный запуск тестов, но он увеличивает количество тестов, которые можно выполнять одновременно, *путем их распараллеливания на нескольких машинах*.

Сейчас все 50 браузерных тестов выполняются на одном компьютере поочередно. Каждый тест выполняется в среднем около 2,5 минуты. Сначала Шридхар пытается запускать тесты параллельно, но они настолько требовательны к процессору и памяти, что выигрыш оказывается незначительным (а в некоторых случаях тесты перехватывают ресурсы друг у друга, существенно замедляя работу). Одна машина может выполнять одновременно только один тест.

С помощью шардирования (разделения) выполнения тестов Шридхар распределит набор браузерных тестов так, чтобы он выполнялся одновременно на нескольких машинах, каждая из которых будет по очереди запускать некоторое подмножество тестов, что позволит сократить общее время их выполнения.

Словарик

Здесь мы называем распараллеливание тестов на нескольких машинах *шардингом* (sharding), но в системах непрерывной доставки можно встретить и другие термины. Иногда это называют *расщеплением тестов* (splitting), а иногда просто *параллельным выполнением тестов*. В этом контексте слово «*параллельно*» означает «на нескольких машинах», в отличие от данной главы, где термин «*параллельно*» используется для обозначения одновременного запуска нескольких тестов на одной машине. В любом случае шардинг в целом можно рассматривать как распараллеливание тестов (многоядерное), но на нескольких машинах (многомашинное).

Что, если Шридхар увеличит мощность компьютера? Сможет ли он тогда обойтись параллельным выполнением тестов на одной машине?

Это могло бы помочь, но как вы, скорее всего, знаете, мощность машин постоянно увеличивается, а мы в ответ создаем все более сложные программы и тесты! Поэтому хотя использование более мощных компьютеров могло бы помочь Шридхару, я покажу, что можно сделать, когда такой возможности нет. Я не буду углубляться в подробности работы процессора и объема памяти, которые он использует, поскольку то, что кажется мощным сегодня, окажется сущим пустяком завтра!

Как использовать шардинг

Словарик

Шард (shard) — отдельная машина, доступная для выполнения подмножества тестов.

Шардинг подразумевает ускорение выполнения набора давно работающих тестов за счет использования дополнительных единиц оборудования (запуск на нескольких компьютерах вместо одного). Но как это выглядит на практике? Возможно, вы представляете себе сложную систему, требующую рабочих узлов, взаимодействующих с центральным контроллером, но не волнуйтесь, все гораздо проще!

Суть в том, чтобы сформировать несколько шардов и назначить каждому из них выполнение определенного подмножества тестов. Распределить тесты по шардам можно несколькими способами. Рассмотрим их в порядке возрастания сложности:

1. Запустить тесты в определенном порядке и назначить каждому шарду набор индексов для выполнения.
2. Явно назначить каждому шарду подмножество тестов для выполнения (например, по имени).
3. Отслеживать атрибуты тестов из предыдущих запусков (например, сколько времени требуется каждому тесту для выполнения) и использовать эти атрибуты для распределения тестов по шардам (возможно, с использованием их имен, как в варианте 2).

Разберемся, что такое шардинг тестов, рассмотрев вариант 1 более подробно. Например, представьте, что следующие 13 тестов распределены между тремя рабочими компьютерами:

На 3 шардах есть 13 тестов: 13/3 = = 4,333, которые я округлю до 5, отдав каждому из первых двух шардов по 5 тестов, а последнему — оставшиеся 3.

```
0. test_login
1. test_post_pic
2. test_rate_pic
3. test_browse_pics
4. test_follow_dog
5. test_view_leaderboard
6. test_view_logged_out
7. test_edit_pic
8. test_post_forum
9. test_edit_forum
10. test_share_twitter
11. test_share_instagram
12. test_report_userOur
```

Первый шард с номером 0 получит 5 тестов: от начального индекса 0 до конечного индекса 4.

Шард 1 также получит 5 тестов: от начального индекса 5 до конечного 9.

Шард 2 получит остаток: от начального индекса 10 до конечного 12.

Можно распределить эти тесты с помощью первого метода, запустив подмножество всех тестов на каждом из трех шардов. Если вы работаете на Python, один из способов сделать это — использовать библиотеку Python pytest-shard:

```
pytest --shard-id=$SHARD_ID --num-shards=$NUM_SHARDS
```

Например, на шарде 1 будет запускаться следующее:

```
pytest --shard-id=1 --num-shards=3
```

Более сложный шардинг

Шардинг по индексам достаточно прост, но как быть, если случай нестандартный? Браузерные тесты Шридхара выполняются в среднем за 2,5 минуты, но что, если некоторые из них занимают гораздо больше времени?

Вот тут-то и пригодятся более сложные схемы шардинга. Например, в третьем варианте из ранее перечисленных можно отслеживать атрибуты тестов из предыдущих прогонов и использовать их для распределения тестов по шардам с помощью их имен.

Для этого необходимо хранить данные о времени выполнения тестов. Например, возьмем 13 тестов из предыдущего примера и представим, что храним данные о том, сколько минут в среднем потребовалось на выполнение каждого из них за последние три прогона:

0.	`test_login (1.5, 1.7, 1.6)`	*В среднем 1,6 минуты*
1.	`test_post_pic (3, 3.1, 3.2)`	*В среднем 3,1 минуты*
2.	`test_rate_pic (0.8, 0.9, 0.7)`	*В среднем 0,8 минуты*
3.	`test_browse_pics (2, 2, 2)`	*В среднем 2,0 минуты*
4.	`test_follow_dog (0.8, 0.8, 0.8)`	*В среднем 0,8 минуты*
5.	`test_view_leaderboard (1.8, 2.0, 1.9)`	*В среднем 1,9 минуты*
6.	`test_view_logged_out (1.7, 2.1, 1.9)`	*В среднем 1,9 минуты*
7.	`test_edit_pic (2.1, 2.6, 2.2)`	*В среднем 2,3 минуты*
8.	`test_post_forum (1.8, 1.9, 1.7)`	*В среднем 1,8 минуты*
9.	`test_edit_forum (1.6, 1.5, 1.7)`	*В среднем 1,6 минуты*
10.	`test_share_twitter (2.1, 1.9, 2.0)`	*В среднем 2,0 минуты*
11.	`test_share_instagram (2.0, 1.9, 2.1)`	*В среднем 2,0 минуты*
12.	`test_report_userOur (1.3, 1.2, 1.1)`	*В среднем 1,2 минуты*

Чтобы организовать шардинг для следующего запуска, следует проанализировать данные о среднем времени выполнения тестов и сформировать группы таким образом, чтобы каждый из трех шардов выполнял тесты примерно за одно и то же время.

Шард 0: 7,4 минуты

```
test_edit_pic (2.3)
test_share_instagram (2.0)
test_post_forum (1.8)
test_report_user (1.2)
test_follow_dog (0.8)
```

Шард 1: 8,1 минуты

```
test_post_pic (3.1)
test_view_leaderboard (1.9)
test_login (1.6)
test_rate_pic (0.8)
```

Шард 2: 7,5 минуты

```
test_browse_pics (2.0)
test_share_twitter (2.0)
test_view_logged_out: (1.9)
test_edit_forum: (1.6)
```

Мы не будем подробно разбирать этот алгоритм (хотя это действительно интересный и отличный практический вопрос для собеседования!). Если вам нужен такой шардинг, возможно, вам придется создать его самостоятельно, однако ваша система непрерывной доставки (или инструменты языка программирования) может сделать это за вас. Например, такая возможность предусмотрена в системе CircleCI с помощью передачи имен тестов в команду разделения, которая не привязана к языку:

```
circleci tests split --split-by=timings
```

Шардированный пайплайн

Все этапы шардинга можно выполнять в рамках одной задачи пайплайна или, если это позволяет система непрерывной доставки, разделить задачу на несколько частей.

Если вы осуществляете простой шардинг тестов по индексу, вам, вероятно, не понадобится этот шаг. Но для более сложных задач, например для распределения тестов по времени их выполнения, он необходим.

Чтобы разбить этот этап на несколько задач, система непрерывной доставки должна поддерживать итерацию процедур в пайплайнах. Если она не предусматривает такой возможности, объедините этот этап в одну задачу.

Чтобы поддерживать шардинг, набор тестов должен удовлетворять следующим требованиям:

- Тесты не должны зависеть друг от друга.
- Тесты не должны мешать друг другу. Если тесты используют общие ресурсы (например, все подключаются к одному и тому же экземпляру зависимости), они будут конфликтовать друг с другом (а может, и нет; самый простой способ узнать это — попробовать).
- Если вы распределяете тесты по индексам, тесты должны запускаться в определенном порядке, чтобы определенные тесты на всех шардах имели одинаковые индексы.

Если параллельный запуск юнит-тестов — всего лишь запах кода, то распределение их по шардам — это просто кошмар!

Как уже отмечалось, если юнит-тесты настолько медленные, что их требуется запускать параллельно, чтобы время их выполнения оставалось приемлемым, это признак того, что с ними что-то не так (возможно, они делают слишком много). Если они *настолько медленные*, что вам требуется шардинг, то с вероятностью 99,99999% это не юнит-тесты, и не помешает добавить их в код вместо замаскированных интеграционных/системных тестов. (Предлагаю впредь называть действительно плохой код *вонючим кодом*.)

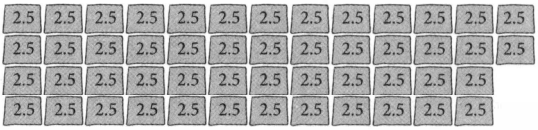
Шардинг браузерных тестов

Шридхар собирается решить проблему медленных браузерных тестов путем применения шардинга! Вот основная цель, к которой он стремится:

Весь набор тестов должен выполняться в среднем за 30 минут или менее.

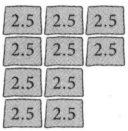
Интеграционные и юнит-тесты занимают в среднем 5 минут, поэтому ему нужно добиться, чтобы браузерные тесты выполнялись примерно за 25 минут.

На один тест в среднем уходит 2,5 минуты, а всего их 50. Время выполнения каждого теста практически одинаковое, поэтому Шридхар решает пойти простым путем и использовать шардинг по индексам. Сколько же шардов ему понадобится?

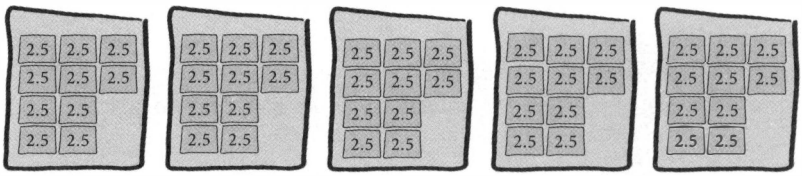


Поскольку цель — завершить все тесты за 25 минут, значит, на каждом шарде тесты могут выполняться до 25 минут. Сколько браузерных тестов можно выполнить за 25 минут?

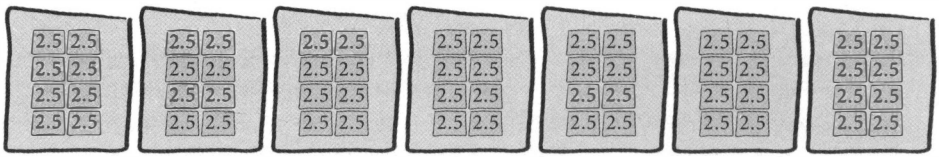
Если каждый из них занимает в среднем 2,5 минуты, то 25 минут / 2,5 минуты = 10. За 25 минут на одном шарде можно выполнить 10 тестов.



Если всего тестов 50 и на каждом шарде можно выполнить 10 тестов за 25 минут, то ему нужно 50 / 10 = 5 шардов.



Пяти шардов будет достаточно для достижения цели, но Шридхар знает, что в компании нет проблем с оборудованием, поэтому он щедро выделяет на браузерные тесты целых 7 шардов.

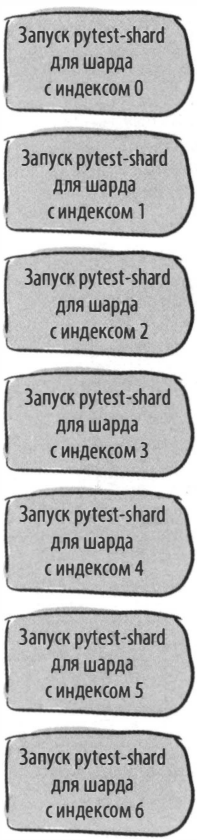


При использовании 7 шардов каждый из них должен будет выполнить 50/7 тестов; наибольшее количество тестов будет — 50/7 = 8 тестов. Восемь тестов, при средней продолжительности каждого 2,5 минуты, будут выполнены за 20 минут. Это позволит Шридхару немного превзойти целевое значение в 25 минут, и разработчики смогут свободнее добавлять новые тесты, прежде чем потребуется увеличить количество шардов.

Шардинг в пайплайне

Для браузерных тестов будет работать простой шардинг на основе индексов, поэтому Шридхару остается только добавить параллельно выполняемые задачи, по одной для каждого шарда, и заставить их использовать pytest-shard для запуска подмножества браузерных тестов. Его распараллеленные задачи для браузерных тестов будут запускать следующий скрипт на Python, используя Python для вызова pytest:

```
# когда пайплайн будет запущен, он передаст этому скрипту
# индекс шарда и общее количество шардов
# в качестве аргументов
shard_index, num_shards, path_to_tests = get_arguments()

# мы будем инициировать pytest в качестве команды для запуска
# соответствующего набора тестов для данного шарда
run_command(
  "pytest --shard-id={} --num-shards={} {}".format(
    shard_index, num_shards, path_to_tests
))
```



Чтобы включить этот скрипт в пайплайн, Шридхару необходимо только добавить набор задач, которые выполняются параллельно, — в его случае семь, по одной для каждого из семи шардов. Нужно ли ему для этого жестко прописывать семь отдельных задач в пайплайне? Это зависит от особенностей системы непрерывной доставки. Большинство систем предоставляют возможность распараллелить задачи. Для этого вам потребуется указать, сколько экземпляров задачи вы собираетесь запустить, а затем система передаст в качестве аргументов (часто переменных среды) информацию запущенным задачам о том, сколько всего экземпляров запущено и в каком они экземпляре. Например, в GitHub Actions, чтобы запускать одно и то же задание несколько раз, можно использовать стратегию матрицы:

Для обозначения того, что в этой книге называется «задачами» (tasks), в GitHub Actions используется термин «задания» (jobs).

```
jobs:
  tests:
    strategy:
      fail-fast: false
      matrix:
        total_shards: [7]
        shard_indexes: [0, 1, 2, 3, 4, 5, 6]
```

Это необходимо, чтобы гарантировать, что если на одном шарде произойдет остановка, то остальные все равно завершат работу.

При такой конфигурации задание `tests` будет запущено семь раз, а шагам в каждом задании можно передать информацию об общем количестве шардов и о том, в качестве какого шарда они выполняются, в следующих контекстных переменных:

```
${{ matrix.total_shards }}
${{ matrix.shard_indexes }}
```

Эти имена переменных для matrix произвольны; подробнее см. jobs.<job_id>.strategy.matrix на GitHub Actions.

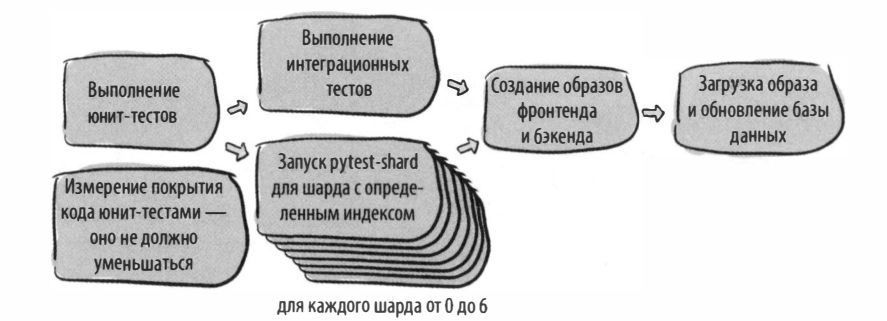
Пайплайны сайта Dog Picture

Шридхар достиг своей цели — выполнять браузерные тесты за 25 минут (по факту за 20 минут), теперь он может объединить все тесты вместе, и весь набор будет выполняться в среднем за 30 минут или даже меньшее время. Это значит, что он может перейти к последней задаче:

Запускать тесты при каждом изменении до того, как оно будет отправлено.

Шридхар добавляет браузерные тесты в пайплайн перед слиянием и делает так, чтобы они запускались параллельно с интеграционными тестами. Теперь пайплайн может запускать все тесты до слияния, и продолжительность их выполнения будет равна длительности шардированных браузерных тестов (20 минут) плюс время на выполнение юнит-тестов (менее 1 минуты).

Шридхар аналогично обновляет пайплайн ночного релиза, чтобы он также заработал быстрее:



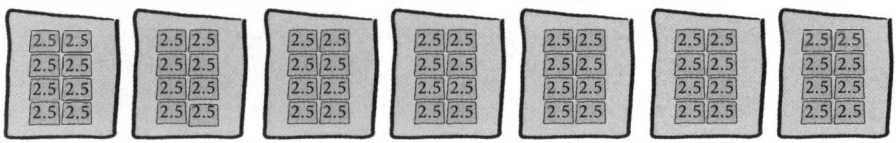
ВАЖНО

Параллельный запуск тестов увеличит нагрузку на оборудование, но сэкономит бесценный ресурс: время! Если тесты выполняются медленно, сначала оптимизируйте распределение тестов, работая с юнит-тестами; затем при необходимости можно воспользоваться распараллеливанием и шардингом.

Пища для ума

Чтобы выполнить 50 тестов за 25 минут и меньше, Шридхару нужно 5 шардов, но он добавляет еще 2, чтобы получить в общей сложности 7, и тем самым ускоряет время выполнения тестов и обеспечивает резерв для будущих тестов. *Но что, если количество тестов продолжает расти — означает ли это, что придется добавлять все больше и больше шардов?* И сработает ли это?

Когда количество тестов браузера увеличится с 50 до 70, на каждом из 7 шардов будет выполняться по 10 тестов, а общее время выполнения составит 25 минут.



Если добавить еще несколько тестов, то время выполнения браузерных тестов превысит 25 минут и потребуется добавить еще несколько шардов. Означает ли это, что придется продолжать добавлять шарды до бесконечности? Не окажется ли их в итоге слишком много?

Такое вполне может произойти; вы, наверное, помните, что архитектура сайта Dog Picture довольно монолитна:

Если функционал сайта Dog Picture продолжит расти, то компании придется разделить функции бэкенд-сервиса на отдельные службы, и каждая из них будет иметь собственные наборы тестов.

Это означает, что когда что-то нужно будет изменить, разработчики смогут запускать только те тесты, которые связаны с этим изменением, а не абсолютно все. Подобное разделение функций, вероятно, потребуется и для того, чтобы соответствовать динамике роста компании (по мере увеличения числа сотрудников их необходимо будет разделить на эффективные команды с отдельными сферами ответственности).

Информация к размышлению: перенесемся в будущее, когда сайт Dog Picture будет состоять из множества сервисов, каждый со своим набором сквозных тестов. Достаточно ли запускать каждый набор в отдельности, чтобы проверить работоспособность всей системы? Нужно ли в этих целях проводить все тесты вместе перед релизом? Все зависит от конкретной ситуации; но помните, что ни в чем нельзя быть уверенным на сто процентов. Главное — найти компромиссные решения, которые подойдут для вашего проекта.

Ваш ход: как ускорить тесты

У сайтов Dog Picture и Cat Picture есть общий конкурент: набирающий обороты сайт Bird Picture. Он столкнулся с аналогичной проблемой медленных тестов, но немного в другой ситуации.

Весь набор тестов команда Bird Picture выполняет примерно за 3 часа, но в отличие от команды Dog Picture, она выполняет его для каждого PR. Когда разработчики готовы внести изменения, они создают PR, а затем оставляют его открытым, часто до следующего дня, дожидаясь выполнения всех тестов. Такой подход позволяет выявить проблемы до слияния, но на внесение изменений часто уходят дни (иногда это называют *борьбой с тестами*).

Набор тестов для сайта Bird Picture распределяется следующим образом:

- 10 % юнит-тестов;
- 0 % интеграционных тестов;
- 90 % сквозных тестов.

Покрытие кода юнит-тестами составляет 34 %, и на их выполнение уходит 20 минут. Что необходимо сделать команде Bird Picture, чтобы ускорить тесты?

Ответы

При анализе тестов сайта Bird Picture сразу бросаются в глаза несколько моментов:

- Так называемые юнит-тесты довольно медленные для такого рода тестов; в идеале они должны выполняться максимум за пару минут, если не за секунды. Скорее всего, это интеграционные тесты.
- Покрытие юнит-тестами (или, возможно, интеграционными тестами) довольно низкое.
- Сквозных тестов по сравнению с юнит-тестами *слишком много*; может быть, тестов просто самих по себе мало, но есть и вероятность, что команда Bird Picture слишком увлеклась сквозными тестами.

Исходя из этого, команда Bird Picture может сделать следующее:

- Разобраться с медленными юнит-тестами; если какие-то из них действительно представляют собой модульные тесты (то есть выполняются за несколько секунд или менее), запускать их отдельно от других медленных тестов (которые на самом деле являются интеграционными) и в первую очередь, чтобы получать от них быстрый сигнал.
- Измерить покрытие кода быстрыми юнит-тестами; оно будет еще ниже, чем и без того низкое покрытие в 34 %. Сопоставить непокрытые области с огромным набором сквозных тестов и определить, какие сквозные тесты можно заменить юнит-тестами.
- Ввести задачу измерения и отчета о покрытии юнит-тестами при каждом PR и не производить слияние для PR, которые уменьшают покрытие юнит-тестами.
- После этого вернуться к распределению тестов и решить, что делать дальше. Скорее всего, многие сквозные тесты можно преобразовать в интеграционные; вполне вероятно, что для желаемого результата будет достаточно запуска не всей системы, а лишь пары ее компонентов; вдобавок наверняка это будет быстрее.

Заключение

Со временем на выполнение набора тестов для сайта Dog Picture уходило все больше и больше времени. И вместо того, чтобы искать способы ускорить тестирование, разработчики просто перестали заниматься тестами, по сути, откладывая решение проблемы в долгий ящик. Хотя вначале это и помогло им ускорить работу, затем это стало сильно ее тормозить. Шридхар понимал, что единственный выход — проанализировать набор тестов и максимально оптимизировать его. А когда возможности оптимизации оказались исчерпаны, он применил распараллеливание и шардинг. Эти инструменты помогли ускорить тестирование достаточно, чтобы вновь проводить его перед слиянием, и разработчики смогли быстрее получать обратную связь.

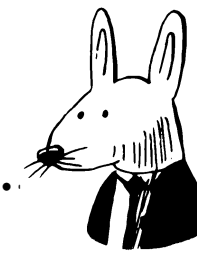
Итоги

- При работе с медленным набором тестов добиться эффекта в короткие сроки помогает независимый запуск в первую очередь самых быстрых тестов.
- Прежде чем пытаться сокращать время выполнения медленных тестов с помощью различных технологий, сначала критически оцените сами тесты. Использование пирамиды тестов поможет выделить главное, а контроль покрытия кода тестами — обеспечить надежную базу юнит-тестов.
- Тесты могут быть очень надежными, но чересчур медленными. В этом случае для ускорения тестирования можно использовать распараллеливание и шардинг, задействовав больше ресурсов оборудования, но получив выигрыш во времени.

Далее...

В следующей главе мы продолжим тему своевременного получения сигналов и рассмотрим, на каком этапе могут появляться баги. Вы узнаете, какие процессы необходимы, чтобы как можно раньше получать сигналы о том, что что-то пошло не так.

Нужные сигналы в нужное время | 7



В этой главе

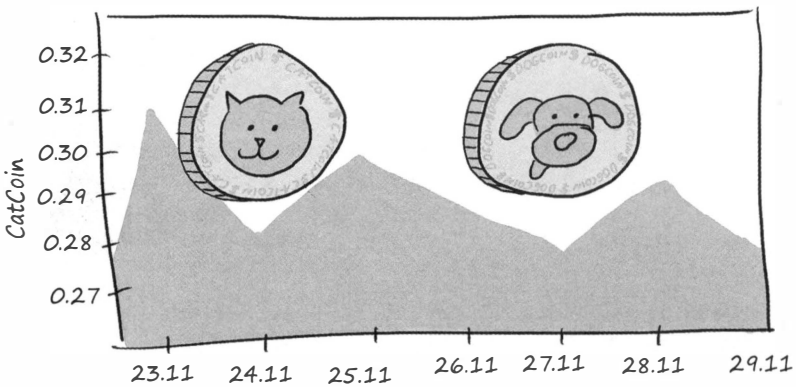
- ✓ Точки жизненного цикла изменений, в которых могут возникнуть баги
- ✓ Как гарантировать, что конфликт изменений не приведет к появлению багов
- ✓ Плюсы и минусы методов устранения конфликтов
- ✓ Как выявлять баги на всех этапах жизненного цикла изменения с помощью непрерывной интеграции, проводимой до и после слияния, а также периодически

В предыдущих главах подробно рассматривались пайплайны непрерывной интеграции (CI), работающие на разных этапах жизненного цикла изменений. Например, если пайплайны запускаются после внесения изменений, необходимо соблюдать важное правило: *прекратить слияние, если происходит сбой пайплайна*. Также мы рассмотрели случаи, когда линтинг и тесты запускаются до слияния изменений, — это оптимальный вариант, позволяющий избежать поломок кода.

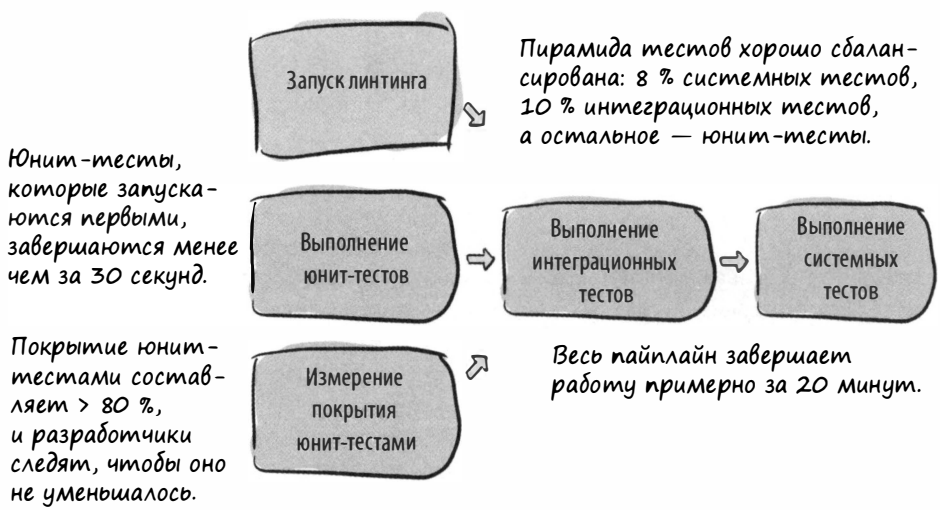
В этой главе мы разберем жизненный цикл изменений. Вы узнаете, где вообще могут появиться баги, а также когда запускать пайплайн, чтобы получать сигналы о багах и исправлять их как можно быстрее.

Веб-сайт CoinExCompare

CoinExCompare — сайт, на котором публикуются курсы обмена цифровых валют. Пользователи могут зайти на сайт и сравнить курсы обмена, например, таких валют, как CatCoin и DogCoin.



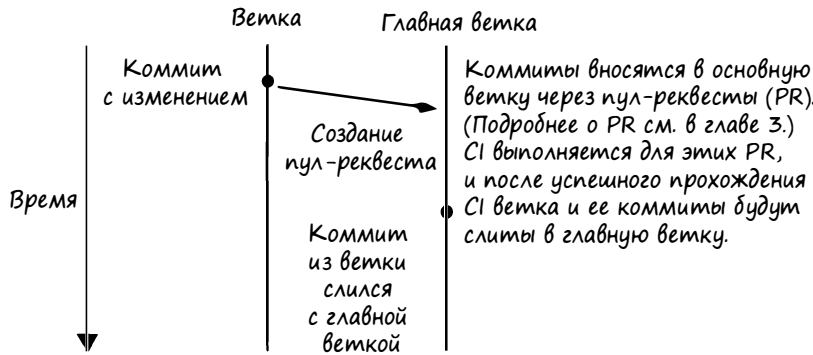
Компания быстро развивалась, но в последнее время у нее участились баги и сбои в работе. Разработчики недоумевают, потому что они внимательно изучили пайплайны и считают, что обеспечили качественное покрытие кода тестами.



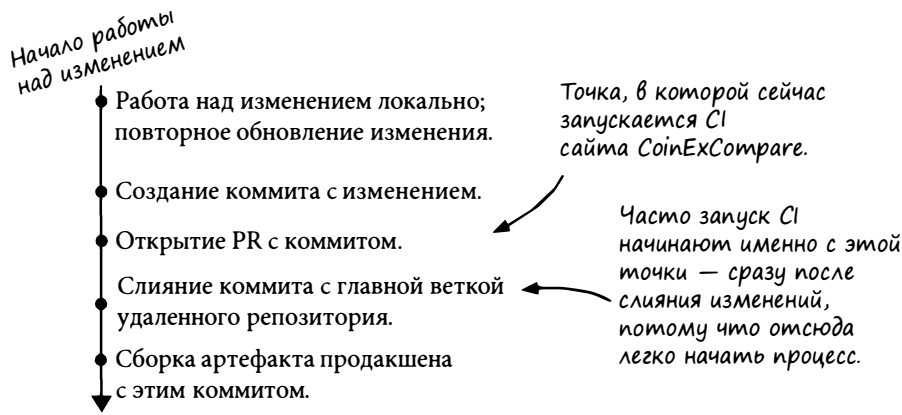
Пайплайн CI отличный, и что вообще может пойти не так?

Жизненный цикл изменения

Чтобы понять, где в работе сайта CoinExCompare могут возникнуть баги, разработчики составляют схему жизненного цикла изменений. Они используют магистральную разработку (Trunk-Based Development) с ветками и пул-реквестами, имеющими непродолжительное время жизни (подробнее об этом в главе 8):



Жизненный цикл изменения в самом коммите с течением времени выглядит следующим образом:



Словарик

Термин «*продакшен*» (production) используется для обозначения среды, в которой программный продукт становится доступным для пользователей. Если вы запускаете сервис, то конечная точка (точки), доступная вашим клиентам, может называться *продакшен*. Артефакты, такие как образы и двоичные файлы, которые работают в этой среде (или рассылаются непосредственно пользователям), могут называться *рабочими артефактами* или *артефактами продакшена* (например, *рабочими образами*). Этим термином обозначаются среды, отличные от временных сред (например, *промежуточной среды — предпродакшен, или стейджа* — stage), либо артефакты, отличные от тех, которые могут использоваться для проверки или тестирования, но никогда напрямую не передаются пользователям.

Проводите непрерывную интеграцию только перед слиянием

Если у вас полностью отсутствует автоматизация, проще всего запустить CI сразу после слияния изменений.

Пример вы видели в главе 2, когда Тофер настроил автоматизацию веб-хуков для сайта Cat Picture так, чтобы тесты запускались при каждом внесении изменений. Это быстро привело к тому, что команда усвоила важное правило:

Не отправлять изменения, когда произошел сбой пайплайна.

Это по-прежнему отличный вариант старта и самый простой способ подключить автоматизацию, особенно если вы используете ПО для контроля версий, которое не содержит дополнительных функций автоматизации, и вам нужно создавать их самостоятельно (как это делал Тофер в главе 2). Однако у него есть и недостатки:

Действительно ли проще всего начать с запуска CI после слияния изменений, зависит от того, какие инструменты вы уже используете. Некоторые из них, например GitHub, позволяют очень легко организовать CI на основе PR и, как вы увидите в этой главе, получать сигналы раньше.

- О проблемах станет известно только *после* того, как они уже будут в коде. Поэтому код может оказаться в состоянии, небезопасном для выпуска релиза, а одно из условий непрерывной доставки (CD) — *возможность вносить изменения в программный продукт безопасно и в любое время*. Если код будет регулярно ломаться, это условие выполняться не будет.
- Требование остановить внесение изменений при сбое CI тормозит всех разработчиков, что в лучшем случае создает дискомфорт, а в худшем — приводит к убыткам.

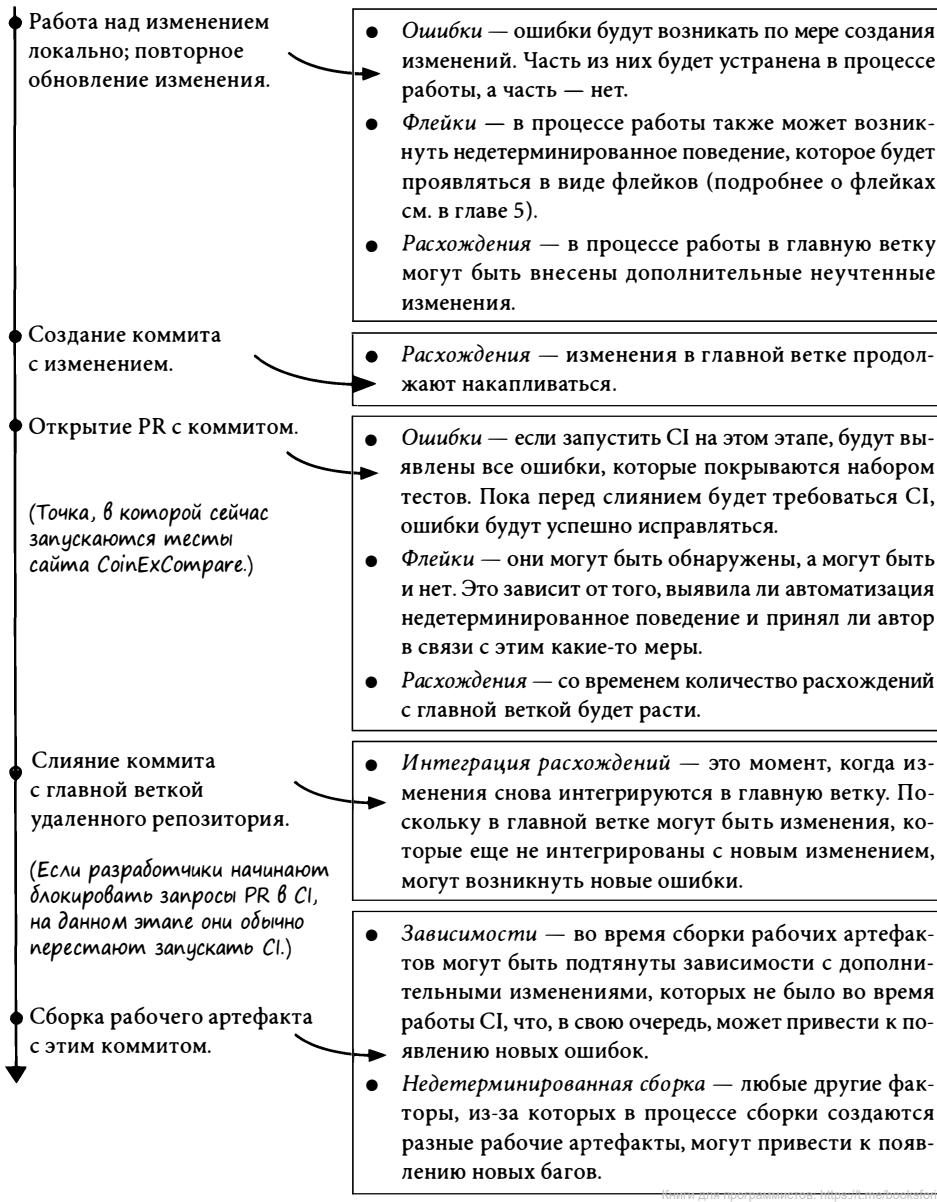
Именно в таком положении оказалась команда CoinExCompare около шести месяцев назад. Но компания решила вложиться в автоматизацию, которая позволила бы запускать CI перед слиянием, чтобы разработчики могли предотвратить поломки кода. Это позволило нивелировать оба недостатка запуска CI после слияния изменений:

- Вместо того чтобы узнавать об ошибках после того, как они уже появились в коде, можно вообще предотвратить их добавление.
- Можно избежать остановки всего процесса при внесении некорректного изменения и поручить автору изменения разобраться с проблемой. После исправления автор сможет произвести слияние изменений.

Именно так сейчас работает сайт CoinExCompare: члены команды запускают CI перед слиянием изменений и не производят слияние, пока CI не пройдет.

Последовательность появления багов при внесении изменений

Процесс разработки CoinExCompare подразумевает прохождение CI перед слиянием изменений, но баги в рабочей версии никуда не делись. Как такое может быть? Чтобы понять это, рассмотрим, в каких местах могут появиться баги в процессе внесения изменений, то есть в местах, в которых необходим сигнал о том, что что-то идет не так:

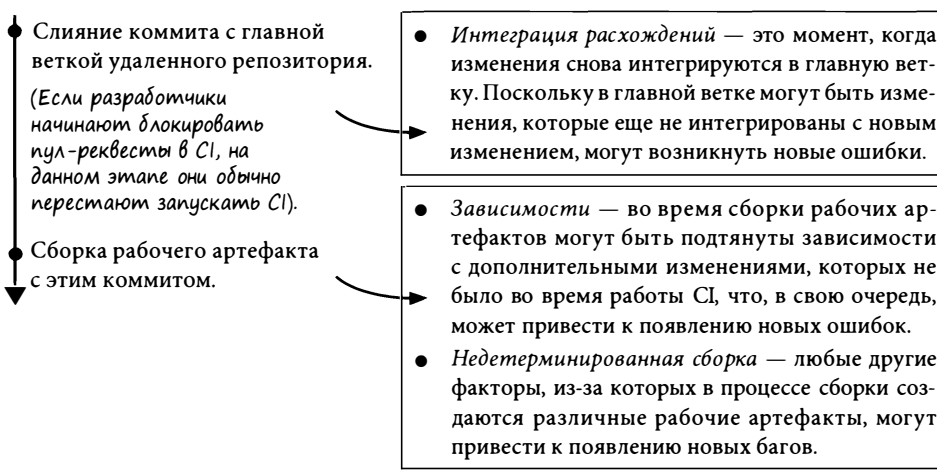


CI, проводимая только перед слиянием, пропускает ошибки

Пока что система сайта CoinExCompare блокирует процесс слияния PR на этапе прохождения CI, но сейчас CI запускается *всего один раз*. А как выясняется, баги могут закрасться еще в нескольких местах после ее прохождения:

- *Расхождение с главной веткой* — если CI запускается только перед тем, как изменение интегрируется обратно в главную ветку, в главной ветке могут оказаться изменения, которые новое изменение не учитывает и для проверки которых CI вообще никогда не выполнялась.
- *Изменения в зависимостях* — для работы большинства артефактов требуются пакеты и библиотеки, отсутствующие в их коде. При сборке рабочих артефактов будет задействована одна из версий этих зависимостей. Если это будет версия, отличная от той, с которой выполнялась CI, могут появиться новые баги.
- *Недетерминированность* — проявляется как в виде пропущенных флейков, так и в виде неочевидных различий между очередной сборкой артефакта и его последующими сборками, что может привести к появлению багов.

Если взглянуть на временну́ю шкалу внесения изменений, то можно увидеть, что эти три источника багов могут появиться даже после того, как CI на основе PR будет успешно пройдена:



Правильные сигналы

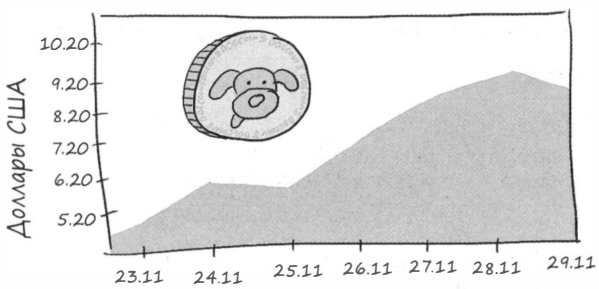
Для каждой точки, где могут возникнуть баги, желательно настроить пайплайн CI таким образом, чтобы получать сигнал как можно раньше — в идеале непосредственно перед возникновением проблемы. Если вы получите сигнал о том, что что-то пошло не так или вот-вот пойдет не так, вы сможете оперативно вмешаться и исправить ошибку. Подробнее о сигналах читайте в главе 5.

История двух графиков: установка значения по умолчанию «семь дней»

Посмотрим, как CoinExCompare работает с каждым из этих источников багов. Недавно команда столкнулась с технологической ошибкой, которая была вызвана первым из них:

Расхождение с главной веткой

Ния работает над функцией построения графика динамики курса за последние 7 дней для конкретной монеты. Например, если пользователь перейдет на целевую страницу DogCoin, он увидит подобный график, показывающий курс закрытия для монеты в долларах США в каждый из последних 7 дней.



В процессе работы над этой функциональностью Ния обнаруживает функцию, которая уже существует и, похоже, может заметно облегчить ее работу. Функция `get_daily_rates` возвращает пиковые ежедневные значения курса для определенной монеты (к доллару США) за некоторый промежуток времени. По умолчанию функция возвращает данные за все время, на что указывает соответствующее значение параметра, равное `0` (то есть переменной `MAX`):

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
  rate_hub = get_rate_hub(coin)
  rates = rate_hub.get_rates(num_days)
  return rates
```

Просматривая код, Ния с удивлением обнаруживает, что ни одна вызывающая данную функцию процедура не использует по умолчанию значение параметра `num_days`, равное `MAX`. Поскольку ей приходится вызывать эту функцию несколько раз, она решает, что логично установить значение по умолчанию, равное 7 дням, и это обеспечит ей требуемую функциональность, поэтому она меняет функцию так, чтобы значение по умолчанию равнялось 7 дням, а не `MAX`, и добавляет для проверки юнит-тест:

```
def get_daily_rates(coin, num_days=7):
  rate_hub = get_rate_hub(coin)
  rates = rate_hub.get_rates(num_days)
  return rates
...
  def test_get_daily_rates_default(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0])
```

Все тесты, включая новый, проходят, так что она спокойно создает PR для изменения.

История двух графиков: установка значения по умолчанию «30 дней»

Но Ния не подозревает, что кто-то еще вносит изменения в тот же код! Ее коллега по CoinExCompare Цзыхао работает над графиком для другой страницы. Его функция должна отображать данные за последние 30 дней для конкретной монеты.

К сожалению, ни Ния, ни Цзыхао не понимают, что над похожим алгоритмом работает не один человек! А великие умы думают одинаково: Цзыхао замечает ту же функцию, что и Ния, и надеется, что она поможет ему добиться нужного результата:

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
  rate_hub = get_rate_hub(coin)
  rates = rate_hub.get_rates(num_days)
  return rates
```

Вы только что видели, как Ния изменила эту функцию, но она еще не осуществила слияние изменений, поэтому Цзыхао о них не знает.

Цзыхао проводит тот же анализ, что и Ния, и замечает, что никто не использует эту функцию с аргументами по умолчанию. Поскольку ему приходится вызывать ее несколько раз, он считает логичным изменить принятые по умолчанию аргументы функции так, чтобы она возвращала показатели за последние 30 дней, а не за все время. Однако он делает это немного иначе, чем Ния:

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
  rate_hub = get_rate_hub(coin)
  rates = rate_hub.get_rates(30 if num_days==MAX else num_days)
  return rates
```

Цзыхао также добавляет юнит-тест для своих изменений:

```
  def test_get_daily_rates_default_thirty_day(self):
    rates = get_daily_rates(«catcoin»)
    self.assertEqual(rates, [2.0]*30)
```

Кто изменил функцию верно?

Ния изменила сам аргумент по умолчанию, а Цзыхао оставил значение параметра по умолчанию прежним, но изменил место использования переменной `MAX`. Изменение Нии лучше: в версии Цзыхао параметр по умолчанию дважды устанавливается на два разных значения, — не говоря о том, что аргумент MAX больше не работает, потому что, даже если кто-то укажет его явно, алгоритм все равно вернет значение 30 дней. Это как раз тот случай, который желательно обнаружить при проверке кода. На самом деле этот пример немного притянут за уши, чтобы показать, что происходит, когда вносятся конфликтующие изменения, которые система контроля версий не в состоянии отловить.

И Ния, и Цзыхао изменили одну и ту же функцию, чтобы она работала по-разному, и уверены, что все сделали правильно. Ния рассчитывает, что функция по умолчанию будет возвращать данные за 7 дней, а Цзыхао — что она будет возвращать данные за 30 дней.

Конфликтующие изменения не всегда удается отловить

Ния и Цзыхао оба изменили логику установки значения параметра по умолчанию в одной и той же функции, но ведь когда придет время слияния, эти конфликтующие изменения будут обнаружены, правда?

К сожалению, нет! В большинстве систем контроля версий логика поиска конфликтов слишком проста и не учитывает фактическую семантику внесенных изменений. При слиянии изменений, если изменяются совершенно одинаковые строки, система контроля версий поймет, что что-то не так, но не сможет пойти дальше этого.

Ния и Цзыхао изменили разные строки в функции `get_daily_rates`, поэтому слияние изменения может произойти без конфликтов! Цзыхао сливает свои изменения первым, изменяя состояние `get_daily_rates` в главной ветке для реализации нового алгоритма установки значения по умолчанию:

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
  rate_hub = get_rate_hub(coin)
  rates = rate_hub.get_rates(30 if num_days==MAX else num_days)
  return rates
```

Одновременно с этим Ния сливает свои изменения. Изменения Цзыхао уже находятся в главной ветке, поэтому ее изменения в строке, расположенной двумя строками выше изменений Цзыхао, также успешно сливаются, в результате чего получается следующая функция:

```
MAX=0

def get_daily_rates(coin, num_days=7):
  rate_hub = get_rate_hub(coin)
  rates = rate_hub.get_rates(30 if num_days==MAX else num_days)
  return rates
```

Изменение Нии устанавливает значение по умолчанию для аргумента.

Между тем Цзыхао полагался на то, что аргумент по умолчанию будет равен MAX.

В итоге функция Цзыхао для построения графика сливается первой и работает просто отлично до слияния изменений, внесенных Нией, возвращающих прежнюю функцию. Изменения Нии нарушают работу функции Цзыхао: теперь, когда значе-

ние по умолчанию равно 7, а не MAX, тройное условие Цзыхао всегда будет ложным (если только кто-нибудь, вызывающий эту функцию, не вздумает явно передать значение MAX), и таким образом функция по умолчанию будет возвращать данные только за 7 дней. Это означает, что функция Нии будет выполняться как положено, а функция Цзыхао теперь не работает.

> ***Такое действительно может произойти?***
>
> Конечно да! Этот пример немного надуман, поскольку более очевидным решением для Цзыхао было бы также изменить значение аргумента по умолчанию, и конфликт стал бы очевиден. Более реалистичный и частый сценарий включает изменения нескольких файлов, например, внесение изменений, зависящих от конкретной функции, в то время как кто-то другой меняет саму эту функцию.

А как же юнит-тесты?

Ния и Цзыхао добавили юнит-тесты. Это значит, что конфликтующие изменения будут обнаружены?

Если бы они оба добавили тесты в одну точку файла, система контроля версий распознала бы это как конфликт, поскольку они оба изменяли бы одни и те же строки. К сожалению, в нашем примере тесты были добавлены в разных местах файла, поэтому конфликт не был выявлен. Конечным результатом слияния будет наличие обоих юнит-тестов:

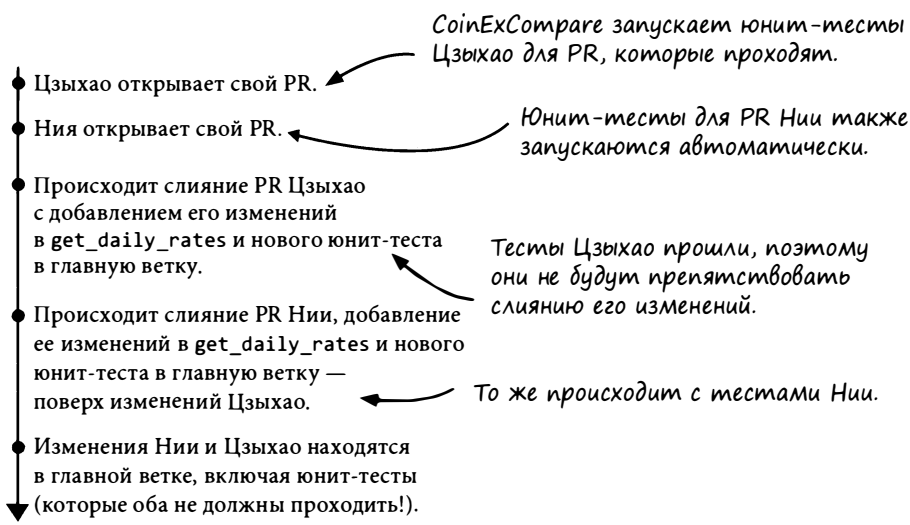
```
  def test_get_daily_rates_default(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0])
...
  def test_get_daily_rates_default_thirty_day(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0]*30)
```

В юнит-тесте Нии ожидается, что функция по умолчанию вернет данные за 7 дней.

Юнит-тест Цзыхао предполагает, что точно такая же функция, вызванная с точно такими же аргументами, вернет данные за 30 дней.

Система контроля версий не смогла обнаружить конфликт, но ведь не может же быть так, чтобы оба теста прошли? Значит, тесты обязательно обнаружат проблему?

И да и нет! Если оба теста будут запущены одновременно, один из них провалится (оба пройти просто не могут, если только не произойдет что-то нестандартное). Но будут ли оба теста выполняться вместе? Рассмотрим временную шкалу происходящего с изменениями Нии и Цзыхао и узнаем, в какой момент будут запущены тесты:

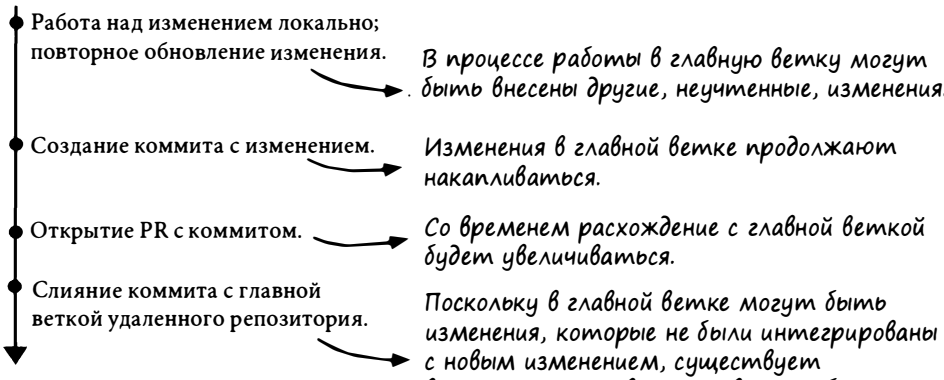


Тесты запускаются автоматически для каждого PR в отдельности. В системе CoinExCompare предусмотрен только запуск CI (включая тесты) для каждого PR, без автоматической непрерывной интеграции объединенных изменений после их слияния.

Запуск на основе PR не защищает систему от багов

Запуск CI на основе PR-запроса — отличный способ обнаружить баги до того, как они появятся в главной ветке. Но как вы видели на примере Нии и Цзыхао, чем дольше изменения находятся в отдельной ветке и не интегрируются в главную, тем больше вероятность того, что будет внесено конфликтующее изменение, которое может привести к непредсказуемым ошибкам.

Еще один способ снизить такого рода риски — как можно быстрее интегрировать изменения в главную ветку. Подробнее об этом в следующей главе.



Я регулярно вытягиваю изменения из главной ветки; разве это не решает проблему?

Это, конечно, снижает вероятность пропустить конфликтующие изменения в главной ветке. Однако пока вы не гарантируете, что последние изменения подтянуты, что CI запускается непосредственно перед слиянием и что за это время не будут внесены другие изменения, сохраняется риск что-то пропустить, если осуществляется только CI на основе PR.

ВАЖНО

Запуск CI на PR перед слиянием не гарантирует, что будут выявлены все конфликтующие изменения. Если они затронули одни и те же строки, система контроля версий может обнаружить конфликт и заставить выполнить обновление (и повторный запуск CI) перед слиянием, но если изменения внесены в разные строки или в разные файлы, может возникнуть ситуация, когда CI перед слиянием прошла успешно, но после слияния главная ветка не работает.

Непрерывная интеграция до И после слияния

Что может сделать команда CoinExCompare, чтобы получать сигналы о том, что существуют конфликты, и избежать проблем с главной веткой? И Ния, и Цзыхао добавили тесты, покрывающие их функциональность. Если бы эти тесты были запущены после объединения изменений (после слияния), проблема была бы обнаружена сразу. CoinExCompare ставит новую цель:

> **Объединять изменения с последней версией главной ветки, а затем запускать для них CI перед слиянием.**

Что нужно сделать, чтобы достичь этой цели? У команды CoinExCompare есть несколько вариантов:

1. Периодически запускать CI на главной ветке.
2. Требовать, чтобы все ветки обновлялись до слияния с главной.
3. Использовать автоматизацию для слияния изменений с главной веткой и повторного запуска CI перед слиянием (то есть использовать *очередь слияния*).

Далее я более подробно разберу все три варианта, а пока кратко перечислю преимущества и недостатки каждого из них:

- Вариант 1 будет отлавливать ошибки, но только после того, как они будут внесены в главную ветку, а значит, риск сломать главную ветку остается.
- Вариант 2 предотвратит появление такого рода ошибок, и кроме того, он поддерживается некоторыми системами контроля версий (например, GitHub). Но на практике он может стать огромной головной болью.
- Вариант 3, при правильной реализации, также может предотвратить появление этих ошибок. Как встроенная функция, он работает очень хорошо, но его довольно сложно реализовывать и поддерживать самостоятельно.

Вариант 1: периодический запуск CI

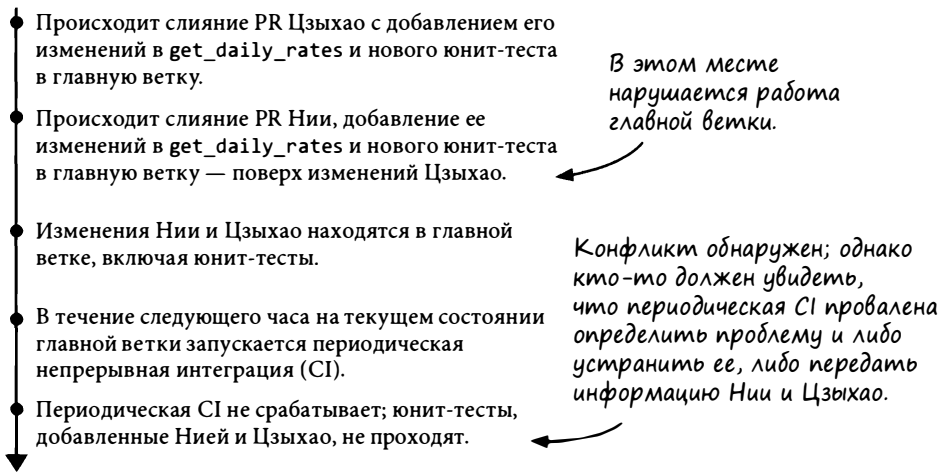
Рассмотрим подробнее первый вариант. В случае с Нией и Цзыхао неприятно то, что проблема не была выявлена до тех пор, пока не попала на продакшен, хотя юнит-тесты могли ее обнаружить!

В этом варианте сделан упор на простоту обнаружения нештатного случая, а не на предотвращение его возникновения. Но подобные баги, вызванные взаимодействием нескольких изменений, случаются не очень часто.

Простой способ обнаружить такого рода проблемы — *периодически* запускать CI для главной ветки в дополнение к запуску на основе пул-реквеста. Это может быть запуск CI по ночам или чаще (например, каждый час), если задачи выполняются достаточно быстро. Конечно, у этого способа есть недостатки:

- Такой подход не защищает главную ветку от поломок.
- Кому-то придется следить за периодическими тестами или, по крайней мере, принимать меры в случае их сбоя.

Если CoinExCompare решит использовать периодическую CI в качестве решения проблемы, как это будет выглядеть с точки зрения Нии и Цзыхао? Допустим, команда CoinExCompare решила запускать тесты каждый час:



По крайней мере, теперь проблема будет обнаружена и может быть устранена до того, как попадет в продакшен, но достигают ли при этом разработчики CoinExCompare своей цели?

Объединять изменения с последней версией главной ветки, а затем запускать для них CI перед слиянием.

Поскольку CI происходит после слияния, вариант 1 не соответствует поставленной цели.

Вариант 1: настройка периодической CI

Команда CoinExCompare решила не переходить на периодическую CI (пока), но прежде чем приступить к рассмотрению других вариантов, обсудим вкратце, что нужно для ее настройки. До сих пор сайт CoinExCompare использовал GitHub Actions, поэтому внести изменения не составит труда. Допустим, разработчики собираются запускать пайплайн каждый час. В рабочем процессе GitHub Actions они могут использовать синтаксис расписания (schedule), включив директиву `schedule` в раздел активации `on`:

У периодической CI есть и другие преимущества, которые мы рассмотрим в этой главе чуть позже. Читайте дальше!

```
on:
  schedule:
    - cron: ´´0 * * * *´
```

Директива schedule в системе GitHub Actions использует синтаксис вкладки cron для указания времени запуска.

Хотя настроить периодический (вернее, «по расписанию») запуск несложно, сложно определить, что же делать с результатами. При запуске CI на основе PR, вопроса, кто должен предпринимать какие-то действия в случае ошибки, не возникает — решать проблему должен автор самого PR. И это в первую очередь в его интересах, потому что ему нужно, чтобы CI прошла, чтобы он мог произвести слияние.

При использовании периодической CI ответственность гораздо более размыта. Чтобы CI прошла с пользой, кто-то должен получать уведомления о сбоях и исправлять ошибки. Уведомления можно настроить с помощью списка рассылки или создания панели инструментов; сложнее найти ответственного за решение проблемы.

В таких случаях обычно устанавливают ротацию (подобно рабочим дежурствам) и распределяют ответственность между членами команды. При наступлении сбоя разработчик, который в этот момент является ответственным, должен назначить проблеме приоритет и решить ее соответствующим образом.

Если периодическая CI часто завершается сбоем, постоянные исправления заметно снижают продуктивность разработчиков, вынужденных этим заниматься. Кроме того, это подрывает моральный дух команды. Поэтому для обеспечения надежности CI, чтобы сбои не стали обычным делом, (еще более) важно работать совместно.

Способы исправления нестабильной CI см. в главе 5.

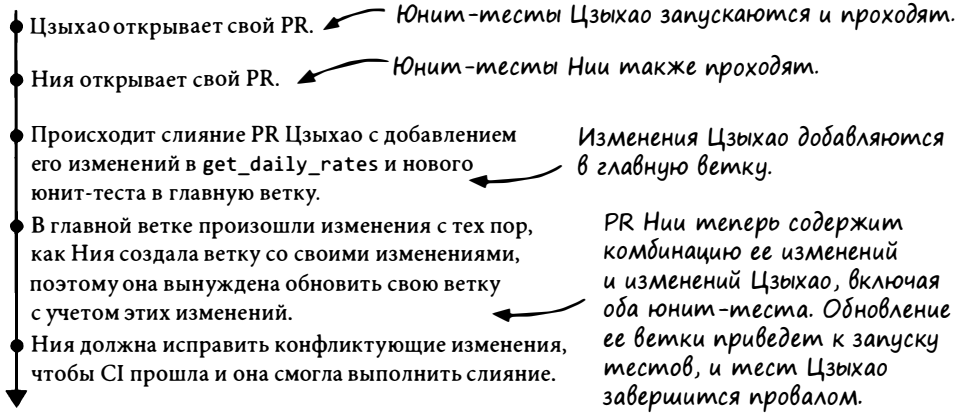
ВАЖНО

Периодический запуск CI способен выявить (но не предотвратить) описанный класс багов. Хотя настроить и запустить систему очень просто, но чтобы она была эффективной, кто-то должен отслеживать регулярные тесты и реагировать на сбои.

Вариант 2: постоянное обновление веток

Вариант 1 позволит обнаружить проблему, но не предотвратит ее появление. В варианте 2 проблема гарантированно не возникнет, поскольку при обновлении главной ветки потребуется обновить добавляемую ветку перед слиянием — и при ее обновлении сработает CI.

Решило бы это проблему Нии и Цзыхао? Посмотрим, что бы произошло.



Как только Цзыхао произведет слияние, Ния не сможет производить слияния, пока не подтянет последнюю версию главной ветки, включая изменения Цзыхао. Это снова запустит CI, что, в свою очередь, приведет к запуску юнит-тестов Нии и Цзыхао. Тест Цзыхао завершится с ошибкой, и проблема будет обнаружена!

Однако эта стратегия сопряжена с дополнительными издержками: при каждом обновлении главной ветки необходимо обновлять и все PR для веток, которые не содержат этих изменений. В случае Нии и Цзыхао это важно, потому что их изменения конфликтуют, но это необходимо делать всегда, независимо от того, важно ли вытягивать изменения.

Можно ли автоматизировать обновление всех веток?

Автоматизировать обновление всех веток можно (и это неплохая дополнительная функциональность для систем контроля версий). Однако помните, что для распределенных систем контроля версий, таких как Git, ветка, поддерживающая PR, является *копией* ветки, которую разработчик редактировал на своей машине. Таким образом, разработчику придется вытягивать любые автоматически добавленные изменения, необходимые для продолжения работы, — это некритично, но создает дополнительные сложности. Кроме того, с такой автоматизацией процессов мы уже приближаемся к варианту 3.

Вариант 2: какой ценой?

Обновляя сливаемую ветку в соответствии с изменениями в главной ветке перед слиянием, можно решить проблему Нии и Цзыхао, но такой подход затронет каждый PR и каждого разработчика. Стоит ли игра свеч? Посмотрим, как эта стратегия повлияет на несколько PR:

- Открыт PR № 45.
- Открыт PR № 46.
- Открыт PR № 47.
- Выполняется слияние для PR № 45, происходит обновление главной ветки.
- Главная ветка обновлена, поэтому слияние PR № 46 и № 47 заблокировано до их обновления.
- Открыт PR № 48.
- Для PR № 46 проведено обновление, а затем слияние.
- Главная ветка снова обновилась, теперь PR № 48 тоже заблокирован до своего обновления, а PR № 47 продолжает блокироваться изменениями, произведенными PR № 45, а теперь и PR № 46.

Каждое слияние PR влияет (и блокирует) на все другие открытые PR! В компании CoinExCompare около 50 разработчиков, и каждый из них сливает свои изменения в главную ветку ежедневно или почти каждый день. Следовательно, в день происходит около 20–25 слияний с главной веткой.

См. главу 8, чтобы понять, почему полезно так часто проводить слияние.

Представьте, что в любой момент времени открыто 20 PR, и авторы этих пулреквестов пытаются произвести их слияние после открытия в течение целого дня. После слияния один PR блокирует остальные 19 до тех пор, пока они не будут обновлены с учетом последних изменений.

Вариант 2 гарантирует, что CI всегда будет работать с последними изменениями, но цена этому — множество утомительных обновлений всех открытых PR. В худшем случае разработчики будут постоянно гнаться за тем, чтобы успеть слить свой PR раньше, чем его заблокируют чужие изменения.

ВАЖНО

Требование обновлять ветку перед слиянием предотвращает возникновение конфликтующих изменений, но этот подход оптимален, если над кодом работает всего несколько человек. В противном случае он нецелесообразен.

Вариант 3: автоматическое слияние силами CI

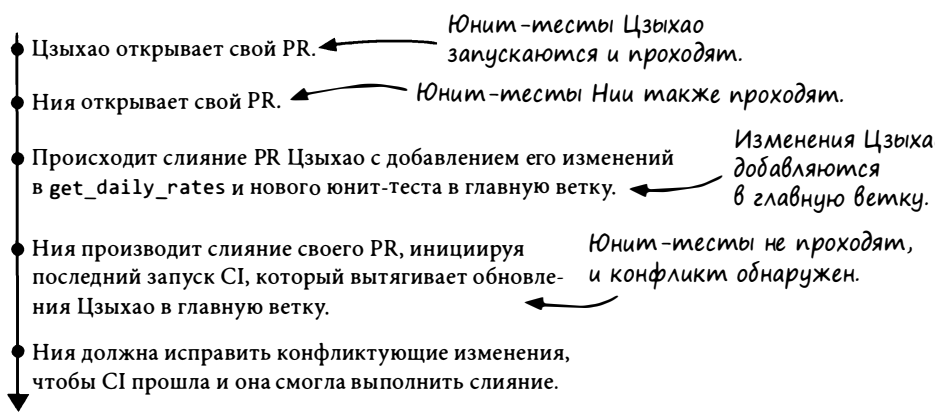
Команда CoinExCompare решает, что дополнительные издержки и неудобства, связанные с требованием постоянно обновлять ветки перед слиянием, не стоят полученной выгоды. Что еще может сделать команда?

При текущей настройке системы CoinExCompare тесты запускаются для PR Нии и Цзыхао перед слиянием. Эти тесты запускаются снова, если что-то в этих PR меняется. Это прекрасно работало для изменений, сделанных Цзыхао, но не позволяло обнаруживать проблемы, возникающие в изменениях Нии. Если перед слиянием запускать еще раз только CI Нии и включать в нее последние изменения из главной ветки при выполнении тестов, то проблема будет обнаружена.

Итак, возможное решение — автоматизировать запуск финальной CI перед слиянием, для изменений, слитых с последним вариантом кода из главной ветки. Для этого нужно сделать следующее:

1. Перед слиянием запустите CI снова, даже если она уже успешно прошла, включая последнее состояние главной ветки (даже если сама ветка не обновлялась).
2. Если главная ветка изменилась во время этого финального запуска, запустите CI снова. Повторяйте запуски до тех пор, пока CI не будет успешно выполнена именно с тем состоянием главной ветки, с которым вы будете производить слияние.

Что произошло бы с изменениями Нии и Цзыхао, если бы они использовали такую автоматизацию?



Если CI вытянет последнюю версию главной ветки (с изменениями Цзыхао) и запустится еще раз, прежде чем Ния произведет слияние, конфликтующие изменения будут обнаружены и не попадут в главную ветку.

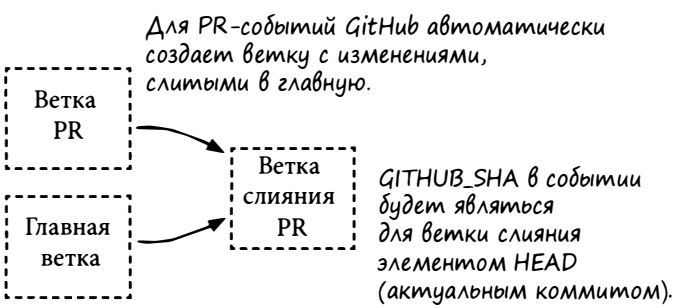
Вариант 3: запуск CI с последней версией главной ветки

Теоретически имеет смысл запустить CI перед слиянием с последней версией главной ветки и убедиться, что в нее нельзя внести изменения без повторного запуска CI, но как это сделать? Можно разбить элементы на более мелкие части. Итак, нам нужно следующее:

- Механизм объединения сливаемой ветки с последними изменениями в главной ветке, который подходит для CI.
- Механизм, запускающий CI перед слиянием и блокирующий слияние до тех пор, пока CI не пройдет успешно.
- Способ обнаруживать обновления в главной ветке (и инициировать повторный запуск CI перед слиянием) *или* блокировать внесение изменений в главную ветку в то время, когда CI перед слиянием запущена.

Как объединить сливаемую ветку с последними изменениями в главной ветке? Один из способов — сделать это самостоятельно с помощью задач CI, вытянув главную ветку и выполнив слияние.

Но часто это не нужно, потому что некоторые системы контроля версий выполняют эту работу самостоятельно. Например, когда GitHub запускает события веб-хука (или при использовании GitHub Actions), GitHub предоставляет *коммит слияния (объединенный коммит)* для повторного тестирования: он создает коммит, в котором PR-изменения уже слиты в главную ветку.



Использование коммита слияния для всех CI, инициированных PR, повысит вероятность обнаружения скрытых конфликтов.

Пока задачи получают этот коммит слияния (предоставляемый как GITHUB_SHA в инициирующем событии), пункт (1) можно считать выполненным!

Вариант 3 (автоматическое слияние CI) представляет собой измененный вариант 2. Вариант 2 требует обновлять ветки перед слиянием, а вариант 3 обеспечивает актуальность веток при запуске CI благодаря автоматизации обновления ветки до текущего состояния перед запуском CI вместо блокировки и ожидания, пока автор обновит свой PR с последними изменениями.

Вариант 3: события слияния

В GitHub запуск при слиянии устроен довольно хитро: эквивалентом события слияния является событие `pull_request` с действием типа `close`, когда поле `merged` в полезных данных имеет значение `true`. Не так-то просто!

Итак, у нас есть первый требуемый компонент. Но нам по-прежнему необходимо следующее:

- Механизм, запускающий CI перед слиянием и блокирующий слияние до тех пор, пока CI не пройдет успешно.
- Способ обнаруживать обновления в главной ветке (и инициировать повторный запуск CI перед слиянием) *или* блокировать внесение изменений в главную ветку в то время, когда CI перед слиянием запущена.

Большинство систем контроля версий позволяют запускать CI в качестве реакции на определенные события, такие как открытие PR, его обновление или, в данном случае, его слияние, так называемое *событие слияния* (merge event). Если вы запускаете CI в ответ на событие слияния, вы получите уведомление, когда произойдет слияние, и сможете перезапустить CI в ответ. Однако это не совсем то, что нужно:

- Событие слияния будет срабатывать *после* того, как произойдет слияние (после того, как PR будет слит обратно в главную ветку), поэтому если проблема будет обнаружена, она уже попадет в главную ветку. Да, вы будете знать об этом, но главная ветка будет сломана.
- Не существует механизма, гарантирующего, что изменения в главной ветке, произошедшие во время выполнения автоматизации, вызовут повторный запуск CI, поэтому конфликты все равно могут остаться незамеченными.

Посмотрим, как это будет выглядеть для сценария Нии и Цзыхао.

- Происходит слияние PR Цзыхао с добавлением его изменений в `get_daily_rates` и нового юнит-теста в главную ветку.
- Слияние Цзыхао инициирует повторный запуск CI для его изменений вместе с последней версией главной ветки.
- Происходит слияние PR Нии, добавление ее изменений в `get_daily_rates` и нового юнит-теста в главную ветку.
- Слияние Нии инициирует повторный запуск CI для ее изменений вместе с последней версией главной ветки.
- Изменения Нии и Цзыхао находятся в главной ветке.

Сможет ли CI обнаружить конфликт, зависит от времени ее запуска: CI для слияния Нии может запуститься, когда CI для Цзыхао еще работает, в этом случае изменения Цзыхао еще не будут находиться в главной ветке и CI для слияния Нии не выявит проблему.

Независимо от того, обнаружит ли автоматический запуск CI проблему, главная ветка сломана.

Увы, запуск CI при слияниях не принесет желаемого результата. Он повысит вероятность обнаружения конфликтов, но они уже будут в главной ветке, а во время работы автоматического процесса могут добавляться и новые конфликтующие изменения.

Вариант 3: очереди слияний

Если срабатывание на событие слияния не дает результата, что еще можно сделать? Чтобы добиться цели, нам необходимо следующее:

- Механизм объединения сливаемой ветки с последними изменениями в главной ветке, который подходит для CI.
- Механизм, запускающий CI перед слиянием и блокирующий слияние до тех пор, пока CI не пройдет успешно.
- Способ обнаруживать обновления в главной ветке (и инициировать повторный запуск CI перед слиянием) *или* блокировать внесение изменений в главную ветку в то время, когда CI перед слиянием запущена.

Мы выполнили первое требование, но не нашли подходящего решения для двух других. На самом деле секрет кроется в создании автоматической CI, которая будет полностью отвечать за слияние PR-запросов. Подобную автоматизацию часто называют *очередью слияния* (merge queque) *или цепочкой слияний* (merge train); слияние никогда не выполняется вручную, только автоматически, что обеспечивает выполнение двух последних требований.

> В приложении B рассматриваются такие функции, как очереди слияния и запуск на основе событий в системах контроля версий.

Вы можете создать очередь слияния самостоятельно, но к счастью, это не обязательно! Многие системы контроля версий теперь содержат встроенную функцию очереди слияния.

Очереди слияния, как следует из названия, управляют очередностью PR, готовых к слиянию (например, если они уже прошли все необходимые CI):

- Каждый подходящий PR добавляется в очередь слияния.
- Для каждого PR в порядке очереди слияния создается временная ветка, которая сливает изменения в главную ветку (применяя логику, которую GitHub использует для создания коммита слияния (или объединенного коммита), передаваемого в события PR).
- Очередь слияния запускает обязательную CI на временной ветке.
- Если CI проходит, очередь слияния продолжит работу и выполнит слияние. Если нет, то слияния не будет. В это время слияние других запросов невозможно, поскольку все они должны пройти через очередь слияния.

Для очень загруженных репозиториев некоторые очереди слияния оптимизируются путем *пакетной обработки* пул-реквестов для последующего слияния и запуска CI. Если CI не проходит, можно использовать такой подход, как бинарный поиск, чтобы быстро выявить неисправные PR: например, разделить пакет на две группы, повторно запустить CI для каждой из них и повторять этот процесс до тех пор, пока не обнаружится, какой PR (или несколько) вызвал сбой CI. Учитывая, насколько редки конфликты после слияния, эта оптимизация может сэкономить много времени, если несколько PR долго ожидают слияния в очереди.

Вариант 3: очередь слияния для CoinExCompare

Посмотрим, как очередь слияния будет работать в случае конфликта изменений Нии и Цзыхао:

- PR Цзыхао готов к слиянию.
- PR Цзыхао добавляется в очередь слияния.
- Очередь слияния создает новую ветку с копией главной ветки, производит слияние изменений Цзыхао и запускает CI.
- PR Нии готов к слиянию.
- PR Нии добавляется в очередь слияния.
- Очередь слияния выполняет CI для PR Цзыхао, поэтому PR Нии придется подождать.
- PR Цзыхао проходит, и очередь слияния вносит его изменения в главную ветку.
- Пришел черед Нии: очередь слияния создает новую ветку с копией главной ветки (включая недавно слитые изменения Цзыхао), сливает ее с изменениями Нии и запускает CI.
- CI проваливается, и Ниа должна разобраться с конфликтующими изменениями, прежде чем сможет произвести слияние своего PR.

Даже если попытки Цзыхао и Нии выполнить слияние накладываются друг на друга, очередь слияния справляется с состоянием гонки и предотвращает возникновение конфликтов.

Конфликт был пойман раньше и не попал в главную ветку!

Создание очереди слияния

Создать очередь слияния самостоятельно можно, но это требует много усилий. На высоком уровне потребуется сделать следующее: создать систему, которая имеет информацию о статусе всех находящихся активных PR; заблокировать все PR для слияния (например, с помощью правил защиты веток), пока система не даст зеленый свет; добиться, чтобы эта система выбирала PR, готовые к слиянию, сливала их с главной веткой и запускала CI, и наконец, чтобы эта система выполняла фактическое слияние. В такой сложной процедуре велика вероятность возникновения ошибок, но если гарантировать отсутствие конфликтов абсолютно необходимо, а система контроля версий не поддерживает очереди слияния, то усилия оправданны.

ВАЖНО

Очереди слияния предотвращают появление конфликтов, управляя слиянием и обеспечивая запуск CI для комбинации сливаемых изменений и последнего состояния главной ветки. Многие системы контроля версий предоставляют такую функциональность, что радует, поскольку создать подобную систему самостоятельно довольно сложно.

Ваш ход: сопоставьте варианты с их недостатками

Вернемся к трем вариантам обнаружения конфликтов слияния:

1. Периодически запускать CI.
2. Требовать, чтобы все ветки обновлялись в соответствии с последними изменениями.
3. Использовать очередь слияния.

Для каждого из них выберите из следующего списка два самых характерных недостатка:

A. Замедляет время слияния PR.
B. Кто-то должен следить за результатами и нести за них ответственность.
C. Приводит к тому, что для многих PR блокируется возможность слияния, пока авторы не разблокируют их.
D. Допускает поломку главной ветки.
E. Сложно реализовать, если система контроля версий не поддерживает эту функцию.
F. Долгий процесс, если разработчиков много.

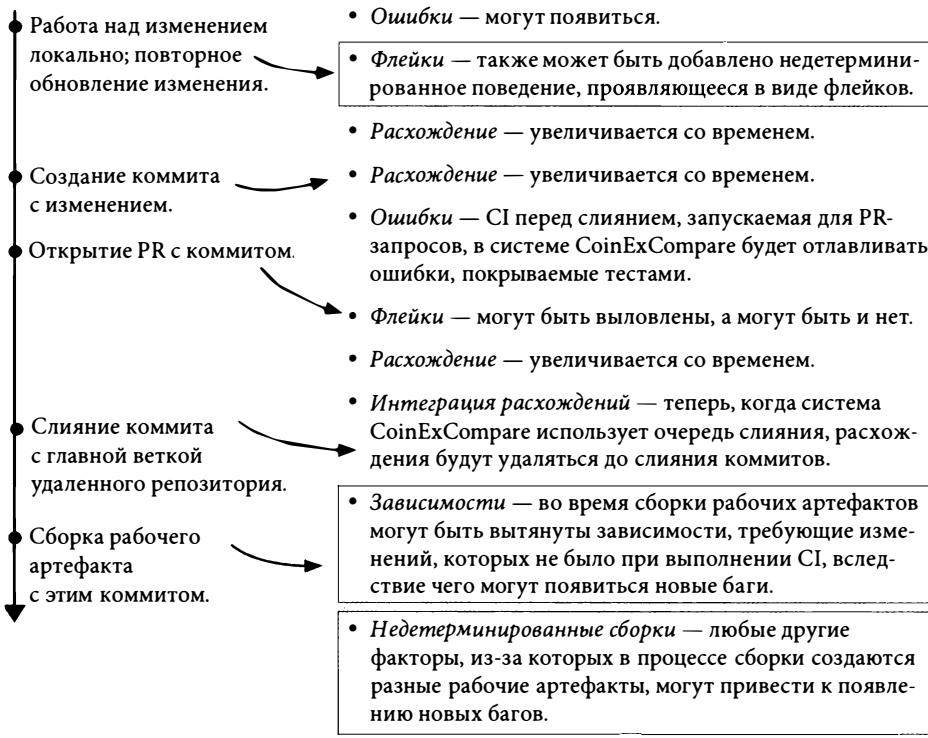
Ответы

1. Периодически запускать CI: лучше всего подходят варианты B и D. Конфликты, возникшие между PR, при периодическом запуске CI будут обнаружны уже после слияния с главной веткой. Если не следить за периодическими запусками, они будут бесполезны.
2. Требовать актуализации веток: лучше всего подходят варианты C и F. При каждом слиянии все остальные открытые PR блокируются до своего обновления. Если в проекте участвует всего несколько разработчиков, это не проблема, но в большой команде ожидание может быть утомительно.
3. Использовать очередь слияния: лучше всего подходят варианты A и E. Каждому PR потребуется запустить дополнительную CI перед слиянием, и, возможно, ожидать своей очереди. Создать систему очереди слияния самостоятельно довольно сложно.

Где еще могут возникнуть баги?

Команда CoinExCompare решает использовать очередь слияния, и с помощью GitHub сделать это довольно просто, нужно только добавить в правила защиты веток для главной ветки параметр — *требование очереди слияния*.

Теперь, используя очередь слияния, удается ли разработчикам CoinExCompare успешно выявлять и устранять все места, где могут появиться баги? Посмотрим еще раз на временную шкалу изменений и на то, в какой момент могут появляться ошибки:



Даже после введения очереди слияния остается несколько потенциальных источников багов, которые команда CoinExCompare еще не устранила:

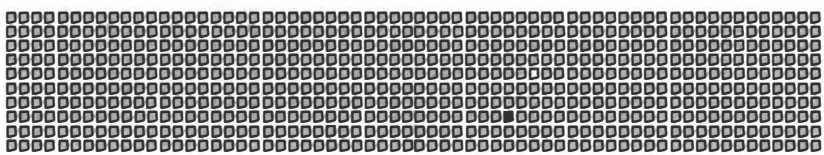
- ~~Расхождение с основной веткой и интеграция с ней~~ (Устранено!)
- Изменения в зависимостях.
- Недетерминированность: в коде и/или в тестах (то есть флейки), и/или в способе сборки готовых артефактов.

Флейки и CI, инициируемая пул-реквестом

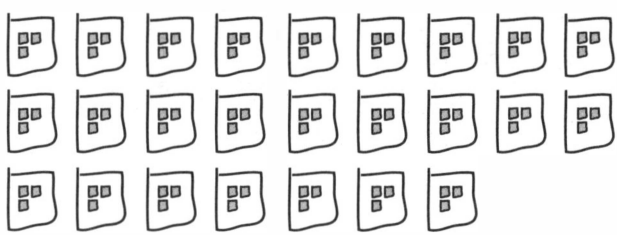
В главе 5 отмечалось, что *флейки* возникают, когда тесты работают непоследовательно: иногда проходят, а иногда нет. Кроме того, выяснилось, что это может быть связано как с проблемой в тесте, так и с проблемой в тестируемом коде, поэтому лучшая стратегия — считать флейки багами и полностью разбираться с ними. Но поскольку флейки возникают не всегда, их бывает трудно обнаружить!

Сейчас система CoinExCompare запускает CI для каждого PR и перед слиянием PR. Именно в этом случае могут проявиться флейки, но, будем честны, их часто игнорируют. Трудно удержаться от соблазна просто запустить тесты заново, выполнить слияние и на этом успокоиться, особенно если кажется, что ваши изменения никак с этим не связаны.

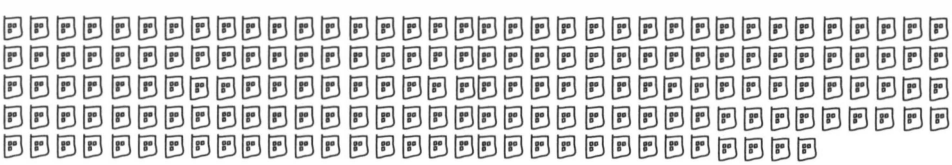
Есть ли более эффективный способ выявлять флейки и бороться с ними? Недавно мы обсуждали периодическую CI и решили, что это не лучший способ борьбы со скрытыми конфликтами. Однако оказалось, что она отлично находит флейки. Представьте себе тест, который дает сбой только один раз на 500 запусков.



У разработчиков сайта CoinExCompare открыто около 20–25 PR в день. Предположим, что CI запускается по крайней мере три раза для каждого PR: один раз в начале, другой — для изменений и, наконец, последний раз в очереди слияния. Это означает, что каждый день появляется около 25 PR × 3 запуска = 75 рисков провалить тест.

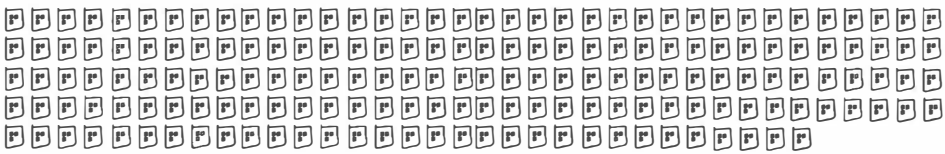


За 7 дней это 525 изменений, которые рискуют провалиться, так что этот тест, скорее всего, провалит один из этих PR. (А создатель PR, скорее всего, просто проигнорирует этот сбой и запустит CI снова!)



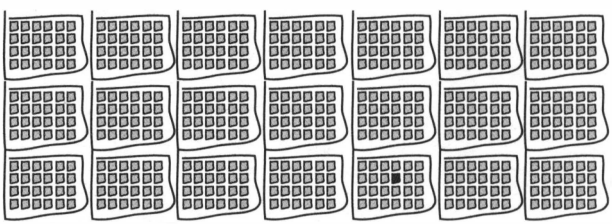
Отлавливание флейков периодическими тестами

Имея для выявления флейков только тесты, основанные на запуске по PR и очередях слияния, система CoinExCompare сможет воспроизвести флейк, который возникает один раз из 500, примерно один раз в 7 дней. И когда он возникнет, автор соответствующего PR, скорее всего, просто перезапустит тесты и пойдет дальше.



Может ли CoinExCompare облегчить воспроизведение флейков, а не полагаться на добросовестность разработчика? Недавно мы говорили о периодических тестах и о том, что это не лучший способ предотвратить появление конфликтов, но оказалось, что такие тесты отлично отлавливают флейки! Что, если в системе CoinExCompare настроить периодическую CI на запуск один раз в час? В день она сможет запускаться 24 раза.

Flaky-тест дает сбой в одном из 500 запусков, поэтому для воспроизведения сбоя понадобится 500 / 24 дня, или примерно 21 день.



Подобные расчеты могут показаться не очень воодушевляющими, но суть в том, что если периодические тесты поймают сбой, они не помешают чьей-то работе, не связанной с этим флейком. Если существует определенный процесс обработки сбоев, выявленных с помощью периодической CI, то существует и больше шансов обработать и тщательно проанализировать флейк, чем если он обнаружится неожиданно и заблокирует чью-то работу, не имеющую к нему отношения.

ВАЖНО

Периодические тесты помогают выявлять и исправлять недетерминированное поведение в коде и тестах, не блокируя не связанную с этим работу.

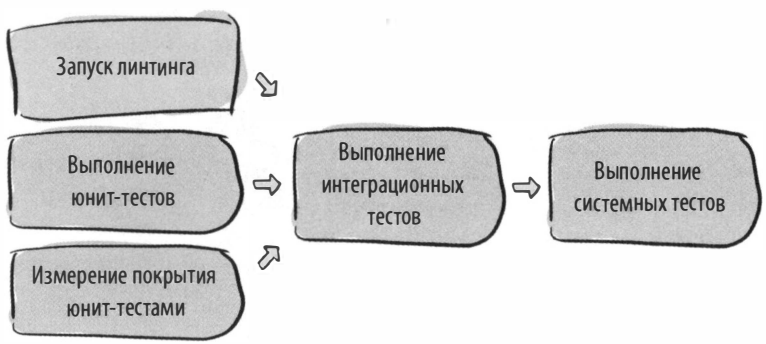
Баги и сборка

Добавив очередь слияния и периодические тесты, компания CoinExCompare успешно устранила большинство потенциальных источников багов, но у багов все еще остались способы проникнуть в систему:

- ~~Расхождение с основной веткой и интеграция с ней~~ (Устранено!)
- Изменения в зависимостях.
- Недетерминированность: ~~в коде и/или тестах (то есть флейки)~~ (Устранено с помощью периодических тестов!) и/или в способе сборки артефактов.

Последние два источника багов связаны с процессом сборки. В главе 9 будет описано, как структурировать процесс сборки, чтобы избежать этих проблем, а пока вопрос: как выявлять и исправлять баги, возникающие во время сборки, не меняя процесс создания образов в CoinExCompare? Вспомним схему пайплайна:

> Спойлер: лучший способ борьбы с багами, возникающими при изменениях в зависимостях, — всегда закреплять свои зависимости. См. главу 9.



Последняя задача в пайплайне запускает системные тесты. Как и все они, эти тесты проверяют систему CoinExCompare в целом. Этим тестам нужно что-то, с чем можно работать, поэтому часть задачи должна включать настройку *тестируемой системы* (system under test, SUT). Чтобы создать SUT, задача должна произвести сборку образов, используемых сайтом CoinExCompare.

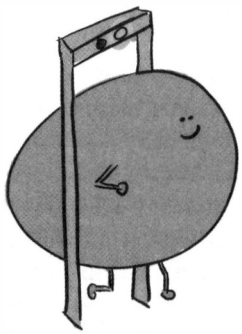
Словарик

Тестируемая система (system under test, SUT) — это система, которую тестируют для проверки того, что она работает правильно.

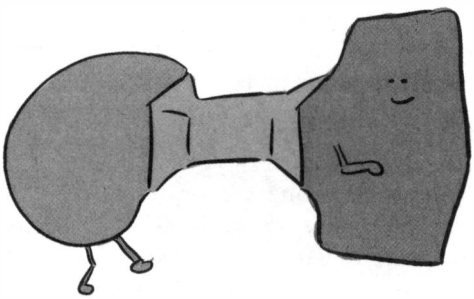
Баги, которые мы сейчас рассматриваем, проникают в систему во время процесса создания образов — так можно ли их выявить системными тестами? Да, но проблема в том, что образы, построенные для системных тестов, не совпадают с образами, которые будут созданы и развернуты на продакшен. Процесс сборки этих образов будет проводиться позже, и тогда баги смогут снова проникнуть в систему.

Непрерывная интеграция либо сборка и развертывание

В главе 2 мы рассматривали создание шлюзов и преобразований.



Задачи, проверяющие код, — это «шлюзы» качества, через которые он должен пройти.



Задачи, которые изменяют форму кода, называются преобразованиями: код поступает в качестве входных данных, а выходит в другом виде.

Система CoinExCompare разделяет задачи шлюзов и преобразований на два пайплайна. Цель пайплайна, который мы рассматривали до сих пор, пайплайна CI, — проверка изменений кода (так называемое шлюзование изменений кода). Другой пайплайн в CoinExCompare используется для сборки и развертывания своего рабочего образа (так называемое преобразование исходного кода в запускаемый контейнер):

Подробнее о реализации пайплайнов см. в главе 13.

В реальности грань между этими задачами может быть размыта. Если вам требуется уверенность в решениях, принимаемых задачами типа «шлюз» (CI), вам необходимо определенным образом «преобразовать» CI. Такая ситуация часто возникает при проведении системных тестов, которые скрыто выполняют определенный объем сборки и развертывания.

Сборка и развертывание с использованием одинаковой логики

Тестирование системы CoinExCompare включает следующие задачи:

1. Настройка среды для работы тестируемой системы.
2. Сборка образа.
3. Передача образа в локальный реестр.
4. Запуск образа.
5. И только *потом* запуск системных тестов на уже работающем контейнере.

Однако — и довольно часто — логика, используемая этой системой, отличается от той, что применяется в пайплайне выпуска релиза для сборки и развертывания образов. Если бы она была одинаковой, задачи были бы те же, что и в пайплайне:

Параметры среды выполнения, предоставляемые этим задачам, могут изменить место фактического обновления и развертывания задач, например, на реальный реестр образов или на временный локальный реестр. Управление поведением задач с помощью параметров позволяет повторно использовать ту же логику.

Это означает, что баги могут появиться во время сборки и развертывания реальных образов, а именно:

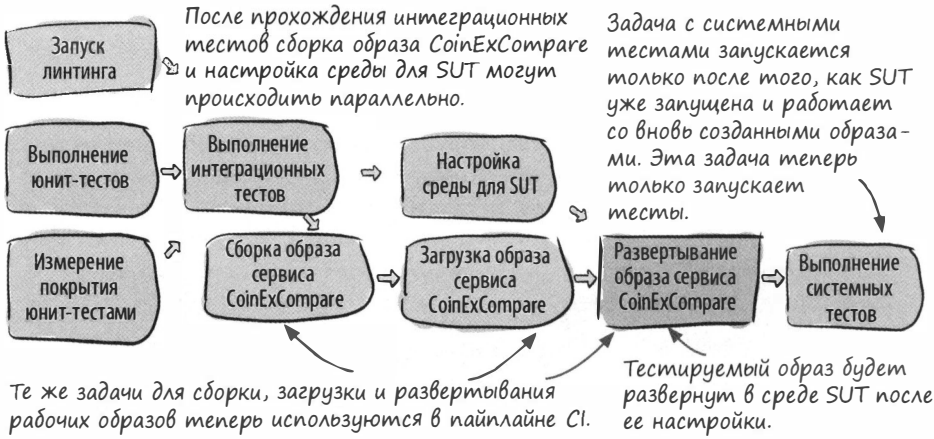
- Различия в зависимости от того, *когда происходит сборка*: например, во время системных тестов извлекается последняя версия зависимости, а при сборке рабочего образа вытягивается еще более новая версия.
- Различия в зависимости от *среды сборки*: например, запуск сборки на другой версии базовой операционной системы.

Чтобы минимизировать эти различия, команда CoinExCompare может сделать следующее:

- периодически запускать задачи развертывания;
- использовать те же задачи для сборки и развертывания при выполнении системных тестов, что и для фактической сборки и развертывания.

Усовершенствованный пайплайн CI с процессом сборки

Компания CoinExCompare обновила свой пайплайн CI, чтобы системные тесты использовали те же задачи, что и для сборки и развертывания на продакшен. Обновленный пайплайн выглядит так:



Периодические тесты будут запускать этот пайплайн каждый час, а добавление в него сборки и развертывания означает, что периодически запускаются и автоматические задачи развертывания.

> CoinExCompare не использует непрерывное развертывание; подробнее о различных методах развертывания см. в главе 10.

Помогло ли это устранить все потенциальные источники багов? Вспомним еще раз, от каких ошибок пытались избавиться разработчики:

- ~~Расхождение с основной веткой и интеграция с ней~~
- Изменения в зависимостях
- Недетерминированность: ~~в коде и/или тестах (то есть флейки) и/или~~ в способе сборки артефактов.

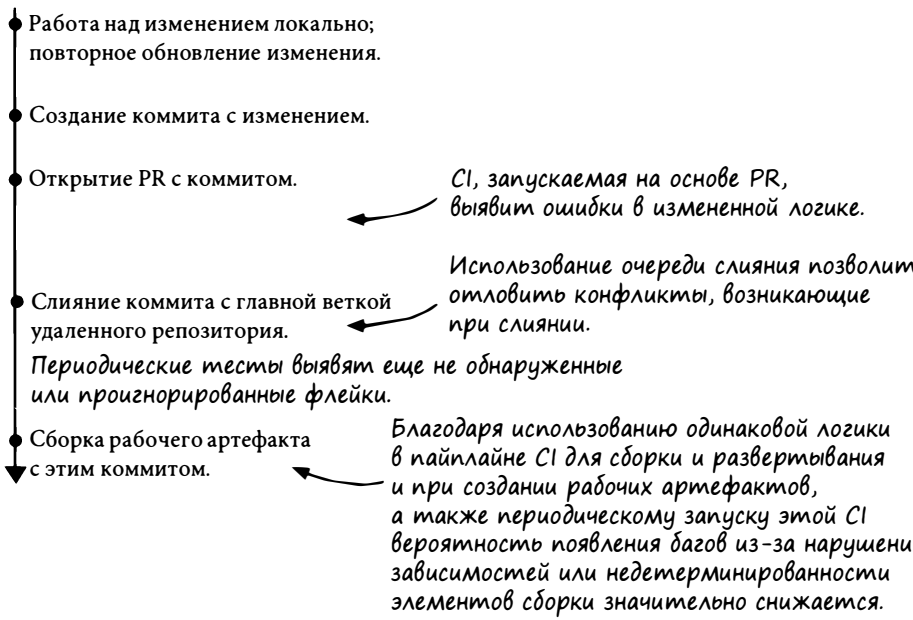
Возможно, в системе еще остались баги каких-то из перечисленных видов. Но поскольку пайплайн выпуска релиза теперь запускается в другое время и с другими параметрами, вероятность появления любого из этих багов стала значительно ниже:

- Вероятность появления изменений в зависимостях сведена к минимуму, поскольку сборка (и тестирование) образов теперь производится каждый час. Если изменение в зависимости вызывает ошибку, то теперь у нее есть всего около часа, чтобы успеть проскользнуть в код; кроме того, ее, скорее всего, отловит следующий запуск периодической CI.
- Количество недетерминированных сборок сокращается, поскольку, используя одинаковые задачи для сборки образов при CI, мы сократили количество переменных, которые могут отличаться.

(Подробнее о том, как полностью устранить эти риски, читайте в главе 9.)

Еще раз о временной шкале изменений

Все ли «ошибкоопасные» места теперь выявлены? Ребята из CoinExCompare садятся за стол, чтобы в последний раз проанализировать, где может закрасться ошибка:



Команда CoinExCompare успешно устранила места (или, по крайней мере, минимизировала их количество), где могут появиться баги, сделав следующее:

- продолжив использовать существующую CI, инициируемую на основе PR;
- добавив очередь слияния;
- начав запускать CI периодически;
- обновив пайплайны CI, чтобы они использовали ту же логику для сборки и развертывания, что и пайплайны выпуска рабочих релизов.

После внедрения этих функций разработчики CoinExCompare с радостью отметили, что количество ошибок и сбоев значительно сократилось.

Артефакты периодической CI как кандидаты на выпуск релиза

Риск возникновения багов из последних двух источников был только смягчен, но не устранен. Самое простое и быстрое решение в этом случае — начать считать артефакты периодической CI кандидатами на релиз и выпускать эти образы как есть, без изменений, то есть больше не запускать отдельный пайплайн для повторной сборки перед релизом.

Ваш ход: определите, что упущено

Для каждой из следующих настроек запуска укажите баги, которые могут возникнуть, и отметьте любые очевидные недостатки этого подхода (считайте, что других запусков CI нет):

1. Периодический запуск CI.
2. Запуск CI после слияния с главной веткой.
3. Запуск CI, инициируемый PR.
4. Запуск CI, инициируемый PR, с использованием очередей слияния.
5. Запуск CI как части рабочего пайплайна сборки и развертывания.

Ответы

1. Периодическая CI может самостоятельно отлавливать ошибки, а иногда и флейки; однако она будет это делать, только если они уже попали в главную ветку. Чтобы периодическая CI работала эффективно, кто-то должен регулярно за ней следить и сортировать возникающие ошибки по их источникам.
2. Запуск после слияния с главной веткой будет отлавливать ошибки, но только если они уже попали в главную ветку. Поскольку запуск происходит сразу после слияния, определить ответственного за изменения будет проще, однако, скорее всего, любые обнаруженные флейки будут проигнорированы. Кроме того, придется соблюдать принцип «не сливать изменения в главную ветку, когда CI не работает», иначе ошибки будут наслаиваться друг на друга и разрастаться.
3. CI, инициируемая на основе PR, достаточно эффективна, но будет пропускать конфликты, возникающие между PR. Выявленные флейки, скорее всего, будут проигнорированы.
4. Добавление очереди слияния к CI, инициируемой на основе PR, устранит конфликты между PR, но флейки, скорее всего, все равно будут проигнорированы.
5. Запуск CI как части пайплайна выпуска рабочего продукта гарантирует, что ошибки, вызванные обновленными зависимостями (и некоторыми недетерминированными элементами), будут обнаружены до релиза, но отслеживание этих ошибок прервет процесс выпуска. Если возможности быстро исправить эти ошибки нет, а при перезапуске они исчезают, то, скорее всего, они будут проигнорированы.

Заключение

Разработчики CoinExCompare полагали, что запуска CI, инициируемой каждым PR, достаточно, чтобы всегда получать сигнал об ошибке, вызванной изменением. Однако на практике оказалось, что при таком подходе отловить все ошибки невозможно. Но благодаря использованию очередей слияния, добавлению периодических тестов и такой модернизации CI, чтобы она использовала ту же логику, что и пайплайны выпуска релиза, они могут обнаруживать практически все баги!

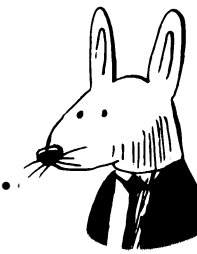
Итоги

- Баги могут возникать как часть самих изменений, как конфликты между изменениями и как расхождения с главной веткой, а также как часть процесса сборки.
- Очереди слияния — эффективный способ предотвратить возникновение изменений, вызывающих конфликты между пул-реквестами. Если эта функция не предусмотрена в системе контроля версий, небольшие команды могут использовать требование обновления веток. Периодические тесты также могут оказаться эффективными (хотя это не позволит избежать возможных поломок главной ветки).
- Периодические тесты стоит добавлять в любом случае, так как они способны выявлять флейки, не прерывая не связанные с этими флейками PR, но для их успешной работы потребуется наладить соответствующий процесс отслеживания.
- Организация процессов сборки и развертывания в пайплайнах CI таким же образом, как и в рабочих релизах, позволит снизить количество ошибок, которые могут появиться между запусками CI и пайплайнов выпуска.

Далее...

В следующей главе мы перейдем к рассмотрению особенностей пайплайнов непрерывной доставки, которые выходят за рамки CI: задач преобразования, используемых для сборки и развертывания кода. Мы подробно разберем эффективные способы контроля версий, которые помогут сглаживать процессы непрерывной доставки, а также узнаем, как измерить эту эффективность.

Часть 3
Сделаем процесс доставки проще



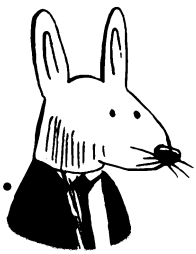
Теперь, когда вы понимаете, как поддерживать продукт в состоянии готовности к доставке, мы перейдем от верификации изменений в продукте к его выпуску.

В главе 8 мы еще раз рассмотрим систему контроля версий. Через призму метрик DORA вы увидите, что способ использования этой системы влияет на скорость выпуска релизов.

В главе 9 рассказывается, как безопасно создавать артефакты, применяя принципы, определенные стандартом SLSA, и объясняется важность версионирования.

В главе 10 мы вернемся к метрикам DORA, сосредоточившись на тех из них, что связаны со стабильностью, и рассмотрим различные методологии развертывания, которые можно использовать для повышения стабильности программного продукта.

8 Простая доставка начинается с контроля версий



В этой главе

- ✓ Использование метрик DORA, измеряющих скорость: частота развертывания и время доставки изменений
- ✓ Повышение скорости и усиление взаимодействия за счет отказа от долгоживущих веток функций и заморозок кода
- ✓ Сокращение времени доставки изменений за счет использования небольших и частых коммитов
- ✓ Безопасное увеличение частоты развертывания за счет использования небольших частых коммитов

В предыдущих главах основное внимание уделялось непрерывной интеграции (CI), но начиная с этой главы, мы обратимся к остальным элементам пайплайна непрерывной доставки (CD), в частности к задачам преобразования, которые используются для сборки, развертывания и выпуска кода.

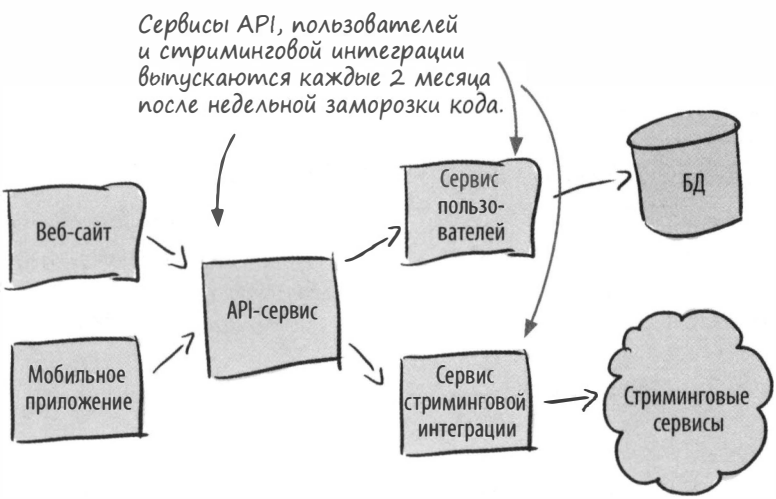
Применение правильных методик CI оказывает непосредственное влияние и на остальную часть процесса CD. В этой главе мы подробно остановимся на эффективных подходах к контролю версий, позволяющих наладить бесперебойную CD, а также на том, как измерять эту эффективность.

Тем временем на сайте Watch Me Watch

Помните стартап Watch Me Watch из главы 3? Так вот, он по-прежнему работает и даже растет! За последние два года компания выросла, и теперь в ней не только Саша и Сара, а более 50 сотрудников.

С самого начала компания Watch Me Watch вкладывала средства в автоматизацию развертываний. Но по мере роста компании разработчиков стало беспокоить, что эти развертывания становятся все более рискованными, поэтому они снизили их частоту.

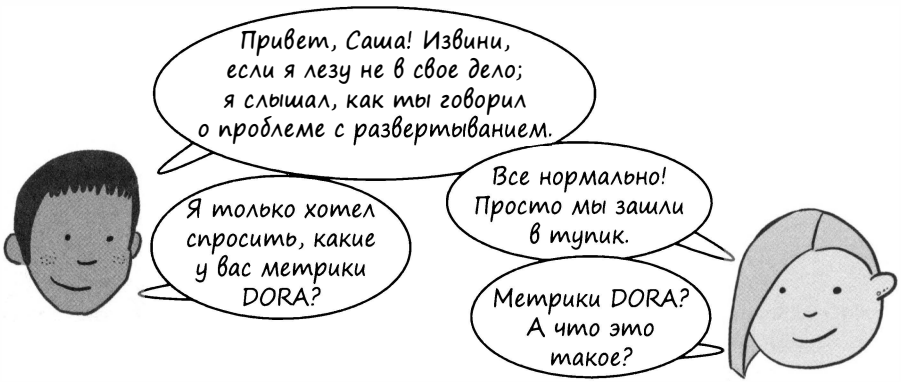
Каждый из сервисов теперь выпускается только в определенное время, один раз в 2 месяца. За неделю до релиза код замораживается, и изменения в него не вносятся.



Несмотря на эти нововведения, кажется, что проблема только усугубляется: каждое развертывание по-прежнему сопряжено с огромным риском, и что еще хуже, функции слишком долго запускаются на продакшен. С тех пор как Саша и Сара начали работу над своим первоначальным замыслом, у них появились конкуренты, и, учитывая медленный темп выпуска функций, похоже, что конкуренты вырываются вперед! Создается впечатление, что любое действие выполняется все медленнее и медленнее.

Метрики DORA

Саша и Сара в тупике, но у нового сотрудника Сэнди есть несколько идей о том, что можно изменить. Однажды Сэнди встречает Сашу в коридоре.



Пока они стоят в коридоре и Сэнди рассказывает про метрики DORA (https://www.devops-research.com/research.html), Саша понимает, что команде пригодятся знания Сэнди, и просит его провести презентацию для всей компании. Сэнди с удовольствием готовит несколько слайдов и быстро знакомит всех с метриками DORA:

История создания метрик DORA

Команда DevOps Research and Assessment (DORA) создала метрики DORA на основе почти десяти лет исследований.

Что такое метрики DORA?

Метрики DORA — это четыре ключевых показателя, которые измеряют эффективность команды разработчиков.

Метрики DORA для скорости

Скорость оценивается по двум метрикам:

- Частота развертывания
- Время доставки изменений

Метрики DORA для стабильности

Стабильность оценивается по двум метрикам:

- Время восстановления работоспособности сервиса
- Частота сбоев при изменениях

Метрики скорости для Watch Me Watch

После презентации метрик DORA Сэнди продолжил обсуждать с Сарой и Сашей, как эти метрики могут помочь в решении проблем Watch Me Watch, связанных с замедлением работы. Сэнди предлагает сосредоточиться на двух метриках скорости и измерить их для Watch Me Watch.

> Интересно узнать о двух других метриках DORA (для стабильности)? Более подробно о них мы поговорим в главе 10, при обсуждении развертывания.

Метрики DORA для скорости

Скорость оценивается по двум метрикам:

- Частота развертывания
- Время доставки изменений

Чтобы начать применять метрики, Сэнди необходимо изучить их немного подробнее:

- *Частота развертывания* показывает, как часто компания успешно выпускает релизы в рабочую среду.
- *Время доставки изменений* определяет количество времени, которое требуется, чтобы коммит попал на продакшен.

В компании Watch Me Watch развертывания могут происходить только так, как предусмотрено графиком развертывания, то есть один раз в 2 месяца. Таким образом, для Watch Me Watch частота развертывания составляет один раз в 2 месяца.

> Под *рабочей средой*, или *продакшен*, понимается среда, в которой программный продукт становится доступен клиентам (в отличие от промежуточных сред, которые могут использоваться для других целей, например для тестирования). Более подробное определение см. в главе 7.

Что делать, если у меня не сервис?

Если вы работаете над проектом, который не размещается и не запускается как сервис (более подробно о видах программных продуктов, которые можно поставлять, в том числе библиотеках, двоичных файлах, конфигурациях, образах и сервисах, см. в главе 1), вам может быть интересно, применимы ли к вашему случаю DORA-метрики.

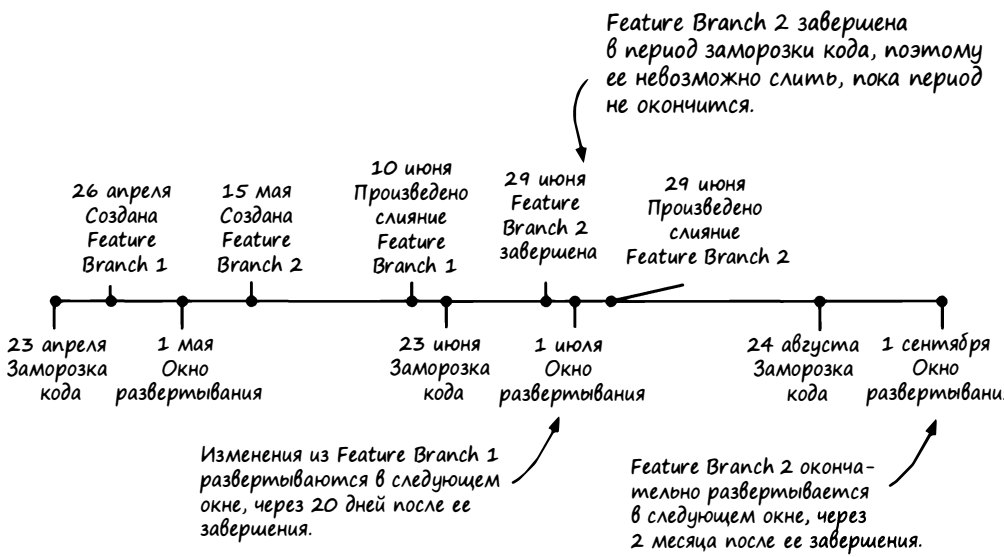
Метрики DORA были изначально определены для работающих сервисов. Тем не менее эти принципы подходят и в других случаях, но с небольшой переработкой. Метрики скорости можно применять к другим продуктам (например, библиотекам и двоичным файлам) следующим образом:

- *Частота развертывания* становится *частотой выпуска*. Чтобы соответствовать определениям, данным в главе 1 (публикация, развертывание, релиз), назовем эту метрику частотой релизов, что лучше отражает ее смысл: как часто изменения становятся доступными для пользователей, то есть выпускаются? (И еще вспомните из главы 1, что развертывание иногда более правильно называть релизом: когда мы говорим о непрерывном развертывании, на самом деле мы говорим о непрерывном релизе.) Размещение и запуск продукта как сервиса осуществляются путем развертывания изменений в продакшен и обеспечения доступа к ним пользователей, но если вы работаете над другим продуктом, например библиотекой или двоичными файлами, его запуск будет представлять собой выпуск новых релизов.
- *Время доставки изменений* остается тем же, но это не то время, которое требуется коммиту, чтобы попасть на продакшен, а время, за которое коммит попадает к пользователям. Например, в случае с библиотеками это будут пользователи, применяющие библиотеки в своих проектах. Сколько времени проходит между созданием коммита и моментом, когда его содержимое становится доступным для использования в выпущенной версии библиотеки?

Возникает вопрос, одинаково ли применимы значения наилучших, высоких, средних и низких показателей этих метрик для продуктов, работающих как сервис, и продуктов, распространяемых в виде библиотек и двоичных файлов. Ответа мы не знаем, однако принцип остается прежним: чем чаще изменения передаются пользователям, тем ниже риск ошибок при каждом релизе.

Время доставки изменений

Чтобы измерить время доставки изменений, Сэнди нужно разобраться в процессе разработки Watch Me Watch. Большинство функций создается в ветке функций, и эта ветка сливается обратно в главную по окончании работы. Некоторые функции можно реализовать всего за неделю, для большинства требуется минимум несколько недель. Вот как выглядит этот процесс для двух недавно разработанных функций, которые создавались в Feature Branch 1 (Ветка функции 1) и Feature Branch 2 (Ветка функции 2):



Время доставки изменений в Feature Branch 1 составило 20 дней. Несмотря на то что Feature Branch 2 была завершена перед самым открытием окна развертывания, это произошло в период заморозки кода, поэтому ее было невозможно слить, что отложило развертывание до следующего окна, открывшегося 2 месяца спустя. Таким образом, время доставки изменений для Feature Branch 2 составило 2 месяца, или около 60 дней. Проанализировав все функции и ветки функций за последний год, Сэнди выяснил, что среднее время доставки изменений составляет около 45 дней.

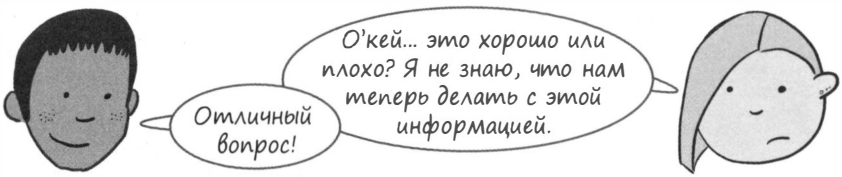
Словарик

Ветвление функции (feature branching) — это стратегия ветвления, согласно которой в начале разработки новой функции создается новая ветка (называемая веткой функции). Разработка функции в этой отдельной ветке продолжается до тех пор, пока функция не будет завершена, после чего она сливается с главной веткой. Подробнее об этом вы узнаете немного позже.

Сайт Watch Me Watch и наилучшие показатели

Сэнди измерил две связанные со скоростью метрики DORA для Watch Me Watch:

- *Частота развертывания* — раз в 2 месяца.
- *Время доставки изменений* — 45 дней.



Глядя на эти показатели по отдельности, трудно сделать выводы или вынести что-то полезное. В рамках определения этих метрик члены команды DORA также произвели классификацию оцениваемых ими команд по общей эффективности и распределили их по четырем категориям: низкоэффективные, среднеэффективные, высокоэффективные и суперэффективные.

Для каждой метрики они указали ее значение для команд в указанных категориях. Для метрик скорости классификация выглядит следующим образом (из отчета за 2021 год):

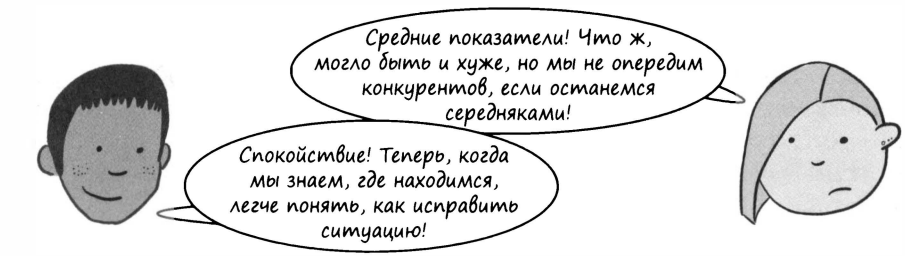
Метрика	Супер-эффективные	Высоко-эффективные	Средне-эффективные	Низко-эффективные
Частота развертывания	Несколько раз в день	От одного раза в неделю до одного раза в месяц	От одного раза в месяц до одного раза в полгода	Реже, чем раз в полгода
Время доставки изменений	Менее одного часа	От одного дня до одной недели	От одного месяца до полугода	Более полугода

Суперэффективные команды каждый день выполняют множество развертываний, а время доставки изменений в них составляет менее часа! Компании с низкими показателями развертывают систему реже чем раз в полгода, а на реализацию изменений у них уходит более 6 месяцев. Сравнивая показатели работы Watch Me Watch с этими значениями, можно сказать, что компания однозначно входит в группу среднеэффективных команд.

Что делать, если я нахожусь между двумя группами?

Результаты, представленные командой DORA, сгруппированы таким образом, что между ними существует небольшой разрыв. Эта классификация основана на значениях, полученных в результате исследования работы команд, и не является абсолютным руководством к действию. Если вы видите, что ваши показатели попадают в промежуток, вам решать, считать ли себя лучшим в нижней группе или худшим в верхней. Возможно, полезнее абстрагироваться и оценить свои показатели по всем метрикам, чтобы получить комплексное представление о своей производительности.

Повышаем скорость работы в Watch Me Watch

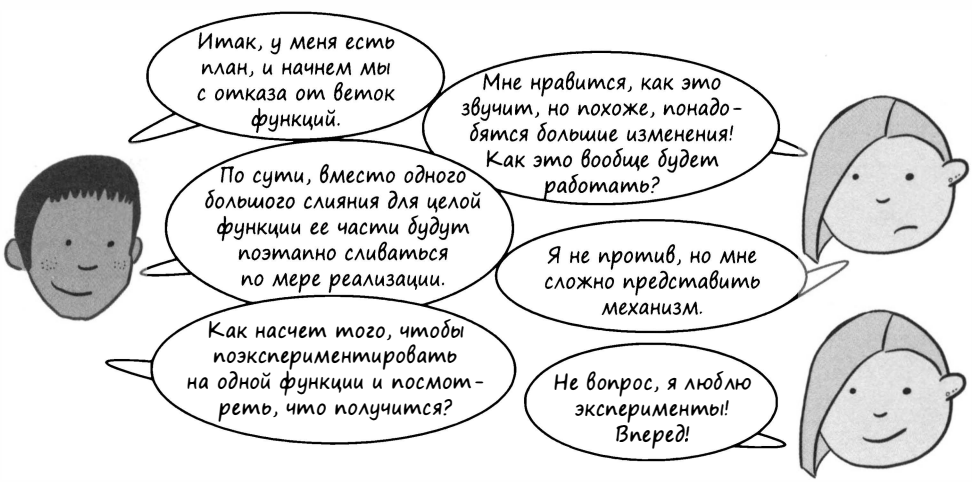


Сэнди приступает к разработке плана повышения скорости работы в Watch Me Watch:

- *Частота развертывания*. Чтобы перейти от среднего показателя к высокому, команда должна делать развертывания не реже одного раза в месяц.
- *Время доставки изменений*. Чтобы перейти от средних показателей к высоким, команде необходимо сократить этот срок с 45 дней до одной недели или даже меньше.

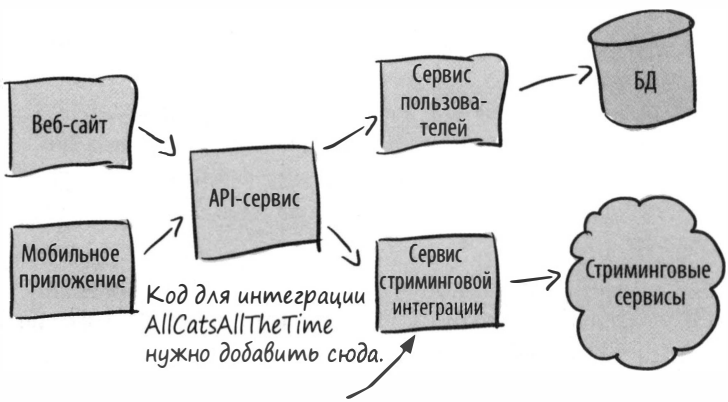
Сейчас частота развертывания в компании определяется фиксированными окнами развертывания — один раз в 2 месяца. Это также влияет на время доставки изменений: ветки функций не сливаются, пока не будет полностью завершена разработка функции, и слияние может производиться только между периодами заморозки кода, а если разработчик пропустит окно развертывания, то его изменения будут отложены на 2 месяца, до следующего окна.

Сэнди предположил, что на обе метрики сильно влияют окна развертывания (и заморозка кода непосредственно перед развертыванием), а усугубляет ситуацию использование веток функций.



Интеграция с AllCatsAllTheTime

Эксперимент с отказом от веток функций Сэнди планирует провести на функции, над которой работает Ян. Ян разрабатывает интеграцию с новым стриминговым провайдером AllCatsAllTheTime (этот провайдер предлагает тщательно отобранный контент о кошках). Чтобы понять, какие изменения придется внести Яну, вспомним схему архитектуры Watch Me Watch. Несмотря на то что компания выросла с тех пор, как мы в последний раз обращались к этой схеме, первоначальные идеи Саши и Сары по-прежнему работают, поэтому архитектура не изменилась:



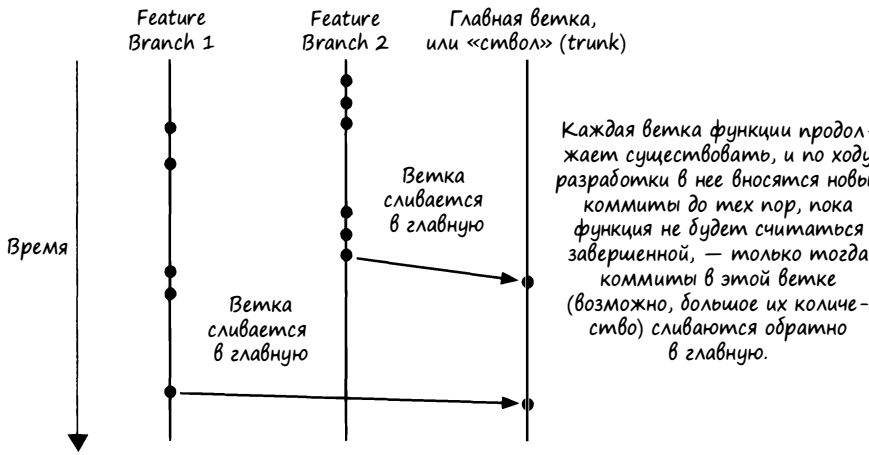
Интеграция с AllCatsAllTheTime в качестве нового провайдера стриминговых услуг означает изменение сервиса стриминговой интеграции. В коде этого сервиса каждый интегрированный стриминговый сервис реализуется как отдельный класс, и предполагается, что он наследует от класса `StreamingService`, реализуя следующие методы:

```
def getCurrentlyWatching(self):
  ...
def getWatchHistory(self, time_period):
  ...
def getDetails(self, show_or_movie):
  ...
```

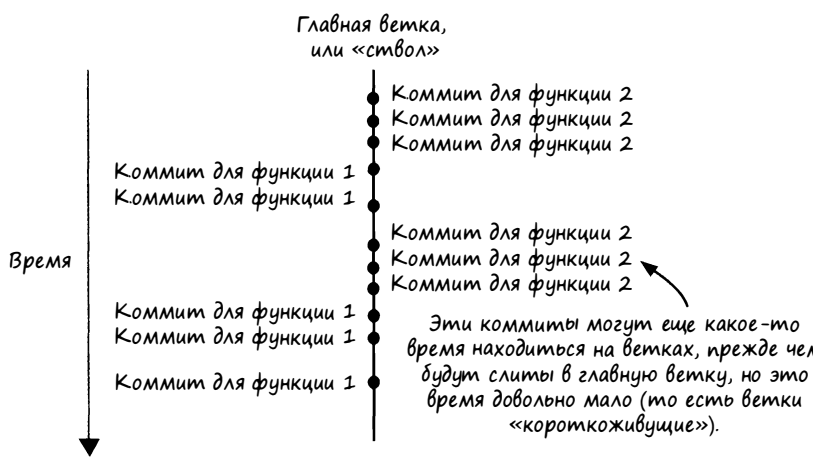
Этот интерфейс позволяет реализовать большинство функций, которые требуются Watch Me Watch от поставщиков стриминговых услуг: узнать, что смотрел пользователь, и получить подробную информацию о контенте, который он смотрел.

Магистральная разработка

Подход, который предлагает Сэнди, — это *магистральная разработка* (trunk-based development), при которой разработчики часто сливают изменения в магистральную (trunk) ветку репозитория, она же главная ветка. Это делается вместо *ветвления функций*, когда долгоживущие ветки создаются для каждой функции, а изменения фиксируются в ветке по мере разработки функции. В конце концов в какой-то момент вся ветка сливается обратно в главную ветку. Ветвление функций выглядит следующим образом:



При магистральной разработке коммиты возвращают в главную ветку как можно чаще, даже если функция целиком еще не завершена (однако каждое *изменение* все равно должно быть завершено, что мы и покажем на примере далее в этой главе):



Инкрементная доставка функций

Ян обсуждает с Сэнди обычный ход своей работы над функцией:

1. Создание ветки функции как ответвления от главной ветки.
2. Начало разработки сквозных тестов.
3. Наполнение каркаса нового класса стримингового сервиса тестами.
4. Добавление тестов и новых классов для каждой отдельной функции.
5. Периодическое подтягивание изменений из главной ветки (если он об этом не забывает).
6. Когда все готово, слияние функции с главной веткой.

При подходе, который предлагает Сэнди, Ян по-прежнему будет создавать ветки, но их нужно будет сливать обратно с главной как можно быстрее, по возможности несколько раз в день. Поскольку этот процесс сильно отличается от того, как привык работать Ян, они подробно обсуждают детали.

Коммит с пропущенными тестами

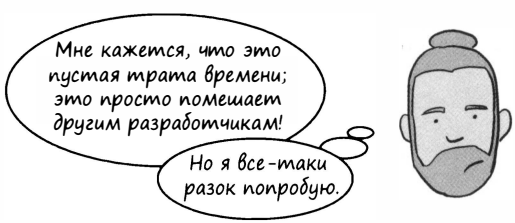
Сэнди убеждает Яна создать сквозные тесты, как он и задумывал, и хотя они не будут проходить до конца работы над функцией, закоммитить их в главную ветку как неактивные. Это позволит быстро выполнить коммит в главную ветку без необходимости держать тесты в долгоживущей ветке функций.

Ян создает набор сквозных тестов для новой интеграции с AllCatsAllTheTime. Эти тесты будут взаимодействовать с реальным сервисом AllCatsAllTheTime, поэтому он заводит тестовый аккаунт (`WatchMeWatchTest01`) и создает в нем активность просмотров, которую обрабатывают его тесты. Например, вот один из сквозных тестов, который покрывает метод `getWatchHistory`:

```
def test_get_watch_history(self):
  service = AllCatsAllTheTime(ACATT_TEST_USER)
  history = service.getWatchHistory(ALL_TIME)

  self.assertEqual(len(history), 3)
  self.assertEqual(history[0].name, «Real Cats of NYC»)
```

Тесты, конечно, не срабатывают, потому что на самом деле Ян не реализовал ни одну из функций, которые вызывают эти тесты. Ему не нравится идея, но он делает то, что предложил Сэнди, и отключает тесты с помощью `unittest.skip` с сообщением о том, что процесс реализации еще не завершен. Он включает ссылку на проблему интеграции AllCatsAllTheTime в баг-трекер (`#2387`), чтобы другие разработчики могли найти дополнительную информацию, если она им понадобится:

```
  @unittest.skip("(#2387) незавершенная интеграция
                  с сервисом AllCatsAllTheTime")
  def test_get_watch_history(self):
```



Ян большой скептик! Он что, плохой программист?

Совсем нет! Быть скептиком естественно, когда пробуешь что-то новое, особенно если у вас большой опыт работы над задачами другими способами. Главное, что Ян согласен пробовать. Вообще, готовность экспериментировать и давать новым идеям шанс — основное условие постоянного развития и совершенствования команды. И оно совсем не означает, что каждая новая идея должна немедленно всем понравиться.

Проверка кода и неполный код

Как такой подход сочетается с проверкой кода? Наверняка такие маленькие неполные коммиты, как этот, трудно проверять? Посмотрим, что произойдет!

Ян создает PR, содержащий новые пропущенные сквозные тесты, и отправляет его на проверку. Когда другой разработчик из его команды, Мелисса, приступает к проверке PR, она, по понятным причинам, сбита с толку, поскольку привыкла проверять полностью готовые функции. Ее первые слова это подтверждают:

Мелисса

Привет, Ян, я не знаю, как это проверять; этот PR не полный. Может быть, ты забыл добавить какие-то файлы?

До сих пор разработчики сайта Watch Me Watch считали PR полным, если он содержит рабочую функцию, все сопутствующие тесты (все успешные и ни одного пропущенного) и документацию для функции.

Переход к инкрементному подходу означает пересмотр понятия «*полный*». Сэнди приводит новое определение полного PR:

- Весь код соответствует проверкам линтинга.
- Докстринги (строки документации) для неполных функций поясняют, почему они неполные.
- Каждое изменение кода сопровождается тестами и документацией.
- Отключенные тесты содержат пояснение и ссылку на отслеживаемую проблему.

Сэнди и Ян объясняют Мелиссе и остальным членам команды, что они пытаются сделать, и попутно вводят новое определение понятия «*полный*». После встречи Мелисса возвращается к PR и оставляет несколько новых замечаний:

Мелисса

Думаю, теперь я поняла! Кажется, для нового инкрементного подхода осталось только обновить документацию по интеграции со стриминговыми сервисами?

Ян понимает, что Мелисса права: он добавил тесты, но не обновил документацию в репозитории, объясняющую порядок интеграции с потоковыми сервисами, поэтому он вносит в PR изменение с добавлением нескольких коротких исходных документов:

```
* AllCatsAllTheTime — (#2387) незавершенная интеграция с провайдером контента
о кошках
```

Мелисса одобряет изменения, и отключенные сквозные тесты сливаются в главную ветку.

Но разве мертвый код — это хорошо?

Многие компании вводят запрет на мертвый код, и на то есть веские причины! *Мертвый код* — это код, который зафиксирован в репозитории, но не доступен; он не вызывается никакими активно работающими путями кода. (Этим термином также может обозначаться код, который выполняется, но ничего не меняет, но здесь имеется в виду не это.)

Важно понимать, почему мертвого кода быть не должно. Основная причина связана с долговременной поддержкой кода. По мере добавления нового кода разработчики будут видеть мертвый код, и им придется как минимум его читать, чтобы понимать, что он не используется, а в худшем случае — тратить время на его обновление. *Однако* это потеря времени только в том случае, если код действительно мертвый (то есть не будет больше использоваться). В нашем случае часть кода, который добавляет Ян, может показаться мертвой, поскольку он не используется, но он просто *еще* не используется. Время, которое другие разработчики тратят на поддержку этого кода, не пропадает зря; на самом деле, как вы увидите дальше, оно чрезвычайно полезно.

Мертвый код — это плохо, только если его присутствие в проекте приводит к потере времени на его поддержку.

Тем не менее лучше не оставлять мертвый код. (Представьте, например, что Watch Me Watch решает не интегрироваться с сервисом AllCatsAllTheTime, и Ян прекращает работу над этой функцией. Если оставить этот код, он станет шумом и на него будет напрасно тратиться время.) Чтобы мертвый код не захламил проект, можно сделать следующее:

- Полностью запретить фиксацию мертвого кода (недоступного, невыполняемого кода), выявляя его до слияния и блокируя его слияние.
- Регулярно запускать автоматизированные процессы для обнаружения мертвого кода и вычищать его автоматически. Лучше всего, если средства автоматизации будут предлагать изменения для удаления кода, но не сливать их, а разработчики будут сами решать, стоит ли оставлять этот код.

Второй вариант является наиболее гибким и позволит использовать подход, похожий на тот, который использует Ян, в сочетании с мерами предосторожности, обеспечивающими удаление кода, если в итоге окажется, что он никогда не будет использован.

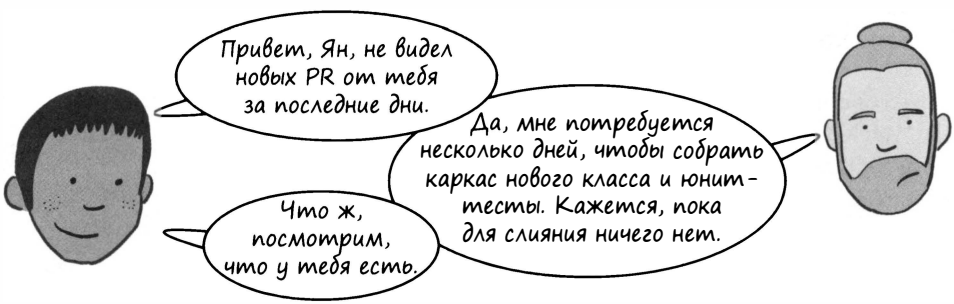
Если ваша компания выберет первый вариант, вы все равно сможете использовать инкрементный подход к разработке функций, но он будет выглядеть несколько иначе. Несколько советов о том, как это сделать, вы найдете ниже, во врезке «Вы можете коммитить чаще!».

Двигаемся дальше

Ян выполняет слияние для своих исходных (отключенных) сквозных тестов. Что дальше? Он делает почти все то же, что и при реализации новой функции, но без специально выделенной ветки функций:

1. ~~Создание ветки функции как ответвления от главной ветки~~ (Ветки функций не используются).
2. ~~Начало разработки сквозных тестов~~ (Сделано, произведено слияние с главной веткой).
3. Наполнение каркаса нового класса стримингового сервиса тестами. *Следующий шаг*
4. Добавление тестов и новых классов для каждой отдельной функции.
5. Периодическое подтягивание изменений из главной ветки (если он об этом не забывает).
6. Когда все готово, слияние функции с главной веткой.

Следующий шаг Яна — приступить к созданию каркаса нового стримингового сервиса и связанных с ним юнит-тестов. Через несколько дней Сэнди проводит проверку:



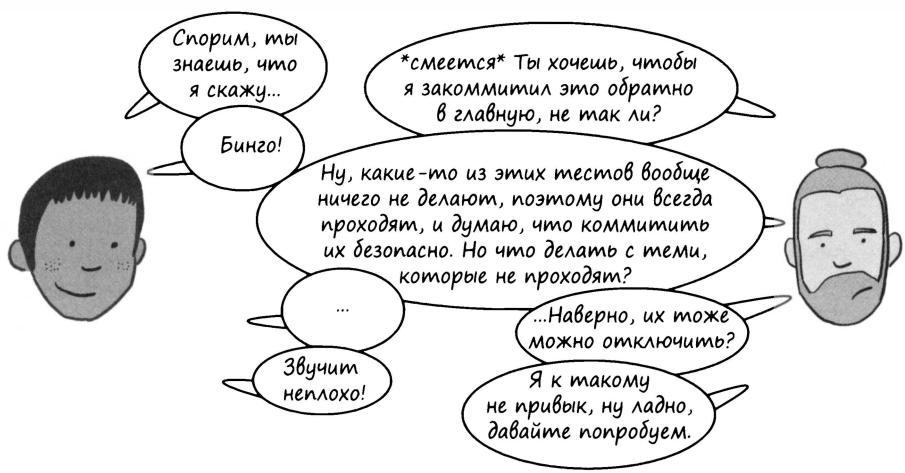
К слову, Сэнди ненавидит микроменеджмент и обычно не вмешивается в работу подчиненных!

Коммит незавершенного кода

Пока что у Яна есть несколько исходных методов для класса `AllCatsAllTheTime`:

```
class AllCatsAllTheTime(StreamingService):
  def __init__(self, user):
    super().__init__(user)

  def getCurrentlyWatching(self):
    """Получить шоу/фильмы, которые, по данным AllCatsAllTheTime,
       сейчас смотрит пользователь self.user"""
    return []

  def getWatchHistory(self, time_period):
    """Получить шоу/фильмы, которые, по данным AllCatsAllTheTime,
       смотрел пользователь self.user"""
    return []

  def getDetails(self, show_or_movie):
    """Получить все атрибуты шоу/фильма, хранящиеся на сервисе AllCatsAllTheTime"""
    return {}
```

Он также создал юнит-тесты для `getDetails` (которые пока не работают, потому что еще ничего не реализовано) и разработал несколько исходных юнит-тестов для других функций, которые совершенно пусты и всегда проходят. Он показывает эту работу Сэнди, который дает несколько комментариев:



Разве Ян не должен был сделать больше за несколько дней?

Может быть, да (а может и нет, ведь создавать макеты объектов и обеспечивать работоспособность юнит-тестов может быть очень сложно). На самом деле я сокращаю примеры, потому что мне нужно уместить их в одну главу. Описываемая здесь идея справедлива даже для небольших примеров и заключается в том, что нужно привыкнуть делать небольшие частые коммиты, даже такие маленькие, как те, которые будет делать Ян.

Ревью незавершенного кода

Ян открывает PR со своими изменениями: пустой каркас нового класса, отключенный юнит-тест, который не проходит, и несколько юнит-тестов, которые ничего не делают, но проходят. К этому моменту Мелисса понимает, почему так много PR находится в процессе разработки, и не возражает. Но тут же дает обратную связь:

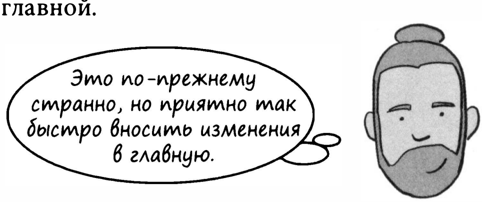
Мелисса

Можно ли добавить еще описаний в документацию? Автоматическая документация пытается добавить этот новый класс, а все докстринги практически пусты.

Ян не ожидал, что PR с таким небольшим контентом может получить полезную обратную связь. Он начинает заполнять докстринги для пустых функций, описывая, для чего они предназначены и что они делают; например, он добавляет такой докстринг в новый класс `AllCatsAllTheTime`:

```
def getWatchHistory(self, time_period):
  """
  Получить шоу/фильмы, которые, по данным AllCatsAllTheTime, смотрел пользователь self.user

  Ресурс AllCatsAllTheTime будет хранить полную историю всех шоу и фильмов,
  просмотренных пользователем с момента регистрации до текущего момента,
  поэтому эта функция возвращает результат от 0 до списка неограниченной длины.

  Интеграция AllCatsAllTheTime находится в стадии разработки (#2387), поэтому
  пока что эта функция ничего не делает и всегда возвращает пустой список.

  :param time_period: Либо значение ALL_TIME для получения полной истории просмотров, либо
  экземпляр TimePeriod, указывающий начальную и конечную дату и время для формирования
  выборки по истории
  :returns: Список шоу-объектов, по одному для каждого из просматриваемых в данный момент.
  """
  return []
```

После того как Ян обновляет PR с докстрингами, Мелисса одобряет его, и он сливается с главной.



На Watch Me Watch докстринги представлены в формате reStructuredText.

Разве Мелисса не тратит время впустую, делая ревью незавершенных изменений?

Если кратко, нет! Мелиссе гораздо проще проверять крошечные PR, чем гигантскую ветку функции! Кроме того, она может тщательнее изучить интерфейсы (например, сигнатуры методов) и дать рекомендации по ним на раннем этапе, пока они еще не полностью доработаны. Вносить изменения в код до его написания проще, чем после того, как он готов!

Вернемся к сквозным тестам

За это время в репозитории без ведома Яна, Сэнди и Мелиссы происходят другие изменения в коде. Ян создает новую ветку, чтобы начать следующий этап, и когда он открывает сквозные тесты и каркас сервиса, над которым работал, он с удивлением видит, что кто-то внес изменения в код, который он уже закоммитил!

Он замечает, что в сквозном тесте вызов метода `getWatchHistory` в классе `AllCatsAllTheTime` имеет несколько новых аргументов:

```
def getWatchHistory(self, time_period, max, index):
    ...
    :param time_period: Либо значение ALL_TIME для получения полной истории
      просмотров, либо экземпляр TimePeriod, указывающий начальную
      и конечную дату и время для формирования выборки по истории
    :param max: максимальное количество возвращаемых результатов
    :param index: указатель общего количества результатов, из которых
      нужно вернуть результаты вплоть до максимального количества.
    ...
```

В getWatchHistory добавлены аргументы для поддержки постраничного просмотра результатов.

Новые аргументы добавлены также в каркас сервиса:

```
def getWatchHistory(self, time_period, max, index):
  return []
```

Эти тесты всегда проходят, потому что тело теста еще не заполнено, но автор указал, что следует сделать в дальнейшем.

И даже есть пара новых юнит-тестов:

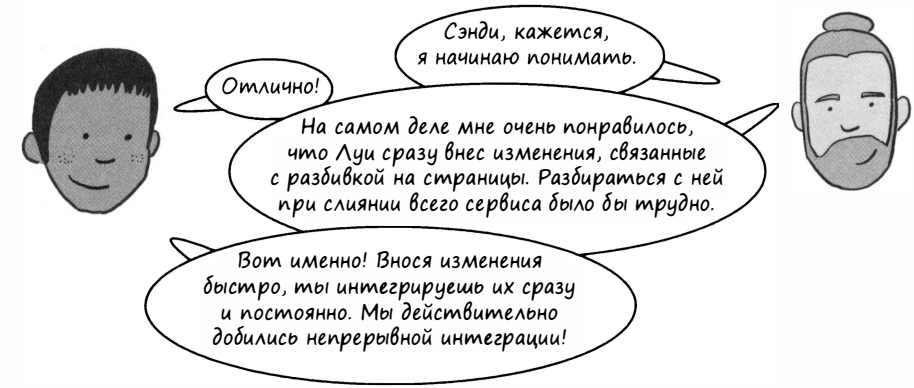
```
def test_get_watch_history_paginated_first_page(self):
  service = AllCatsAllTheTime(ACATT_TEST_USER)
  history = service.getWatchHistory(ALL_TIME, 2, 0)
  # TODO(#2387) убедиться, что возвращается первая страница результатов

def test_get_watch_history_paginated_last_page(self):
  service = AllCatsAllTheTime(ACATT_TEST_USER)
  history = service.getWatchHistory(ALL_TIME, 2, 1)
  # TODO(#2387) убедиться, что возвращается первая страница результатов
```

Просматривая историю изменений, Ян видит, что накануне Луи слил PR, который добавил разбивку на страницы в `getWatchHistory` для всех стриминговых сервисов. Он также видит сообщение от Луи в чате:

> **Луи**
>
> Спасибо, что слил класс `AllCatsAllTheTime` пораньше! Я думал, как отслеживать, чтобы незавершенные интеграции обновлялись с учетом разбивки на страницы; я не хотел, чтобы у тебя были проблемы во время слияния. Круто получать обновления сразу!

Поскольку Ян слил свой код раньше, Луи сразу же внес в него изменения. Если бы Ян оставил этот код в ветке функций, Луи не знал бы о сервисе AllCatsAllTheTime, а Ян не знал бы об изменениях в разбивке на страницы. Когда он наконец собрался бы слить эти изменения, недели или даже месяцы спустя, ему пришлось бы решать проблему конфликта изменений с Луи. Но сейчас Луи решил ее, не откладывая в долгий ящик.

Оценим преимущества



Мы начинаем выходить за рамки непрерывной интеграции и переходим к процессам, которые происходят после нее, то есть к остальной части непрерывной доставки, но на самом деле грань между ними тонкая, и решения, которые команда принимает в процессе CI, в дальнейшем влияют на весь процесс CD. Хотя главная цель Сэнди — повысить скорость, как он только что объяснил Яну, инкрементный подход означает, что процессы CI в команде теперь гораздо ближе к идеалу. В чем же заключается этот идеал? Вспомним определение *непрерывной интеграции*:

> **процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.**

При наличии долгоживущих веток функций изменения кода вносятся только тогда, когда ветки функций возвращаются в главную. Но фиксируя изменения в коде главной ветки максимально быстро, Ян часто объединяет свои изменения с содержимым главной ветки (и дает возможность другим разработчикам объединить свои изменения с его изменениями)!

ВАЖНО

Оптимизация развертывания часто в первую очередь означает повышение качества CI.

ВАЖНО

Отказ от долгоживущих веток функций и использование инкрементного подхода с частыми слияниями с главной веткой (при магистральной разработке) не только улучшает CD, но и обеспечивает более качественную CI.

Вы *можете* коммитить чаще!

Если, глядя на подход Яна и Сэнди, вы говорите себе: «В моем проекте это никогда не сработает», — не отчаивайтесь! Это не единственный вариант, позволяющий избежать веток функций и часто сливать поэтапную работу. Чтобы использовать инкрементный подход без фиксирования незавершенного кода, рассмотрите принцип, который реализуют Ян и Сэнди:

Разработка и развертывание становятся проще и быстрее, если вносить изменения в код как можно чаще.

Цель — коммитить изменения обратно в репозиторий как можно чаще, и в вашей ситуации может быть приемлемо то, что не подходит для данного проекта. Самое эффективное — разбить работу на части, каждую из которых можно выполнить меньше чем за один день. В любом случае это хорошая практика разработки, так как она облегчает планирование работы и упрощает взаимодействие с другими сотрудниками (например, позволяет нескольким членам команды работать над одной функцией).

Что и говорить, это сложно! И может показаться, что проще с этим не связываться. Вот несколько рекомендаций, которые могут помочь создавать небольшие, частые пул-реквесты (подробнее см. в литературе о методологиях программной разработки, таких как agile):

- Приступая к работе, отбросьте все непонятные моменты и начните с черновой *проверки концепций* (POCs, Proof of Concepts), а не пытайтесь сразу писать готовое к выпуску ПО и одновременно изучать новую технологию (например, Ян мог бы создать POC-интеграцию с сервисом AllCatsAllTheTime, прежде чем начинать работать с кодом стриминговой интеграции).
- Разбейте работу на отдельные задачи, каждая из которых займет несколько часов или максимум день. Продумайте, какие небольшие, независимые PR можно для них создать (каждый из PR будет содержать документацию и тесты).
- Проводите рефакторинг в отдельном PR и быстро сливайте его.
- Если вы не можете избежать одной большой ветки функций, по ходу работы следите за ее частями, которые можно закоммитить, и найдите время на создание и слияние отдельных PR для них.
- Используйте *флаги функций*, чтобы предотвратить доступ пользователей к незавершенным функциям, и/или *флаги сборки*, чтобы вообще предотвратить их компиляцию.

Подведя итог: выделите время и заранее продумайте, как разбить работу на части и быстро закоммитить ее обратно, а также уделите внимание созданию небольших независимых PR-запросов, необходимых для поддержки этого процесса. Ваши усилия окупятся!

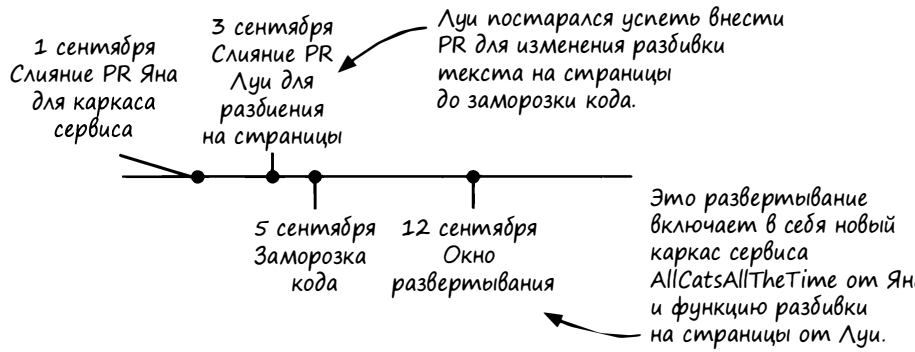
Сокращение времени доставки изменений

Приближая CI к идеалу, Сэнди и Ян изменяют весь процесс CD. В частности, их действия положительно влияют на DORA-метрики Watch Me Watch. Вспомните цели Сэнди:

- *Частота развертывания* — перейти от среднего показателя к высокому, выполняя развертывания не реже одного раза в месяц.
- *Время доставки изменений* — перейти от среднего к высокому результату, сократив время доставки изменений с 45 дней до одной недели или меньше.

Самый последний PR Яна (включающий каркас нового класса для стримингового сервиса и несколько готовых юнит-тестов) был создан всего за несколько дней до заморозки кода и открытия окна развертывания. В результате код новой интеграции Яна попал на продакшен как часть этого развертывания.

Конечно, новый код интеграции еще ничего не делает, но, несмотря на это, изменения, которые вносит Ян, попадают на продакшен. Сэнди оценил время внесения этих изменений:



Ян слил каркас класса за 4 дня до заморозки кода. За 2 дня до заморозки кода Луи обновил метод `getWatchHistory`, чтобы он принимал аргументы разбиения на страницы. Заморозка кода произошла через 2 дня, а через неделю после этого началось развертывание.

Отсчет полного времени доставки изменений в каркас класса начался с момента, когда Ян произвел слияние, 1 сентября, и закончился развертыванием 12 сентября, общее время составило 11 дней.

Сравним его с временем внесения изменений, над которыми работал Луи. Он работал в ветке функции с момента последнего окна развертывания, которое было 12 июля. Он начал работу 8 июля, поэтому полное время доставки изменений составило с 8 июля по 12 сентября — 66 дней.

Хотя изменения Яна являются инкрементными (и пока не работают), Ян смог сократить время внесения каждого отдельного изменения до 11 дней, в то время как изменения Луи пришлось ждать 66 дней.

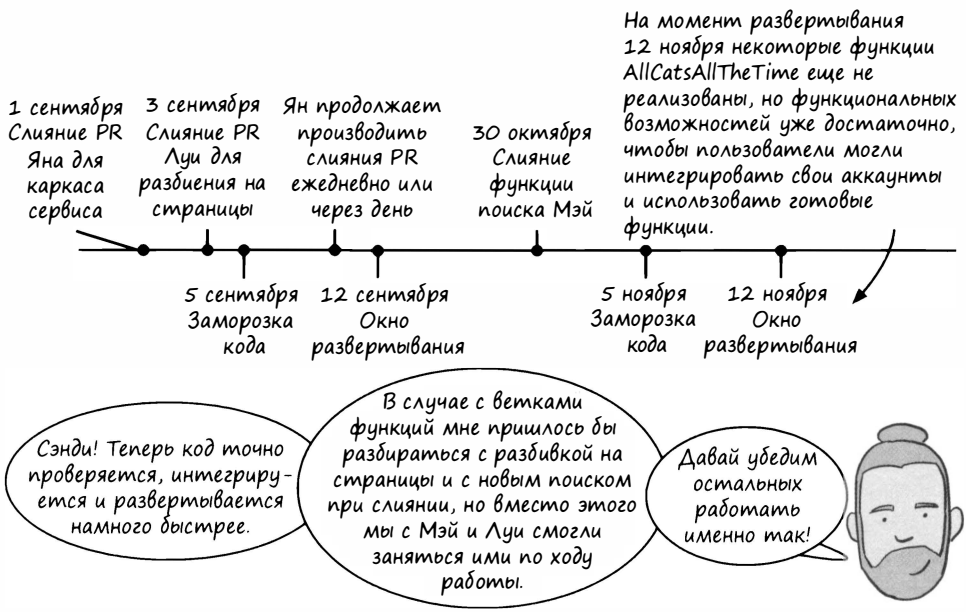
Продолжим работу над AllCatsAllTheTime

Ян продолжил работать с Сэнди, применяя инкрементный подход при реализации остальной интеграции в системе AllCatsAllTheTime. Он реализовывал метод за методом, разрабатывая их, расширяя набор юнит-тестов и добавляя сквозные тесты по ходу работы. Через несколько недель другая участница команды, Мэй, работая над функцией поиска, добавляет новый метод в интерфейс `StreamingService`:

```
class StreamingService:
    ...
  @staticmethod
  def search(show_or_movie):
    pass
```

Этот новый метод позволит пользователям искать конкретные фильмы и шоу по всем стриминговым провайдерам, и автор изменения добавляет его во все существующие интеграции. Поскольку Ян в рамках инкрементного подхода постоянно коммитил класс `AllCatsAllTheTime`, Мей может добавить метод поиска в уже имеющийся класс, даже не сообщая Яну об изменениях! В один из дней Ян создает новую ветку, чтобы начать работу над методом `getDetails`, и видит код, который добавила Мэй.

Разбивка на страницы и поиск — это две самые важные функции, которые были интегрированы с изменениями Яна по мере их разработки и с которыми в обычной ситуации Яну пришлось бы разбираться при слиянии. Кроме того, следующее развертывание (12 ноября), несмотря на то что интеграция еще не завершена, добавляет достаточно функционала, чтобы пользователи могли его использовать, а маркетологи — рекламировать.



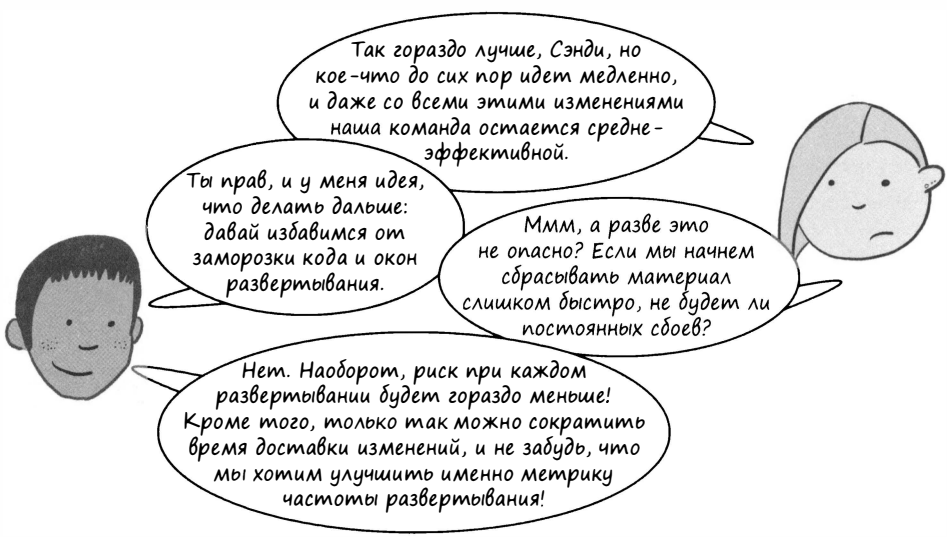
Окна развертывания и заморозка кода

Сэнди и Ян представили Саше и Саре результаты своего эксперимента. Они показали, что отказ от долгоживущих веток функций и инкрементное слияние функций неожиданно оказались очень полезными:

- Сократилось время доставки изменений.
- Множество функций теперь можно интегрировать быстрее и проще.
- Пользователи получают доступ к функциям раньше.

Саша и Сара соглашаются применить эту политику во всей компании и посмотреть, что получится, поэтому Сэнди и Ян начинают обучать остальных разработчиков, как отказаться от веток функций и использовать инкрементный подход.

Через несколько месяцев Сэнди проверяет время внесения всех изменений, чтобы понять, насколько оно снизилось. Среднее время доставки изменений значительно сократилось — с 45 до 18 дней. Отдельные изменения стали быстрее попадать в главную ветку, но они по-прежнему блокируются при заморозке кода, и если их слияние происходит вскоре после развертывания, с их внедрением приходится ждать почти 2 месяца до следующего развертывания. Хотя этот показатель DORA улучшился, он все еще не соответствует цели Сэнди — улучшить время доставки изменений и перейти от среднего результата к высокому (к одной неделе или менее).



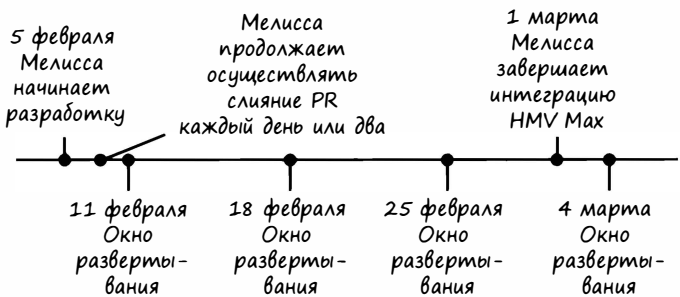
Они обсуждают этот план и соглашаются проводить еженедельные развертывания и полностью отказаться от заморозки кода.

> Подробнее о развертывании, минимизации рисков и способах безопасного развертывания см. в главе 10.

Повышение скорости

Сэнди отслеживает метрики в течение следующих нескольких месяцев и наблюдает за разработкой функций, чтобы понять, ускоряется ли работа и каковы показатели DORA при отсутствии заморозки кода и более частом развертывании.

Мелисса работает над интеграцией с новым стриминговым провайдером Home Movie Theatre Max. На полную реализацию интеграции у нее уходит около 5 недель, и за это время происходит четыре развертывания, каждое из которых включает часть ее изменений.



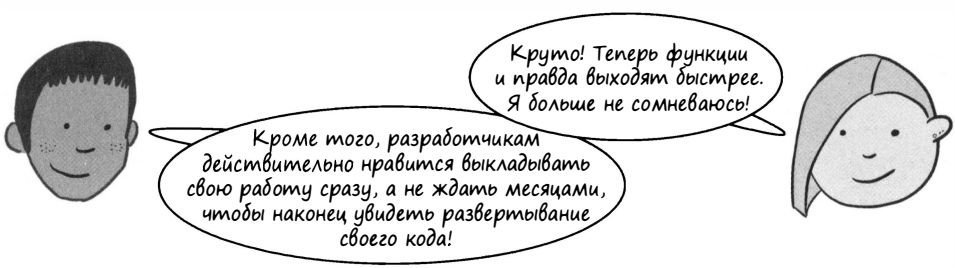
Время доставки изменений Мелиссы составляет максимум 5 дней. Некоторые изменения развертываются уже через день после слияния.

Сэнди просматривает общую статистику Watch Me Watch и обнаруживает, что максимальное время доставки изменений составляет 8 дней, но это редкость, поскольку большинство разработчиков привыкли сливать изменения в главную ветку раз в один-два дня. Средние показатели при этом такие:

- *Частота развертывания* — один раз в неделю.
- *Время доставки изменений* — 4 дня.

Добиться высокой производительности с помощью этой метрики было так же просто, как изменить интервалы между окнами развертывания.

Сэнди достиг цели: по метрикам DORA для скорости Watch Me Watch сравнялась с высокоэффективными компаниями!



Теперь Саша, Сара и Сэнди думают, как из высокоэффективных исполнителей стать суперэффективными, но я приберегу это до главы 10!

Заключение

Компания Watch Me Watch внедрила заморозку кода и редкие окна развертывания, чтобы сделать разработку более безопасной. Однако это лишь замедлило процесс. Оценив свою работу по метрикам DORA, в частности, связанным со скоростью, разработчики смогли наметить пути ускорения.

Отказ от долгоживущих веток функций и от заморозки кода и увеличение частоты развертывания сразу позволили улучшить метрики DORA. Благодаря новому подходу компания избавилась от ощущения, что работа над функциями занимает все больше и больше времени и конкуренты ее опережают, а разработчики стали получать больше удовлетворения от выполненных задач.

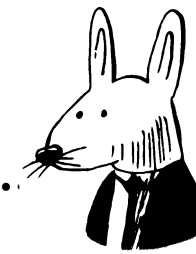
Итоги

- Команда DevOps Research and Assessment (DORA) определила четыре ключевые метрики для измерения производительности команд разработчиков и разбивки их по четырем категориям: низкоэффективные, среднеэффективные, высокоэффективные и суперэффективные.
- Частота развертывания, одна из двух связанных со скоростью метрик DORA, измеряет частоту развертывания на продакшен.
- Другая связанная со скоростью метрика DORA — это время доставки изменений, которое характеризует время от завершения изменения до его запуска на продакшен.
- Сокращение времени доставки изменений требует пересмотра и оптимизации методики CI. Чем лучше CI, тем меньше время доставки изменений.
- Оптимизация методики CD за пределами CI часто предполагает пересмотр самой CI.
- Частота развертывания напрямую влияет на время доставки изменений; увеличение частоты развертывания, скорее всего, приведет к сокращению этого времени.

Далее...

В следующей главе мы рассмотрим главное преобразование, которое происходит с исходным кодом в пайплайне CD: сборку исходного кода в конечный артефакт, который будет выпущен (и, возможно, развернут), а также способы безопасной сборки этого артефакта.

Безопасная и надежная сборка | 9



В этой главе

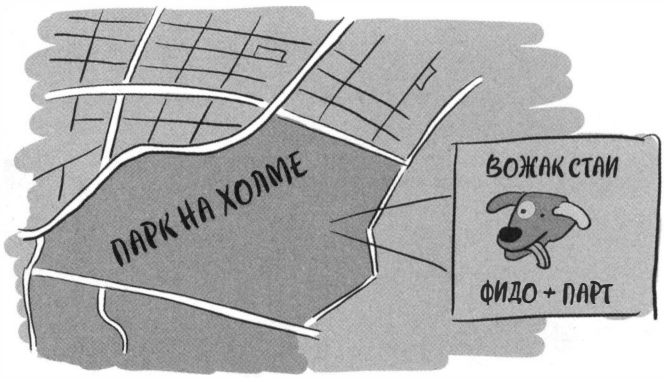
- ✓ Безопасное проведение сборки путем ее автоматизации, использования подхода «конфигурация как код» и запуска сборки на специализированном сервисе с поддержкой временных сред
- ✓ Однозначная и уникальная идентификация артефактов с помощью семантического версионирования (SemVer) и хешей
- ✓ Исключение непредвиденных ошибок из зависимостей благодаря привязке последних к определенным уникальным версиям

Сборка — неотъемлемая часть процесса непрерывной доставки (CD), так что задачи и пайплайны CD часто называют *сборками*. Наша любимая автоматизация процессов возникла из автоматизации сборки программных артефактов из их исходного кода в форму, в которой они используются и распространяются.

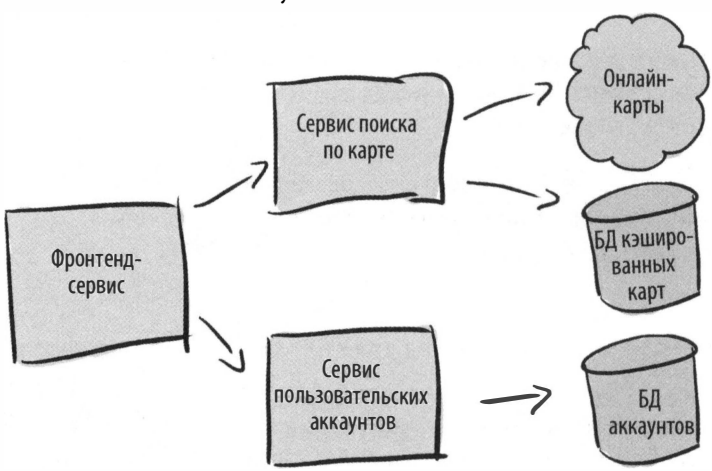
В этой главе мы расскажем о типичных «подводных камнях» при сборке программных артефактов, которые приводят к появлению багов и усложняют жизнь конечным пользователям этих артефактов, а также о том, как создавать такие пайплайны CD, которые позволят полностью избежать этих проблем.

Веб-сайт Top Dog Maps

Компания Top Dog Maps запускает сайт по поиску парков с площадками для выгула собак. Пользователи могут находить парки в определенном районе, оценивать их и даже отмечать их посещения. Пользователь с наибольшим количеством посещений становится вожаком стаи парка.



Архитектура сайта Top Dog Maps состоит из нескольких сервисов, за каждый из которых отвечает отдельная команда: фронтенд-сервис, сервис поиска по карте и сервис пользовательских аккаунтов.



Каждый сервис распространяется и запускается как образ контейнера, а установкой сервисов управляет команда разработчиков фронтенда. Команды поиска по карте и пользовательских аккаунтов выпускают образы контейнеров сборки, а команда фронтенда решает, когда и где их устанавливать, чтобы они получали обновления в удобное время.

В команде поиска по карте недавно произошли перестановки: Джулия, отвечавшая за создание образа контейнера для сервиса поиска по карте, недавно перешла в другую компанию, в результате чего в команде образовалась вакансия.

Когда процесс сборки — это документ

После того как Джулия покинула компанию Top Dog Maps, кто-то из команды поиска по карте должен взять на себя создание образа для этого сервиса. К счастью, Мигель очень хочет улучшить процесс сборки и тут же вызывается выполнять эту работу.

Первым делом он оценивает, как происходит создание образа для поиска по карте. Он быстро выясняет, что сборка выполняется в соответствии с инструкциями, которые Джулия написала в документе:

Создание образа сервиса поиска по карте

Шаг 1:
Клонируйте репозиторий сервиса поиска по карте:
```
$ git clone git@github.com:topdogmaps/map-search.git
```

Шаг 2:
Замените каталог репозиторием:
```
$ cd map-search
```

Шаг 3:
Создайте образ контейнера:
```
$ docker build --tag top-dog-maps/map-search .
```

Шаг 4:
Поместите образ в реестр Top Dog Maps:
```
$ docker push top-dog-maps/map-search
```

Вы уже заметили, что у этого образа нет тега версии? Не волнуйтесь, я обязательно доберусь до этого! А пока у Мигеля есть дела поважнее.

Мигель сразу же обнаруживает в процессе несколько проблем:

- Процесс зависит от человека, который должен прочитать документ и правильно выполнить каждую инструкцию. Это чревато ошибками; легко случайно пропустить какое-либо указание или сделать опечатку.
- Шаги инструкции могут выполняться где угодно: на машине Джулии, на машине Мигеля и еще бог знает где! Согласованности нет, то есть сборка даже одного и того же исходного кода может привести к разным результатам, если шаги выполняются на разных машинах.

Атрибуты безопасных и надежных сборок

Работая над планом улучшения процесса сборки, Мигель составляет список атрибутов безопасных и надежных сборок, которыми, по его мнению, должен обладать сервис поиска по карте, а в перспективе — и остальные сервисы Top Dog Maps:

Постоянная готовность к релизу	Исходный код всегда должен находиться в состоянии готовности к релизу
Автоматические сборки	Сборки должны выполняться автоматически, а не вручную
Сборка как код	Конфигурация сборки должна рассматриваться как код и храниться в системе контроля версий
Использование сервиса CD	Сборки должны происходить через сервис CD, а не на случайных машинах, таких как рабочая станция разработчика
Временные среды	Сборки должны происходить во временных средах, которые создаются и сносятся при каждой сборке

Словарик

Под *сборкой*, или *билдом*, имеется в виду задача (задачи), которая осуществляет сборку программных артефактов из исходного кода.

Стандарты безопасного создания артефактов CD: SLSA

Перечисленные выше требования заданы недавно появившимся набором стандартов для безопасного создания программных артефактов. Эти стандарты под названием «*Уровни цепочки поставок для программных артефактов*» (*Supply Chain Levels for Software Artifacts*), или сокращенно SLSA (произносится как «сальса»), были созданы на основе внутренних стандартов Google, используемых для обеспечения безопасности рабочих нагрузок приложений. SLSA (https://slsa.dev) определяют ряд уровней безопасности при сборке артефактов, переход на которые осуществляется постепенно (инкрементно). Требования, которые здесь обсуждаются, взяты из требований к процессу сборки для достижения уровня 3 версии SLSA 0.1. Если вас интересует безопасность цепочки поставок ПО, уровни SLSA — это то, что нужно, чтобы постепенно перейти от текущего состояния процесса сборки к более и более безопасным процессам.

Герметичные и воспроизводимые сборки

Если ваш процесс сборки соответствует вышеперечисленным требованиям, он соответствует требованиям уровня 3 SLSA версии 0.1 для сборок. (Обратите внимание, что в общем случае в SLSA существуют еще дополнительные требования к уровню 3; но здесь мы сосредоточились только на требованиях к сборке.)

Если вы хотите повысить безопасность процесса, рассмотрите далее требования SLSA для уровня 4, в особенности касающиеся обеспечения *герметичности* сборок (процесс сборки не должен иметь доступа к внешней сети и подвергаться внешним воздействиям, кроме собственных входных данных) и их *воспроизводимости* (все сборки с одними и теми же входными данными должны давать абсолютно одинаковый результат на выходе).

Это благие цели, к которым стоит стремиться, но для их достижения может потребоваться немало усилий, особенно учитывая, что текущий уровень инструментария CD не отличается широкой поддержкой герметичных и/или воспроизводимых сборок. Даже если вы используете сборки на основе контейнеров, большинство систем CD не гарантируют герметичность их выполнения (например, нельзя осуществить блокировку доступа к сети изнутри контейнера), и в них слишком легко создать невоспроизводимую сборку (ведь даже такие незначительные различия, как метки времени в результирующем артефакте, нарушают требование воспроизводимости).

Учитывая все сказанное, мониторьте тренды и следите за появлением дополнительных функций в этой области. А пока просто выполняйте перечисленные выше требования!

Постоянная готовность к релизу

Рассмотрим каждое из требований, сформулированных Мигелем, более подробно, и начнем с поддержки исходного кода в *состоянии постоянной готовности к релизу*. Поддержанию кода в готовом к выпуску состоянии мы посвятили бóльшую часть книги, в частности, изучая, как этому может помочь применение непрерывной интеграции (CI). Помните, что непрерывная интеграция — это

> **часто повторяющийся процесс внесения изменений в код с проверкой каждого изменения при внесении.**

Если каждое изменение проверяется при фиксации в коде (подробнее о том, в какой именно момент проводить проверку, читайте в главе 7), программный продукт остается всегда готовым к релизу.

К счастью, у команды поиска по карте уже есть надежный пайплайн непрерывной интеграции, который запускается при каждом изменении (и включает юнит-тесты, системные тесты и линтинг), поэтому команда абсолютно уверена, что новый образ сервиса можно создать в любое время.

Команда разработчиков сервиса поиска по карте поддерживает код в состоянии готовности к релизу с помощью пайплайна, включающего линтинг, системные и юнит-тесты.

ВАЖНО

Используйте непрерывную интеграцию (как описано в главах 3–7), чтобы поддерживать код в готовом к выпуску состоянии.

Автоматические сборки

Второе требование, которое сформулировал Мигель, — это использование *автоматических сборок*. Его, в свою очередь, можно разбить на два следующих требования:

- Все этапы, необходимые для сборки артефакта, должны быть определены в *скрипте*.

- Выполнение этого скрипта должно инициироваться автоматически, без участия человека.

Сравним их с неавтоматизированной сборкой:

- Предполагается, что создатель артефакта «просто знает», что нужно делать (то есть последовательность шагов держится в голове и передается «из уст в уста»).
- Шаги записаны в документе, который кто-то должен прочитать и действовать в соответствии с написанным.
- Разрабатывается скрипт для сборки артефакта, но при этом требуется, чтобы кто-то запустил его вручную для создания артефакта.

То, чем занимается команда разработчиков сервиса поиска по карте для создания образа контейнера, противоречит обоим требованиям, предъявляемым к автоматическим сборкам: этапы сборки определены в документе, который нужно прочитать, и предполагается, что кто-то будет выполнять каждый из этих шагов вручную.

ВАЖНО

Чтобы осуществлять автоматические сборки, определите все шаги сборки в скрипте и запускайте его автоматически.

Словарик

Мы используем слово «*скрипт*» (или «*сценарий*») для обозначения серии инструкций, которые выполняются программными средствами (а не человеком). Языки сценариев обычно интерпретируемые: каждая команда понимается и выполняется как есть, а не требует предварительной компиляции. В примерах этой главы в качестве языка сценариев мы будем использовать bash. О том, как эффективно использовать скрипты в задачах и пайплайнах, см. главу 12.

Сборка как код

Третье требование, описанное Мигелем, заключается в том, что следует работать со сборкой как с кодом. Понятие «*сборка как код*» — это отсылка к идее «*конфигурация как код*» (см. главу 3). Обработка конфигурации как кода означает следующее:

Хранить все текстовые данные, описывающие программный продукт, в системе контроля версий.

Понятие «*сборка как код*» — это просто оригинальный способ сказать, что конфигурация сборки (и любые необходимые скрипты) является частью текстовых данных, описывающих программный продукт: оно определяет, каким образом продукт из исходного кода становится готовым артефактом.

Вследствие этого, чтобы рассматривать конфигурацию сборки как код, ее необходимо хранить в системе контроля версий и там, где это возможно, применять техники CI, которые были описаны для исходного кода, включая линтинг и даже тестирование, если конфигурация достаточно сложная. Но если не углубляться столь сильно, то суть этого требования в том, чтобы хранить конфигурацию сборки в системе контроля версий вместе с кодом, который она собирает.

Вы хотите сказать, что для сборок нужно писать тесты?

Зависит от обстоятельств! Мы поговорим об этом подробнее в главе 12, но забегая вперед, скажу: как только скрипт превышает несколько простых строк, его нелишне протестировать, как и любой другой код.

Признаки того, что вы *не* рассматриваете сборку как код:

- Установка и настройка сборок происходит исключительно через веб-интерфейс.
- Инструкции по сборке и скрипты хранятся в документации.
- Скрипты для управления сборкой составляются, но хранятся они не в системе контроля версий, а, например, на вашей машине или на общем диске.

Поскольку команда сервиса поиска по карте хранит все инструкции по сборке в документе (который кто-то должен прочитать и соблюдать), она (пока!) не обрабатывает сборку как код.

ВАЖНО

Чтобы реализовать на практике подход «сборка как код», храните конфигурацию сборки и скрипты в системе контроля версий.

Использование сервиса непрерывной доставки

Следующее требование, сформулированное Мигелем, заключается в том, что для безопасных сборок необходимо использовать сервис CD, чтобы обеспечить согласованность способов запуска и выполнения сборки.

Сервис CD, выполняющий сборки, может создаваться, размещаться и работать внутри компании или представлять собой экземпляр стороннего решения, такого как GitLab, размещенный внутри компании; либо это может быть полностью

облачный сервис, размещенный во внешней среде. Важно то, что сборка происходит в *системе CD*, которая размещается и предлагается как действующий сервис (обзор систем CD см. в приложении А). Этот подход разительно отличается от подходов, *не* использующих сервисы CD:

- Отсутствие требований к источникам артефактов сборки.
- Запуск сборок на собственной рабочей станции разработчика.
- Запуск сборок на определенной рабочей станции (например, на компьютере под чьим-то столом) или на случайной виртуальной машине (VM, virtual machine) в облаке, созданной специально для сборки.
- Запуск сборок на рабочих серверах, на которых будут запускаться артефакты.

Если ваша компания размещает и использует собственный сервис CD, с ним следует обращаться как с любым другим рабочим сервисом, то есть запускать его на оборудовании, сконфигурированном и администрируемом с использованием тех же лучших практик, что и рабочее ПО (например, практики «конфигурация как код»).

Команда сервиса поиска по карте не предъявляет вообще никаких требований к месту сборки, и скорее всего, Джулия до сих пор выполняла сборку на своей машине, поэтому этот процесс в команде определенно не соответствует очередному целевому критерию.

ВАЖНО

Чтобы обеспечить согласованность сборок, запускайте их в сервисе CD.

Плохо ли Джулия выполняла свою работу?

Похоже, что процесс сборки, которого придерживалась Джулия, не соответствует почти ни одному критерию, которые сформулировал Мигель. Означает ли это, что Джулия плохо справлялась со своей работой? Действительно, до сих пор процесс сборки не был безопасным или надежным, но ведь он работал, не так ли? Зачастую CD представляет собой цикл настройки процессов, — которые на момент своего запуска казались вполне подходящими, даже если в них и были изъяны, — и их дальнейшего пересмотра, как только они становятся приоритетными. Правда в том, что идеальных процессов не существует и процессы сборки всегда будут (и должны) развиваться с течением времени. Процесс Джулии работал достаточно хорошо (а зачастую это все, что нужно), и до сих пор его не требовалось пересматривать. Не позволяйте лучшему стать врагом хорошего!

Временные среды для сборки

Последнее требование, которое обозначил Мигель, заключается в том, что сборки должны происходить во *временных средах*. Слово «*временные*» (иногда говорят *эфемерные*, ephemeral) здесь означает «длящиеся очень короткое время», и временные среды именно таковы: это среды, которые существуют очень недолго. В частности, они существуют *только в течение времени сборки* и никогда не используются повторно.

Эти среды сносятся и уничтожаются после завершения сборки и не используются ни для чего другого до сборки. Они могут создаваться по запросу или объединяться в пул, чтобы сразу быть готовыми к использованию; главное, чтобы они использовались для одной и только для одной сборки. Временные среды обычно не применяются в следующих случаях:

- одна физическая машина для нескольких сборок (например, компьютер под чьим-то столом);
- создание виртуальной машины и использование ее для нескольких сборок.

Является ли сборка на виртуальной машине более безопасной, чем в контейнере?

Контейнеры используют часть базовой операционной системы совместно друг с другом и со своим хостом, поэтому существует небольшая вероятность перекрестного загрязнения. Однако ее можно свести к минимуму, запуская контейнеры с минимально необходимыми правами доступа и запретив запись в те части файловой системы, которые могут повлиять на другие контейнеры.

Если среда сборки не будет использоваться повторно, значит, что-то, что способно повлиять на ее процесс, никогда не попадет из одной сборки в другую. Этого же эффекта при повторном использовании среды можно попытаться добиться, применяя ее автоматическую очистку между сборками, но это рискованно, поскольку всегда есть вероятность что-то упустить. Работа во временной среде гарантированно исключит перекрестное загрязнение сборок.

Создать временную среду проще всего, используя новую виртуальную машину или новый контейнер для каждой сборки. Контейнеры все чаще применяются для временных сборок, потому что (по сравнению с виртуальными машинами) их очень быстро можно запускать и удалять.

Как вы уже заметили, процесс сборки для сервиса поиска по карте сейчас можно запустить где угодно, поэтому он определенно не отвечает требованию временной среды.

Словарик

Среда сборки — это окружение, в котором происходит сборка (то есть преобразование исходного кода в программный артефакт). Сюда входят ядро, операционная система, установленные программы и файлы. С другой стороны, *среду сборки* можно представить как все файлы и программы, которые доступны для использования в процессе сборки и могут повлиять на конечный артефакт. Примерами распространенных сред сборки являются контейнеры и виртуальные машины, или же это может быть целый компьютер (он же «голое железо»), если не используется виртуализация или контейнеризация.

План Мигеля

Определив требования к безопасной и надежной сборке, Мигель оценивает процесс сборки сервиса поиска по карте на соответствие этим требованиям:

Постоянная готовность к релизу	Да — сервис поиска по карте уже следует передовой практике CI и проверяет каждое изменение при регистрации с помощью линтинга, системных и юнит-тестов
Автоматические сборки	Нет — сборки выполняются вручную
Сборка как код	Нет — шаги сборки описаны в документе, который должен использовать человек
Использование сервиса CD	Нет — скорее всего, сборки выполняются на машине Джулии
Временные среды	Нет — среда, в которой происходит сборка, не контролируется

Им еще многое предстоит сделать, но Мигелю не терпится начать, потому что после того, как он определил все критерии и оценил текущий процесс в соответствии с ними, у него появилось четкое представление, как двигаться дальше. Его план включает два этапа:

1. Переход от рукописных инструкций в документации к скрипту, размещенному в системе контроля версий. Это позволит реализовать принцип «сборка как код» и обеспечит как минимум половину условий для автоматизации сборок, то есть сборку в виде скрипта.
2. Переход от сборки на машине разработчика к использованию сервиса CD и выполнению сборок в контейнерах. Это обеспечит выполнение оставшейся части требований: вторую половину автоматизации сборок (автоматический запуск), использование сервиса CD и, наконец, временную среду сборки.

От документации к скрипту в системе контроля версий

Первый шаг, который нужно сделать, чтобы преобразовать процесс сборки нашего сервиса поиска по карте, — это переход от ручных инструкций в документации к скрипту, зарегистрированному в системе контроля версий. К счастью, Джулия облегчила Мигелю эту задачу, потому что хотя она и не дошла до создания скрипта, она записала в своем документе все команды для возможного скрипта:

```
Создаем образ сервиса поиска по карте

Шаг 1:
Клонируем репозиторий сервиса поиска по карте:
$ git clone git@github.com:topdogmaps/map-search.git

Шаг 2:
Меняем каталог на репозиторий:
$ cd map-search

Шаг 3:
Создаем образ контейнера:
$ docker build --tag top-dog-maps/map-search .

Шаг 4:
Помещаем образ в реестр Top Dog Maps:
$ docker push top-dog-maps/map-search
```

Используя документ Джулии в качестве отправной точки, Мигель создает первоначальный bash-скрипт:

```
#!/usr/bin/env sh
set -xe

cd "$(dirname "$0")"

docker build --tag top-dog-maps/map-search .
docker push top-dog-maps/map-search
```

Некоторые лучшие примеры из bash-шаблонов.

Шаги 3 и 4 из документа Джулии.

Поскольку этот скрипт будет находиться в том же репозитории, что и исходный код, Мигель не включает в него шаги 1 и 2, но они обязательно появятся на следующем этапе. Он коммитит скрипт в репозиторий, и первый этап завершен. Содержимое репозитория выглядит следующим образом:

```
.pylintrc
Dockerfile
map/
search/
test/
build.sh
app.py
README.md
requirements.txt
```

Образ создается на основе этого Dockerfile[1]. Он также может находиться в отдельном каталоге в репозитории — это нормально! Главное, чтобы конфигурация для сборки кода хранилась вместе с кодом.

Новый скрипт Мигеля, заменяющий документ Джулии.

[1] Dockerfile — это файл Docker, который содержит инструкции для сборки образа. — *Примеч. пер.*

Автоматические контейнерные сборки

Сценарий помог пройти часть пути, но Мигель не собирается долго отдыхать! Дальше ему нужно выбрать логику сборки:

- запускается автоматически;
- запускается посредством сервиса CD;
- запускается в контейнерах.

На сайте Top Dog Maps уже используется GitHub Actions для CI, поэтому Мигель решает использовать GitHub Actions и для автоматизации сборки сервиса поиска по карте:

```
name: Build Map Search Service
on:
  push:
    branches: [main]
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    container: docker:20.10.12
    steps:
      - uses: actions/checkout@v2
      - name: Run Build Script
        run: ./build.sh
```

Задача будет запускаться при переносе в главную ветку, то есть при каждом слиянии PR.

Задача checkout выполняет шаги 1 и 2 из документа Джулии, явно проверяя репозиторий.

На этом шаге будет запущен скрипт build.sh, который Мигель только что создал внутри контейнера. Мы опускаем детали аутентификации, которые Мигель также должен был включить, чтобы отправить только что созданный образ.

Этим изменением Мигель завершил второй этап обновления процесса сборки сервиса поиска по карте!

Конфигурация рабочего процесса GitHub Actions должна быть зафиксирована в репозитории, чтобы GitHub подхватил ее (а также потому, что ее в любом случае необходимо внести в репозиторий, чтобы выполнить условие «сборка как код»), поэтому содержимое репозитория теперь выглядит следующим образом:

```
.github/
  workflows/
    build.yml
.pylintrc
Dockerfile
map/
search/
test/
build.sh
app.py
README.md
requirements.txt
```

Этот файл YAML содержит предшествующую конфигурацию GitHub Actions в том месте, где GitHub ожидает ее найти.

Автоматизация сборки с помощью Github Actions

Нужно ли создавать скрипт для контейнерных сборок, как это сделал Мигель?

Скорее всего, вам это не понадобится. Популярные системы CD часто включают специализированные и повторно используемые задачи, например Actions в GitHub Actions.

Мигель мог использовать специфичную для Docker задачу GitHub, чтобы не создавать свой собственный скрипт, проверяющий исходный код и осуществляющий его сборку и отправку.

Однако Мигель продвигается постепенно (инкрементно), внедряя изменения по одному, — это надежно. Следующим логическим изменением этой конфигурации будет использование официальной задачи GitHub Action для Docker и, возможно, полная замена скрипта. (А чем меньше собственных скриптов, тем лучше! Подробнее о скриптах см. в главе 12.)

А что, если я не использую GitHub Actions?

Разумеется, ничего страшного! GitHub Actions приведен здесь просто в качестве примера. Какую бы систему CD вы ни использовали (некоторые варианты см. в приложении А), главное, чтобы она соответствовала требованиям Мигеля.

По возможности выбирайте систему, поддерживающую реализацию на основе контейнеров — стандарт изолированного выполнения задач. Реализация на базе виртуальных машин также соответствует требованиям изолированности, но использование только виртуальных машин приводит к замедлению времени выполнения и к стремлению выполнять несколько задач на одной машине, чтобы его избежать. Как следствие, задачи начинают включать в себя слишком много действий и их становится сложно поддерживать или повторно использовать.

Безопасная и надежная сборка

Мигель завершил оба этапа: создал скрипт, настроил его на запуск с помощью GitHub Actions и внес его в GitHub — и теперь заново оценивает процесс работы команды:

Постоянная готовность к релизу	Да — сервис поиска по карте уже следует лучшим практикам CI и проверяет каждое изменение при регистрации с помощью линтинга, системных и юнит-тестов
Автоматические сборки	Да — шаги сборки определены в скрипте, который запускается с помощью GitHub Actions всякий раз, когда главная ветка находится в рабочем состоянии
Сборка как код	Да — и скрипт, содержащий шаги сборки, и конфигурация GitHub Actions (то есть полная конфигурация сборки) фиксируются на GitHub
Использование сервиса CD	Да — используется GitHub Actions
Временные среды	Да — по умолчанию GitHub Actions будет выполнять каждое задание на новой виртуальной машине, а Мигель еще и использует новый контейнер внутри нее

Мигель успешно справился с поставленными задачами. Он уверенно сообщает другим командам Top Dog Maps, что сборка сервиса поиска по карте стала безопасной и надежной!

Нужно ли выпускать релиз при каждом коммите, чтобы осуществлять автоматические сборки?

Мигель настроил конфигурацию GitHub Actions на создание нового образа при каждом коммите, но для автоматических сборок это не обязательно. Требуется только, чтобы их инициирование было каким-то образом автоматизировано. Другой распространенный подход — запуск новых сборок при добавлении тега в репозиторий, о чем вы узнаете чуть позже.

ВАЖНО

Чтобы добиться безопасности и надежности сборок, используйте сервис CD, который способен выполнять задачи в контейнерах, и принцип «сборка как код» (он же «конфигурация как код»), занося все скрипты и конфигурацию в репозиторий вместе с собираемым кодом.

Ваш ход: что мы упустили?

Команды разработчиков других сервисов заинтересовались методикой Мигеля, но в ходе работы с ними он обнаруживает в их процессах странности, которые явно не соответствуют требованиям методики. Для каждого из следующих технологических решений определите, какое требование оно нарушает:

1. Все сборки фронтенд-сервиса выполняются на виртуальной машине, выделенной несколько месяцев назад. До и после каждой сборки запускается скрипт для очистки виртуальной машины и удаления элементов, оставшихся после других сборок.
2. Сборки для сервиса пользовательских аккаунтов запускает раз в неделю специалист по сборкам в команде сервиса.
3. Члены команды фронтенд-сервиса имеют полный набор системных тестов, которые выполняются перед выпуском нового релиза и его развертыванием. Они запускаются только в этот отрезок времени.
4. Этапы сборки сервиса пользовательских аккаунтов прописаны в файле Makefile и выполняются билд-инженером команды.
5. Сборки фронтенд-сервиса выполняются в сервисе CD с полнофункциональным веб-интерфейсом. Вся конфигурация сборок задается и редактируется через веб-интерфейс.

Ответы

1. Это не соответствует требованию наличия временной среды для осуществления сборки. Даже если команда пытается поддерживать виртуальную машину в чистоте с помощью скрипта, нет гарантии, что удастся вычистить все.
2. Запуск сборок вручную означает, что сборки сервиса пользовательских аккаунтов не соответствуют требованию автоматизации.
3. Команда фронтенд-сервиса совершенно не справляется с непрерывной интеграцией! Откладывая системные тесты, нельзя гарантировать, что код всегда готов к выпуску.
4. Использование Makefile — это нормально (при условии, что он внесен в репозиторий), но проблема в том, что сборки выполняются билд-инженером там, где он считает нужным, а не сервисом CD.
5. Размещая конфигурацию только в веб-интерфейсе, команда не следует принципу «сборка как код».

Изменения интерфейса и баги

Мигель, возможно, и улучшил процесс сборки сервиса поиска по карте, но к сожалению, кое-что все еще может пойти не так! А начинается все с самых благих намерений: устранения технического долга.

У команды сервиса поиска по карте появляется немного времени, чтобы вернуться к некоторым давним незавершенным изменениям, и она решает изменить интерфейс самого важного метода, предоставляемого их сервисом: метода `search`. После создания сервиса в метод `search` естественно добавлялись новые параметры, в результате чего получился такой интерфейс:

```
def search(self, lat, long, zoom, park_types, pack_leader_only):
```

Каждый раз, когда появлялся запрос на новую функцию, в метод поиска добавлялся новый параметр и метод становился все длиннее!

Команда понимает, что если так будет продолжаться, то новые параметры будут добавляться бесконечно, и решает использовать язык запросов, чтобы произвольно добавлять новые атрибуты для запросов по мере необходимости. С этим нововведением метод `search` теперь выглядит совсем иначе:

```
def search(self, query):
```

Вся информация, которая раньше находилась в шести различных параметрах, теперь содержится в одной строке запроса.

Чтобы использовать этот обновленный интерфейс, команде фронтенд-сервиса необходимо полностью изменить способ вызова метода `search`. Например, в первоначальном интерфейсе поиск парков с площадками для выгула и дрессировки собак в центре Ванкувера, Британская Колумбия, осуществлялся следующим образом:

```
maps.search(49.2827, -123.1207, 8, [ParkTypes.training.name], False)
```

Чтобы сделать тот же запрос с помощью нового интерфейса, нужно преобразовать предыдущие параметры в такой запрос:

```
  maps.search(
    "lat=49.2827 && long=-123.1207 && zoom=8 && type in [{}]".format(
      ParkTypes.training.name))
```

Это довольно значительное изменение, и к сожалению, как вы сейчас увидите, оно попало в релиз до того, как команда сервиса поиска по карте предупредила об этом команду фронтенд-сервиса.

Когда сборки вызывают баги

Команда сервиса поиска по карте внесла крупное изменение в самый важный метод, который разработчики предоставляют через свой сервис, — метод `search` и забыла предупредить об этом команду фронтенда! Сервис поиска по карте используется фронтенд-сервисом. Команда фронтенда отвечает за развертывания не только своего сервиса, но и сервисов, от которых он зависит: сервиса поиска по карте и сервиса пользовательских аккаунтов.

Отличается ли непрерывный релиз от непрерывного развертывания?

В главе 1, давая определение *непрерывного развертывания*, я отмечала, что более точный термин — *непрерывный релиз*. Именно этим Мигель и занимается, то есть выпускает релизы для пользователей с каждым изменением. Поскольку фактического развертывания не происходит, будет меньше путаницы, если называть эту процедуру *непрерывным релизом*.

Команда фронтенда обычно раз в неделю развертывает новые версии сервиса поиска по карте и сервиса пользовательских аккаунтов. Конфигурация GitHub Actions, которую установил Мигель, настроена на сборку новой версии образа контейнера сервиса поиска при каждом обновлении главной ветки:

```
name: Build Map Search Service
on:
  push:
    branches: [main]
```

Задача будет запускаться при отправке изменений в главную ветку, то есть при каждом слиянии PR.

Команда сервиса поиска по карте действительно планирует сообщить команде фронтенда об этом изменении, но не все в команде полностью представляют последствия изменений Мигеля. Они не осознают, что фактически занимаются *непрерывным релизом*, то есть выпуском релизов при каждом изменении. И к сожалению, они не знают, что команда фронтенда, по случайному стечению обстоя-

тельств, планирует обновить развертывание сервиса поиска вскоре после слияния нового интерфейса:

- Произведено слияние обновленного метода поиска.
- Новый образ поиска по карте создан с помощью GitHub Actions.
- Команда фронтенда обновляет развертывание сервиса поиска по карте.

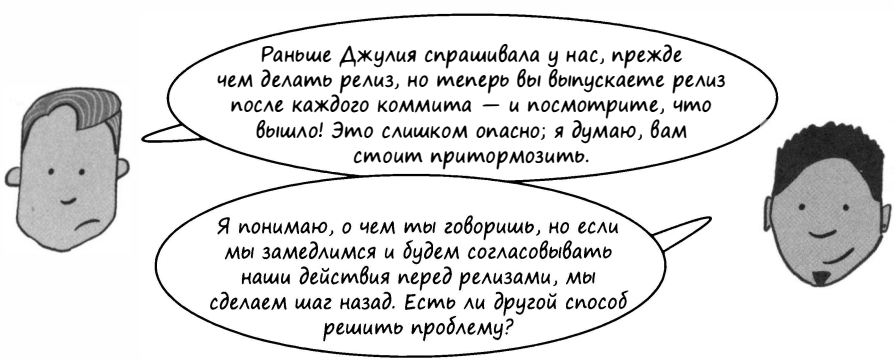
Команда фронтенда подтягивает последний образ контейнера поиска по карте, не понимая, что он содержит новый интерфейс!

Эта злополучная последовательность событий приводит к тому, что в выпущенном рабочем продукте фронтенд-сервис пытается вызвать старый интерфейс поиска, но текущая версия сервиса поиска по карте его уже не поддерживает. Что и приводит к масштабному сбою!

Сборки и коммуникация

Сбой стал настоящим стрессом как для команды фронтенда, так и для команды сервиса поиска. После того как команда фронтенда выяснила, что произошло, и обе команды начали устранять неполадки, между ними по понятным причинам сохранялось некоторое напряжение. Команда фронтенда дежурила в момент сбоя и отнюдь не воодушевилась, узнав, что причиной сбоя стали изменения в сервисе поиска по карте, о которых ее никто не предупредил.

К счастью, понимая чувства разработчиков фронтенда, Мигель не унывает и с радостью готов помочь. Он обсуждает произошедшее с Дани из фронтенд-команды:



Мигель и Дани вместе выясняют, что именно пошло не так, прежде чем решать, как выйти из положения. Они определяют три основные проблемы текущего процесса разработки сервиса поиска по карте:

- Невозможно различить релизы сервиса.
- Команда фронтенда не может контролировать, какой релиз используется.
- Отсутствуют автоматические уведомления о том, какие изменения произошли между релизами.

Эти три проблемы можно свести к одной основной: команда сервиса поиска по карте вообще не контролирует версии своих релизов! Возможно, вы уже заметили это, глядя на описанный Джулией процесс сборки и на скрипт сборки, созданный Мигелем:

```
#!/usr/bin/env sh
set -xe

cd "$(dirname "$0")"

docker build --tag top-dog-maps/map-search .
docker push top-dog-maps/map-search
```

Каждый релиз создает и выкладывает образ с абсолютно одинаковым именем.

Каждый релиз перезаписывает предыдущий! Это означает, что всегда доступна только одна версия сервиса поиска по карте (последняя), что приводит к трем проблемам, указанным Мигелем и Дани.

Семантическое версионирование

Чтобы избежать сбоев и недоразумений в будущем, команда сервиса поиска должна предоставить команде фронтенда возможность контролировать, какая версия сервиса используется, а также информировать ее о различиях между версиями.

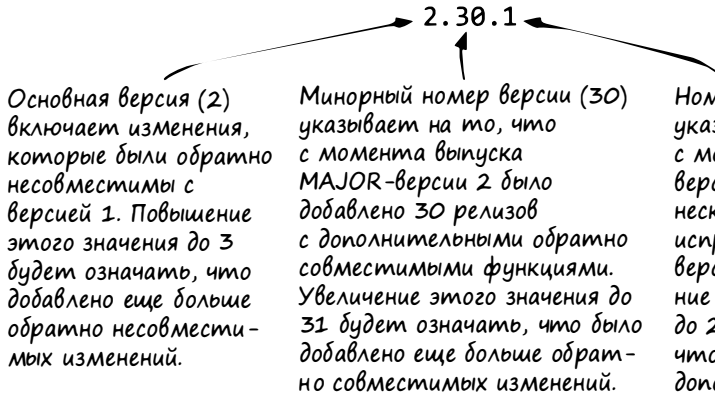
Команда сервиса поиска как минимум должна создавать более одной версии сервиса. В процессе выпуска релизов нельзя просто перезаписывать предыдущую версию; нужно создавать новую версию и обеспечить возможность использования предыдущей.

Часто для управления версиями ПО используется *семантическое версионирование*. В рамках этого подхода определяется способ присвоения версий программе таким образом, чтобы сообщать об изменениях между версиями на высоком уровне. Семантическое версионирование использует строку версии, состоящую из трех чисел, разделенных точками: `MAJOR`, `MINOR` и `PATCH`: `MAJOR.MINOR.PATCH`.

- *Приращение* (увеличение) основного номера версии (`MAJOR`) происходит при внесении изменений, несовместимых с предыдущими версиями.
- Второстепенный (минорный) номер версии (`MINOR`) обновляется при добавлении новых обратно совместимых функций.

Семантическое версионирование позволяет при необходимости включать в версию еще больше меток и метаданных; см. спецификацию семантического версионирования на сайте https://semver.org/.

- Номер патча (*PATCH*) изменяется при добавлении обратно совместимых исправлений имеющихся функций.



Словарик

Обратно совместимые (backward-compatible) изменения (то есть изменения, совместимые с предыдущими версиями) — это изменения, которые пользователь может использовать без необходимости производить обновление. *Обратно несовместимые* (backward-incompatible) изменения, напротив, такие, когда пользователю необходимо внести изменения в способ использования, иначе возникает риск багов и сбоев в работе. Изменение сигнатуры метода поиска было изменением, несовместимым с предыдущими версиями. Это изменение могло быть обратно совместимым, если бы команда добавила новый метод вместо изменения существующей сигнатуры. Сокращение обратно несовместимых изменений до минимума облегчает работу пользователей продукта, насколько это возможно.

Важность версионирования

Чтобы перейти на семантическое версионирование, команде сервиса поиска необходимо перестать записывать новые версии поверх предыдущего релиза, начав присваивать каждому релизу уникальный семантический номер версии. Мигель обновляет строки build и push в файле build.sh, и теперь они выглядят следующим образом:

```
docker build --tag top-dog-maps/map-search:$VERSION
docker push top-dog-maps/map-search:$VERSION
```

Скрипт сборки теперь помечает собранный образ семантическим номером версии, задаваемым с помощью переменной среды $VERSION.

Это изменение позволило команде сервиса поиска избавиться от проблем, выявленных Мигелем и Дани:

- Раньше релизы сервиса нельзя было различить, но теперь каждый релиз получит свой уникальный семантический номер версии.
- Вместо того чтобы контролировать, какой релиз используется, команда фронтенда может явно указать нужный номер версии для развертывания.
- Семантический номер версии теперь будет содержать высокоуровневую информацию о том, какие изменения вносились между версиями. По тому, был ли изменен основной или второстепенный номер или номер патча, команда фронтенда узнает, является ли новый релиз просто исправлением багов (увеличение номера патча), содержит ли он новые функции (изменение минорного номера версии) или содержит несовместимые с прошлым релизом изменения (приращение основного номера версии).

ВАЖНО

Использование четкой, последовательной схемы управления версиями, такой как семантическое версионирование, дает пользователям необходимую информацию и возможность контролировать процесс, тем самым уменьшая негативное восприятие внесенных изменений.

Примечания к релизу необходимы даже при семантическом версионировании

Семантический номер версии на высоком уровне указывает, каких изменений ожидать, но пользователю, скорее всего, потребуется больше информации (то есть что именно изменилось: какие баги исправлены, какие новые возможности добавлены, какие изменения, несовместимые с предыдущими версиями, сделаны). Именно здесь на помощь приходят *примечания к релизу*.

Они прилагаются к релизу и содержат подробный список всех изменений, о которых необходимо знать пользователям. Это особенно важно, поскольку даже если основной номер версии не увеличился, в ней все равно могут оказаться изменения, несовместимые с предыдущей версией. Иногда это просто ошибка, но часто так происходит потому, что очень трудно предсказать, как некоторые изменения повлияют на пользователей (закон Хайрама (Hyrum Wright) гласит: «Все наблюдаемое поведение системы в конечном итоге зависит от пользователей», https://www.hyrumslaw.com/). Создание примечаний к релизу можно автоматизировать на основе комментариев к коммитам и описаний PR, но здесь я не буду вдаваться в подробности.

Включение номера версии в сборку

Анализируя обновления скрипта сборки, сделанные Мигелем, вы можете задаться вопросом, откуда на самом деле будет браться значение **$VERSION**:

```
#!/usr/bin/env sh
set -xe

cd "$(dirname "$0")"

docker build --tag top-dog-maps/map-search:$VERSION
docker push top-dog-maps/map-search:$VERSION
```

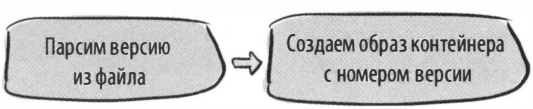
Часто версию получают из тега Git, который содержит семантическую версию. Этот подход позволяет настроить автоматизацию так, чтобы она запускалась при добавлении тега. Например, при использовании GitHub Actions синтаксис запуска может быть следующим:

```
on:
  push:
    tags:
      - '*'
```

Задача будет запускаться при добавлении любых новых тегов в репозиторий.

А внутри шагов значение тега, инициировавшего отправку, можно получить из того, что в GitHub Actions называется *контекстом*: например, при инициировании с помощью тега `v0.0.1` значение `${{ github.ref }}` будет `refs/tags/v0.0.1`. (Подробнее см. документацию по GitHub Actions на https://docs.github.com/ru/actions.)

Однако если Мигель воспользуется этим подходом, он перестанет выпускать релизы непрерывно; он будет выпускать их только тогда, когда тег будет добавляться в репозиторий. Один из способов, позволяющих одновременно выпускать релизы непрерывно и использовать семантическое версионирование, — хранить номер текущей версии в файле в репозитории и требовать, чтобы каждый коммит с изменением кода увеличивал значение номера. Это означает обновление пайплайна сборки таким образом, чтобы он сначала анализировал номер версии из файла в репозитории, а затем передавал это значение скрипту сборки:



Мигель, скорее всего, захочет немного усложнить пайплайн. Иногда указать номер версии забывают, и в таких случаях пайплайн должен выдать ошибку. Кроме того, если изменений, ориентированных на конечного пользователя, не происходит (например, если изменения касаются только юнит-теста или документа разработчика), новую версию создавать не следует. Это будет выглядеть примерно так:

Очередной сбой!

Подробнее об откатах (и других методах безопасного развертывания) вы узнаете в главе 10.

Сразу после того, как Мигель и Дани с большим трудом устранили источник напряженности между командами, предоставив фронтенд-команде контроль над тем, когда и как использовать обновления, поступающие от команды сервиса поиска, произошел очередной сбой.

Словарик

Стороннее программное обеспечение — это ПО, созданное за пределами компании, например другой компанией или в рамках проекта с открытым исходным кодом.

В результате анализа этого сбоя обнаружился баг в сервисе поиска по карте. К счастью, этот сервис версионирован, и команда фронтенда может быстро откатиться к предыдущей рабочей версии. Разработчики фронтенда сообщают о проблеме команде сервиса поиска и оставляют их разбираться с источником проблемы. (К счастью, поскольку фронтенд-команда решила проблему быстро и легко, отношения между командами не пострадали!)

Мигель помогает с отладкой неисправности и с удивлением обнаруживает, что проблема заключается в изменении сторонней библиотеки (querytosql), от которой зависит работа сервиса поиска по карте, связанная с выполнением поисковых SQL-запросов.

Сервис поиска по карте использует библиотеку querytosql для выполнения запросов к базе данных кэшированных карт.

Библиотека изменила интерфейс одного из своих методов на первый взгляд безобидным образом, но к сожалению, этого оказалось достаточно, чтобы сервис поиска по карте перестал работать. Функция, вызываемая в рамках поискового функционала, в библиотеке querytosql раньше выглядела так:

```
def execute(query, db_host, db_username, db_password, db_name):
```

В последней версии авторы querytosql поняли, что они могут упростить сигнатуру функции, создав объект для хранения всей информации о подключении к БД, и обновили сигнатуру:

```
def execute(query, db_conn):
```

Этот параметр теперь содержит хост БД, имя пользователя, пароль и имя БД.

Новый код теперь намного чище, но к сожалению, это изменение несовместимо с предыдущими версиями, поэтому функция `execute` в сервисе поиска по карте теперь не работает! Команде необходимо обновить каждый экземпляр приложения, где вызывалась эта функция, чтобы использовать новый интерфейс и обеспечить совместимость с этими изменениями.

Ошибки в зависимостях во время сборки

Мигель оценил иронию ситуации, заключающуюся в том, что сервис поиска столкнулся почти с той же проблемой, какую его команда недавно создала для фронтенд-сервиса, когда обновила сигнатуру своего метода поиска! Но он озадачен: почему ее не обнаружили тесты, выполняемые в рамках CI?

Из главы 7 вы, возможно, помните, что баги могут проникать в систему в нескольких местах, и одно из них — это этап сборки, если на нем происходят изменения зависимостей:

- Осуществляем сборку рабочего артефакта с соответствующим коммитом.

Зависимости: во время сборки рабочих артефактов могут быть подтянуты зависимости, влекущие за собой дополнительные изменения, которых не было во время CI, что чревато появлением новых ошибок.

Оказывается, именно это и произошло с сервисом поиска по карте:

- Открыт PR (добавлено обратно совместимое изменение).
- На основе PR запущена CI (включая тесты и линтинг), которая подтянула версию 1.3.2 библиотеки querytosql.
- Тесты и линтинг, включенные в CI, успешно пройдены.
- После проверки произведено слияние PR.
- Команда quertytosql выпустила новую версию своей библиотеки 1.4.0 с обновленной функцией `execute`.
- После слияния PR было запущено действие сборки, которое создало новый образ сервиса поиска по карте с использованием querytosql версии 1.4.0.

По невероятному стечению обстоятельств новая версия querytosql была выпущена в промежутке между запуском тестов и сборкой нового образа.

В главе 7 команда, столкнувшаяся с аналогичной проблемой, обошла ее, внедрив периодические сборки для отслеживания изменений в зависимостях. Но даже при периодических сборках остается промежуток времени, когда в систему могут за-

красться изменения зависимостей (то есть между периодической сборкой и релизом). Чтобы снизить риск, можно использовать периодическую сборку для создания артефактов релиза, лучший способ гарантировать, что изменения в зависимостях не возникнут, — это явно *привязать* продукт к определенным версиям зависимостей.

Это изменение было обратно несовместимым — почему команда querytosql не изменила основной номер версии?

Правильно подмечено! В действительности разработчики проектов совершенно по-разному следуют правилам семантического версионирования и трактуют их. К сожалению, довольно часто при добавлении таких изменений основной номер версии не повышают. Это еще одна причина, по которой стоит осуществлять явную привязку зависимостей, чтобы контролировать время получения обновлений и следить за появлением изменений, несовместимых с предыдущими версиями.

Привязка зависимостей

Лучший способ свести к минимуму количество багов и прочее непредсказуемое поведение, вызванное изменениями в зависимостях, — использовать тот же подход, который Мигель и Дани разработали для команды фронтенда. Точно так же как команда фронтенда сейчас однозначно полагается на конкретную версию сервиса поиска по карте, сервису необходимо явно указать версии зависимостей, на которые он опирается. Вот содержимое файла requirements.txt для сервиса поиска по карте:

```
beautifulsoup4
pytest > 6.0.0
querytosql
```

Ни одно из требований окружения не привязано к конкретной версии; наиболее корректно описан pytest, который должен быть не ниже версии 6.0.0, но допускается любая версия выше этой.

Поскольку для библиотеки querytosql не определены требования окружения, подходит любой номер версии. Поэтому когда выполнялась последняя сборка для образа контейнера поиска по карте, была просто подтянута последняя версия. Как только Мигель понял, что произошло, он быстро обновил файл requirements.txt, явно указав в нем номера версий, к которым следует проводить привязку:

```
beautifulsoup4 == 4.10.0
pytest == 6.2.5
querytosql == 1.3.2
```

Мигель привязывает требование к ранее выпущенной версии querytosql, с которой, как он знает, совместим код сервиса поиска по карте. Таким образом, команда сервиса может спокойно обновлять querytosql до последней версии, при этом безопасно создавая новые версии образа своего сервиса.

Теперь команда сервиса поиска контролирует использование обновлений для необходимых библиотек и осуществляет сборку безопасно, не беспокоясь о неожиданных изменениях в функциональности из-за изменений в зависимостях.

А что, если я использую Pipfile вместо requirements.txt? Или вообще не использую Python?

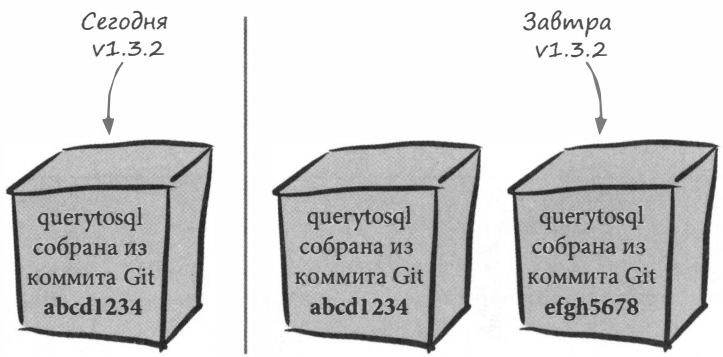
Независимо от языков или инструментов, привязка зависимостей к определенной версии — самый безопасный вариант. Pip-файлы также поддерживают синтаксис для указания версий зависимостей, и то же самое справедливо для большинства языков и инструментов. Если язык или инструмент не дает такой возможности, считайте это его минусом и найдите альтернативу (или создайте свой собственный инструмент).

Зачем делать привязку к версии патча? Разве не следует подтягивать исправления багов автоматически?

Вместо привязки к конкретной версии для некоторых инструментов и языков можно указать диапазон версий. Например, в файле requirements.txt можно задать querytosql == 1.3.*, чтобы указать, что разрешены любые релизы патчей версии 1.3. Однако не все инструменты и языки поддерживают подобный синтаксис. Даже исправления багов — это изменения в функциональности (еще один пример действия закона Хайрама!). Контроль за использованием изменений делает созданные артефакты более предсказуемыми.

Привязка к версии не панацея

Оказывается, в большинстве языков и инструментов привязка к определенной версии еще не является гарантией того, что в используемой библиотеке не произойдет изменений между двумя моментами времени. Это может показаться невероятным, но причина в том, что обычно существует возможность перезаписать предыдущую версию библиотеки, образа и т. д. совершенно новой, имеющей тот же номер. Дело в том, что версии обычно рассматриваются как теги, и ничто не мешает изменить сущность, на которую указывает этот тег. Например, возможно, что в какой-то момент команда querytosql решит перезаписать свою версию 1.3.2 новой:



Подобная перезапись уже выпущенных версий — плохая практика, поскольку она сводит на нет весь смысл использования номера версии (то есть предоставления потребителям контроля и информации об изменениях), но время от времени такое случается (обычно когда кто-то с благими намерениями хочет выпустить релиз с исправлением как можно быстрее и проще, пока никто не успел воспользоваться вёрсией, содержащей баг).

Но не все потеряно! Можно сделать привязку зависимостей с *еще большим контролем*, чем просто указание номера версии, дополнительно указав ожидаемый *хеш* содержимого пакета. Это позволит избежать случайного использования изменений, к которым вы не готовы, — если содержимое зависимости изменится, то изменится и хеш.

Словарик

Термин «*хеш*» здесь — сокращенное обозначение результата криптографической функции хеширования, применяемой к данным. В контексте сборок и зависимостей мы говорим о применении хеш-функции к содержимому программного артефакта. Суть в том, что полученный хеш (то есть результат применения функции хеширования к артефакту) всегда будет одним и тем же, поэтому если содержимое артефакта изменится, то изменится и хеш. Примерами функций хеширования являются **md5** и различные версии **sha**, например **sha256**.

Привязка к хешам

Мигель должен удостовериться, что образ сервиса поиска по карте, создаваемый командой, надежен и что ни одна из его зависимостей внезапно не изменится, поэтому он делает еще одно обновление в файле requirements.txt, включая в файл ожидаемые хеши зависимостей:

```
beautifulsoup4 == 4.10.0 \
  --hash=sha256:9a315ce70049920ea4572a4055bc4bd700c940521d36fc858205ad4fcde149bf
pytest == 6.2.5 \
  --hash=sha256:7310f8d27bc79ced999e760ca304d69f6ba6c6649c0b60fb0e04a4a77cacc134
querytosql == 1.3.2 \
  --hash=sha256:abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234abcd1234
...
```

При использовании файла requirements.txt указание --hash для любых зависимостей автоматически требует предоставления хешей для всех зависимостей, включая зависимости зависимостей, поэтому Мигель должен добавить и эти хеши. Но это стоит затраченных усилий и гарантирует также защиту от изменения в зависимостях зависимостей!

Осуществляя привязку к хешу пакета, команда использует *однозначные идентификаторы*, которые нельзя использовать для идентификации более чем одного объекта. Версии и теги неоднозначны, поскольку их можно неоднократно использовать или изменять так, чтобы они указывали на другие данные.

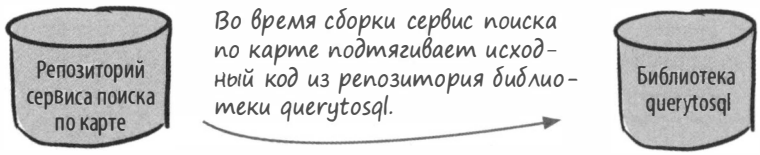
Использование однозначных идентификаторов в других языках и инструментах

Разумно ожидать, что любой применяемый язык или инструмент также будет поддерживать указание ожидаемых хешей используемых артефактов. На самом деле одним из главных преимуществ использования образов контейнеров является то, что содержимое автоматически хешируется и хеши можно использовать при извлечении и запуске образов.

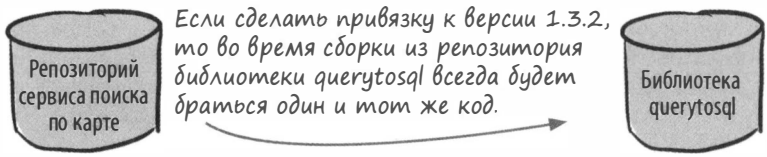
ВАЖНО

Для полной уверенности в том, что зависимости (и зависимости зависимостей) не изменятся без вашего ведома и не станут источником багов или нежелательных изменений в поведении продукта, явно привязывайте их не только к конкретным версиям, но и к хешам конкретных версий, которые вы собираетесь использовать.

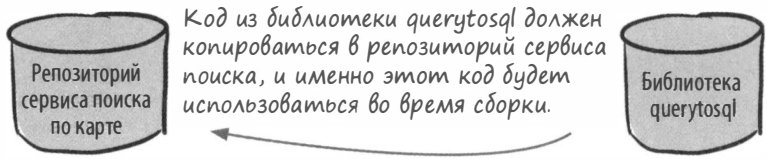
Причиной возникновения проблемы между сервисом поиска по карте и библиотекой querytosql стал тот факт, что код сервиса и код для библиотеки хранились в разных репозиториях:



Это стало проблемой, поскольку при таком подходе образ сервиса поиска по карте собирается с тем, что на данный момент имеется в репозитории querytosql. Команда сервиса решила эту проблему путем извлечения кода одного и того же коммита с определенным тегом из репозитория библиотеки querytosql:



Другой подход заключается в том, чтобы вообще не извлекать этот код во время сборки, скопировав его изначально в репозиторий сервиса поиска по карте:



Данный подход гарантирует, что во время сборки всегда будет использоваться одна и та же версия библиотеки querytosql, поскольку в этот момент ничего ниоткуда не извлекается.

Эту идею можно развить еще дальше. В главе 3 кратко упоминалась концепция *монорепозитория*: одного репозитория, в котором хранится весь исходный код не только проекта, но и всей компании. Если бы Top Dog Maps использовала монорепозиторий, весь код сервиса поиска карт, фронтенда и всех зависимостей хранился бы и версионировался в одном репозитории.

Такой подход в некоторых случаях позволяет свести к минимуму необходимость явного версионирования, но не может избавить от него полностью и всегда (например, фронтенд-сервис фактически полностью зависит от образа сервиса поиска по карте, а не от исходного кода, поэтому образ последнего по-прежнему нуждается в версионировании). Кроме того, он может добавить сложности (большинство инструментов не предусматривают использования монорепозиториев), но в то же самое время значительно снизить неопределенность (то есть вы всегда будете видеть именно тот исходный код, который используется) и облегчить реализацию (например, обширных радикальных изменений по всем проектам).

Заключение

Даже если процесс сборки налажен, нелишним будет регулярно пересматривать его и проверять, можно ли что-то улучшить. На ранних этапах проекта такие меры, как ручная сборка артефакта, могут быть оправданны, но они не выдерживают проверки временем. К счастью, как убедился Мигель в проекте Top Dog Maps, оказалось относительно просто перейти от ручного процесса к автоматизированному и значительно повысить надежность сборки. Мигель также на собственном опыте убедился, как важно использовать передовые практики сборки, чтобы облегчить взаимодействие с пользователями продукта и защититься от проблем, связанных со сторонними инструментами, которые он использует в работе.

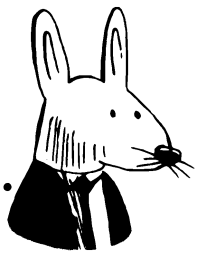
Итоги

- Безопасными и надежными являются сборки, которые автоматизированы и соответствуют требованиям SLSA: применение принципа «сборка как код» (вариация принципа «конфигурация как код»), запуск на сервисе CD и выполнение во временных средах.
- Управление версиями продукта с помощью последовательной схемы версионирования, такой как семантическое версионирование, позволяет эффективно предоставлять пользователям контроль и информацию об изменениях от версии к версии.
- Защитите себя от нежелательных изменений в зависимостях, привязывая их не только к явным версиям, но и к хешам явных версий, и интегрируйтесь с ними без ограничений.

Далее...

Автоматизированный (и даже непрерывный) выпуск релизов — это здорово, но если вы работаете с сервисом, вас наверняка интересует, как осуществить развертывание созданных артефактов. В следующей главе мы рассмотрим практики, которые позволяют быстро и просто включить этап развертывания в пайплайн CD.

10 | Надежное развертывание



В этой главе

- ✓ Метрики DORA для стабильности: частота сбоев при изменениях и время восстановления работоспособности сервиса
- ✓ Безопасное развертывание путем реализации стратегии отката
- ✓ Применение сине-зеленых и канареечных развертываний для уменьшения влияния неудачных развертываний
- ✓ Как непрерывное развертывание помогает достичь высших (элитных) значений метрик DORA

Для многих проектов этап развертывания — это момент истины. Если выполнять развертывание без автоматизации или недостаточно осторожно, оно может доставить массу хлопот.

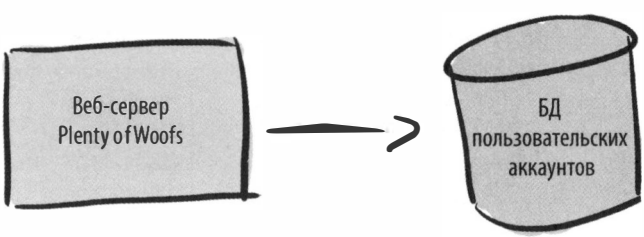
В этой главе мы рассмотрим, как максимально исключить человеческий фактор из процесса развертывания и как измерить эффективность самого процесса. Также мы обсудим другую расшифровку аббревиатуры CD — непрерывное развертывание (continuous deployment) — и узнаем, что требуется для его реализации, а также какие компромиссы влечет за собой этот подход.

Множество проблем с развертыванием

Plenty of Woofs, еще одна социальная сеть, популярная среди любителей собак, помогает владельцам собак найти подходящие места для игр своих питомцев. Собаководы могут найти других владельцев собак поблизости, отфильтровать питомцев по размеру, совместимости и любимым играм, а также общаться и обмениваться фотографиями.

К сожалению, в последние месяцы на сайте Plenty of Woofs регулярно происходят сбои, когда основные функции не работают по несколько дней подряд! Сбои возникали сразу после развертывания, что заставляло разработчиков судорожно искать проблемы и устранять их как можно быстрее.

Компания небольшая, менее 20 человек, с довольно простой архитектурой сайта, включающей монолитный сервис, который выполнял все задачи, опираясь на базу данных:



Такая архитектура привела к тому, что каждый сбой затрагивал всю компанию, и сотрудники все время оставались в напряжении. Они начали привыкать заниматься устранением этих сбоев по выходным, и в компании появились признаки выгорания.

Метрики DORA для стабильности

Хотя разработчики Plenty of Woofs постоянно борются со сбоями, они успели внедрить несколько полезных методик, включая отслеживание метрик DORA. В главе 8 мы уже рассматривали метрики DORA; более подробную информацию о них см. на сайте https://www.devops-research.com/research.html.

Метрики DORA для скорости

Скорость оценивается по двум метрикам:

- частота развертывания;
- время доставки изменений.

Метрики DORA для стабильности

Стабильность оценивается по двум метрикам:

- время восстановления работоспособности сервиса;
- частота сбоев при изменениях.

Мы рассматривали метрики DORA с акцентом на скорости, оцениваемой частотой развертывания и временем внесения изменений. Две другие метрики касаются стабильности, и именно их в первую очередь учитывают разработчики Plenty of Woofs при анализе проблемы отказов:

- *Время восстановления работоспособности сервиса* показывает, сколько времени требуется компании для восстановления после сбоя.
- *Частота сбоев при изменениях* определяет, какой процент развертываний приводит к сбою.

Как вы, наверное, помните, члены команды DORA также на основе этих метрик составили классификацию команд по их производительности, распределив их по четырем категориям: низкоэффективные, среднеэффективные, высокоэффективные и суперэффективные (элитные). Вот как выглядит таблица значений метрик стабильности:

Что случилось в промежутке времени от одной недели до 6 месяцев? Этот разрыв показывает, что перейти от низкого уровня к среднему невозможно за счет простого сокращения времени восстановления работоспособности с 6 месяцев до 5; необходимо найти способ восстанавливать работоспособность сервиса за 1 неделю.

Метрика	Супер-эффективные	Высоко-эффективные	Средне-эффективные	Низко-эффективные
Время восстановления работоспособности сервиса	Менее одного часа	Менее одного дня	От одного дня до одной недели	Более шести месяцев
Частота сбоев при изменениях	0–15%	16–30%	16–30%	16–30%

Команда DORA выявила явное различие между суперэффективными командами и всеми остальными, при этом в высоко-, средне- и низкоэффективных командах процент отказов оказался примерно одинаковым.

Что делать, если у меня не сервис?

Если вы работаете над проектом, который не размещается и не запускается как сервис, может быть сложно понять, применимы ли к нему эти метрики. (Подробнее о видах программных продуктов, которые можно поставлять, в том числе библиотеках, двоичных файлах, конфигурациях, образах и сервисах, см. в главе 1.)

Возможно, вам и не нужна вся эта глава. Стратегии, описанные в ней, специфичны для конкретного развертывания, поэтому, к сожалению, они, скорее всего, не очень вам помогут, но сами метрики DORA все же применить получится. Кроме того, не забывайте, что, как уже обсуждалось в главе 1, когда мы говорим о *непрерывном развертывании*, на самом деле мы имеем в виду *непрерывный релиз*, когда каждое изменение становится доступно пользователям по мере добавления.

Метрики, связанные со стабильностью, можно применить к другим типам ПО (например, библиотекам и двоичным файлам) следующим образом:

- *Частота сбоев при изменениях* остается прежней, но рассматривайте ее не как процент развертываний, которые вызывают сбой, а как процент релизов, которые содержат достаточно серьезную ошибку, чтобы для ее исправления требовался выпуск патча (то есть значительный баг, исправление которого нельзя отложить на потом и включить в более поздний релиз).
- *Время восстановления работоспособности сервиса* становится *временем релиза исправлений*, то есть временем, которое потребуется, чтобы выпустить патч в открытый доступ с момента сообщения о серьезном баге.

Всегда ли значения этих метрик применимы для супер-, высоко-, средне- и низкоэффективных команд? Принцип остается тем же: чем быстрее вы обнаруживаете проблемы и выпускаете их исправления для пользователей, тем стабильнее ваш продукт.

Однако решения для автоматизации процессов, позволяющие значительно улучшить эти метрики для сервисов (о чем вы узнаете в этой главе), часто недоступны для других видов ПО, кроме сервисов. Иногда можно автоматизировать откат (возврат) коммитов, но это не просто вопрос отката к предыдущей версии, как в случае с развернутым ПО. Зачастую без исправления ошибки обойтись не получится, поэтому стремиться к суперэффективным показателям времени восстановления работоспособности или времени доставки изменений менее часа может быть нецелесообразно.

Метрики DORA для сайта Plenty of Woofs

Обе DORA-метрики стабильности могут зависеть от метрик скорости, поэтому (как вы сейчас увидите) оценивать метрики стабильности в отрыве от метрик скорости не получится. Ниже приведен полный список метрик и их значений для супер-, высоко-, средне- и низкоэффективных команд:

Метрика	Супер-эффективные	Высоко-эффективные	Средне-эффективные	Низко-эффективные
Частота развертывания	Несколько раз в день	От одного раза в неделю до одного раза в месяц	От одного раза в месяц до одного раза в полгода	Реже, чем раз в полгода
Время доставки изменений	Менее одного часа	От одного дня до одной недели	От одного месяца до полугода	Более полугода
Время восстановления работоспособности сервиса	Менее одного часа	Менее одного дня	От одного дня до одной недели	Более 6 месяцев
Частота сбоев при изменениях	0–15%	16–30%	16–30%	16–30%

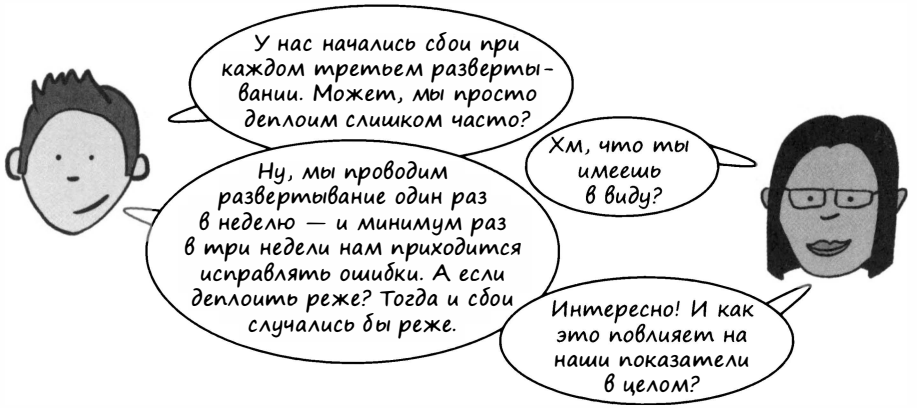
А это значения, которые команда Plenty of Woofs определила для своей производительности:

- *Частота развертывания* — раз в неделю (*высокая*).
- *Время доставки изменений* — менее одной недели (*высокий показатель*).
- *Время восстановления работоспособности сервиса* — минимум один день, но чаще несколько дней (*среднее*).
- *Частота сбоев при изменениях* — в среднем за год составляет 10 % (одно из каждых десяти развертываний). В последнее время это скорее одно из трех развертываний, или около 33 % (ниже максимального значения для высоко-, средне- и низкоэффективных команд, поэтому можно с уверенностью считать, что это соответствует *низкой* эффективности).

Компания Plenty of Woofs демонстрирует высокую эффективность, если рассматривать только метрики скорости, но при учете метрик стабильности скатывается к средним и низким показателям.

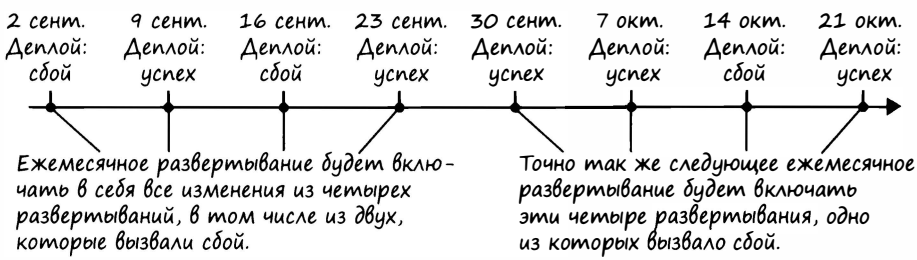
А если делать развертывания реже?

Арчи и Сарита работают над проектом Plenty of Woofs и вызвались решить проблему сбоев, с которой столкнулась команда. Арчи обращается к Сарите с идеей о том, что можно сделать.



Арчи предлагает проводить развертывание (деплой) раз в месяц, а не раз в неделю, и предположить, как при этом будут выглядеть метрики DORA.

Сарита для начала пытается разобраться с частотой сбоев при изменениях, анализируя несколько предыдущих развертываний, которые привели к сбоям, и сравнивая показатели с тем, что было бы, если бы деплой проводился раз в месяц:



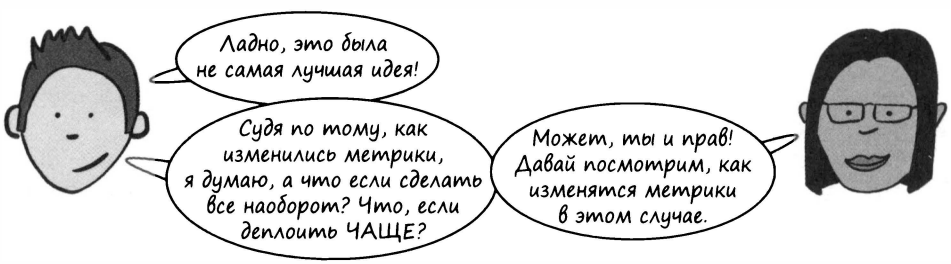
Каждое из ежемесячных развертываний включало бы как минимум один набор изменений, вызывающий сбой. При ежемесячном развертывании их метрики DORA стали бы такими:

- *Частота развертывания* — раз в месяц (*средняя*).
- *Время доставки изменений* — около месяца (*среднее*).
- *Время восстановления работоспособности сервиса* — может остаться на уровне одного или нескольких дней, а может и ухудшиться, если придется работать с бóльшим количеством изменений одновременно (в лучшем случае *среднее*).
- *Частота сбоев при изменениях* — если посмотреть на предыдущие сбои и сопоставить их с датами ежемесячных релизов, то при переходе на ежемесячные релизы сбои будут при *каждом* развертывании, то есть 100% (крайне *низкая*).

А если делать развертывания чаще?

Арчи предположил, что менее частое развертывание может решить проблему сбоев на сайте Plenty of Woofs, но если посмотреть на метрики DORA, то становится ясно, что эффективность работы команды при этом снизится по всем параметрам:

- *Частота развертывания* — с одного раза в неделю (*высокая*) до одного раза в месяц (*средняя*).
- *Время доставки изменений* — от срока менее недели (*высокий показатель*) до примерно одного месяца (*средний показатель*).
- *Время восстановления работоспособности сервиса* — несколько дней или хуже (в лучшем случае *среднее*).
- *Частота сбоев при изменениях* — поскольку ежемесячные развертывания будут включать в себя изменения по крайней мере четырех еженедельных развертываний (треть из которых вызывает сбои), то все ежемесячные развертывания, скорее всего, будут приводить к сбоям: 100 % (крайне *низкая*).



Чтобы понять возможный результат, они глубже изучили одно из недавних развертываний, которое вызвало сбой. Они обратили внимание, когда именно были внесены изменения, вошедшие в это развертывание:

Развертывание 14 октября включало в себя изменения за 5 дней. Если бы развертывание проводилось в каждый из этих дней, скорее всего, в результате было бы четыре успешных развертывания и одно со сбоем.

Ежедневные развертывания и сбои

Рассматривая сбой от 14 октября, Сарита и Арчи заметили, что изменение, вызвавшее сбой, было внесено во вторник, поэтому если бы они выполняли ежедневные развертывания, то сбой произошел бы только в этот день. Они расширяют анализ и рассматривают последние восемь развертываний, каждое из которых содержит изменения за 5 рабочих дней, и на основании даты внесения изменений со сбоем вычисляют, какой процент ежедневных развертываний может вызвать сбои:

- *2 сентября (сбой)* — одно ежедневное развертывание из пяти могло привести к сбою.
- *9 сентября (успешно)* — пять успешных ежедневных развертываний.
- *16 сентября (сбой)* — два ежедневных развертывания из пяти могли привести к сбою.
- *23 сентября (успешно)* — пять успешных ежедневных развертываний.
- *30 сентября (успешно)* — пять успешных ежедневных развертываний.
- *7 октября (успешно)* — пять успешных ежедневных развертываний.
- *14 октября (сбой)* — одно ежедневное развертывание из пяти могло привести к сбою.
- *21 октября (успешно)* — пять успешных ежедневных развертываний.

За эти 8 недель из 40 ежедневных развертываний 4 привели бы к сбою: 4 / 40 = 10 % развертываний вызвали бы сбой. В целом метрики DORA при ежедневном развертывании выглядели бы так:

- *Частота развертывания* — ежедневно (почти наивысшая, но пока только *высокая*).
- *Время доставки изменений* — менее одного дня (также почти наивысший, но пока только *высокий показатель*).
- *Время восстановления работоспособности сервиса* — нельзя определить точно, скорее всего, диагностика и устранение проблем займут столько же времени, сколько и раньше, так что по-прежнему один или несколько дней (*среднее*).
- *Частота сбоев при изменениях* — исходя из последних восьми еженедельных развертываний, похоже, что только 10 % развертываний приведут к сбоям (*наивысшая*).

Увеличение количества развертываний не изменит количество сбоев, но уменьшит вероятность того, что отдельное развертывание будет заканчиваться сбоем.

ВАЖНО

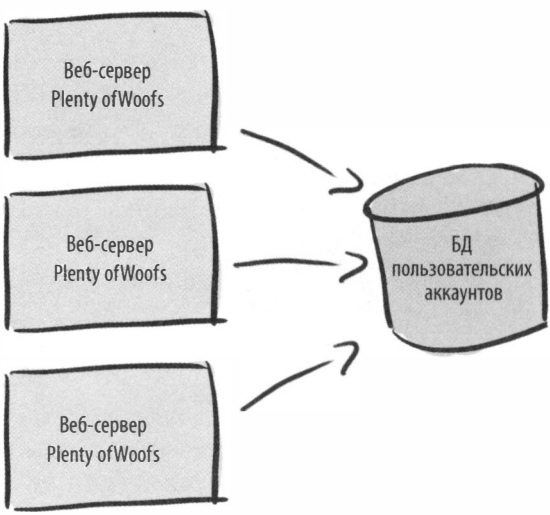
Более частые развертывания снижают степень риска при каждом развертывании. В каждом развертывании будет меньше изменений, поэтому вероятность того, что в развертывании окажется изменение, которое вызовет сбой, ниже.

Как увеличить частоту развертываний

Хотя Сарита и Арчи еще не уверены, как увеличение частоты развертываний поможет справиться со сбоями, они понимают, что оно повысит производительность компании с точки зрения метрик DORA. Они начинают планировать переход к ежедневным деплоям и надеются, что в процессе получат дополнительные данные. Они хотят критически оценить свой процесс развертывания, который сводится к обновлению нескольких деплоев веб-сервера Plenty of Woofs:

> Команда Plenty of Woofs масштабирует свой веб-сервер вручную. Многие виды облачного развертывания позволяют масштабировать развертывания без участия человека (это называется *автомасштабированием*). Здесь мы не будем подробно обсуждать вопросы масштабирования.

Трех экземпляров веб-сервера достаточно, чтобы справиться с нагрузкой на сервис Plenty of Woofs. По мере роста популярности сайта разработчикам может потребоваться добавить больше экземпляров и/или изучить возможности автомасштабирования.



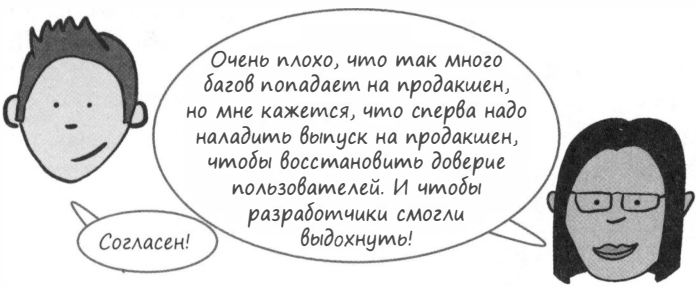
Текущий процесс развертывания выглядит так:

1. Раз в неделю, каждый четверг в середине дня, запускается деплой.
2. Все члены команды должны быть доступны в четверг и пятницу (а возможно, и на выходных) для решения любых возникающих проблем.
3. Во время развертывания метрики экземпляров веб-сервера отслеживаются с помощью стороннего сервиса CellphoneDuty, который уведомляет команду, если значения метрик плохие.
4. Когда возникает проблема, вся команда анализирует ее и принимает решение. После того как исправление вносится в главную ветку, происходит сборка и деплой веб-сервера повторяется.

Устраняем недостатки разработки

Оценив процесс разработки Plenty of Woofs, Сарита резюмирует, что существуют две основные проблемы, которые действительно замедляют его и влияют на метрики:

- Устранение проблем занимает очень много времени, порядка нескольких дней.
- Слишком много ошибок упускается в процессе непрерывной интеграции — и не просто ошибок, а серьезных багов, приводящих к сбоям.



Помимо того что сбои замедляют работу и мешают деплоить часто, развертывания очень напрягают команду. Все стали бояться четвергов! Сарита и Арчи поставили цель:

Найти способ устранять ошибки в выпускаемом продукте так, чтобы на это не уходили часы (или дни!).

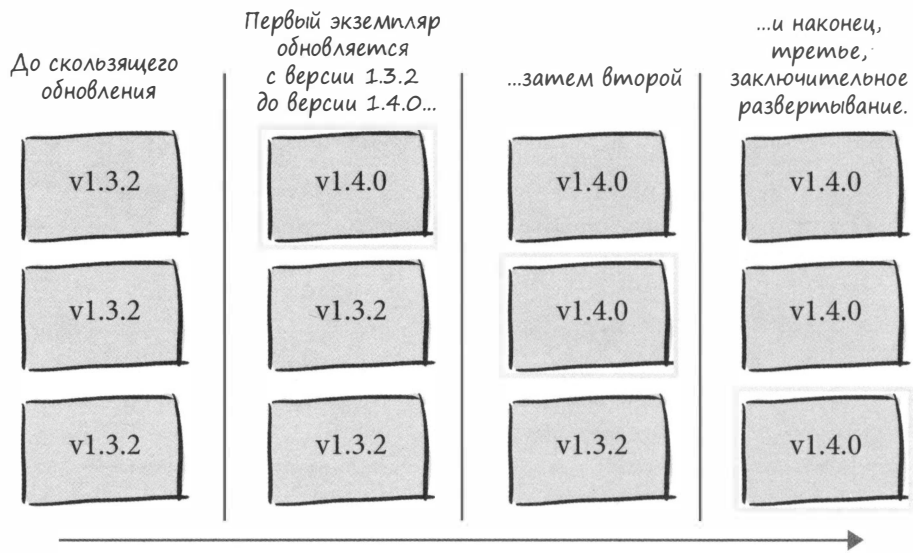
Когда продакшен снова станет стабильным, Арчи и Сарита смогут перейти к тому, как предотвратить появление багов, используя методы, описанные в главах 3–7!

Скользящие обновления

Разработчики Plenty of Woofs проводят скользящие обновления вручную, но многие среды развертывания (например, Kubernetes) предоставляют функциональность для таких обновлений по умолчанию.

Первая проблема, за которую принимаются Сарита и Арчи, заключается в том, что на исправление багов в продакшене уходит от нескольких часов до нескольких дней. Чтобы найти решение, они подробно изучают, как именно происходит развертывание обновлений на трех экземплярах веб-сервера Plenty of Woofs.

При тщательном анализе становится ясно, что команда Plenty of Woofs использует *скользящее обновление* экземпляров веб-сервера. По очереди на каждом хосте останавливается текущий контейнер веб-сервера и запускается новый экземпляр (с новой версией). После обновления одного хоста обновляется следующий, и так далее, пока не будут обновлены все.

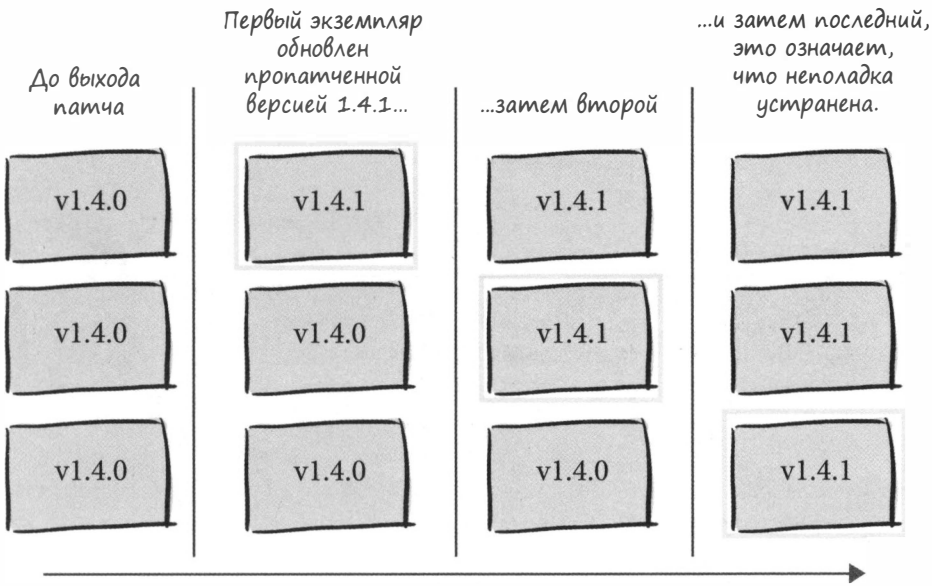


Словарик

Скользящее обновление (rolling update) означает, что экземпляры обновляются до новой версии по очереди. В любой момент времени хотя бы один экземпляр сервиса запущен и работает, что позволяет избежать простоя всего сервиса. Более простой способ — снести все экземпляры и обновить их одновременно, но в таком случае сервис не будет работать, пока происходит обновление. Во время скользящего обновления часть пользователей подключается к более новым экземплярам сервиса, а часть — к старым (в зависимости от того, как направляются запросы к этим экземплярам).

Исправление багов при скользящем обновлении

Если обнаруживается баг, команда Plenty of Woofs осуществляет поиск исправления, а затем выпускает новый релиз, содержащий это исправление, по-прежнему используя *скользящее обновление*:

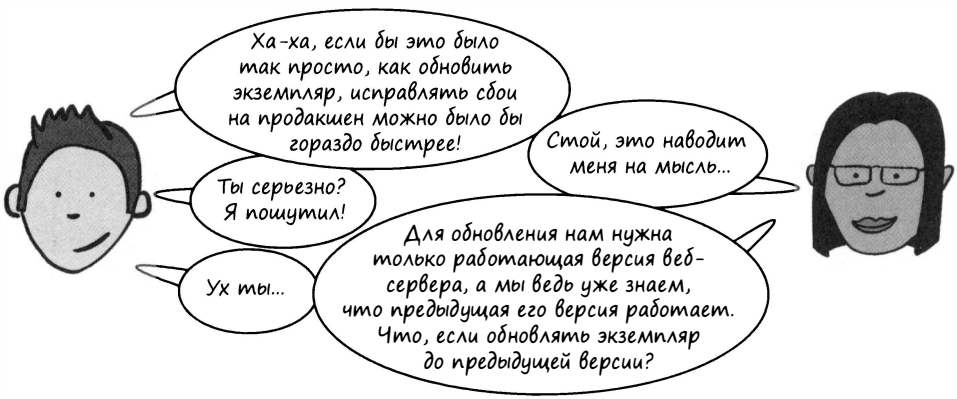


И конечно, скользящее обновление для исправления багов запускается только после того, как баг обнаружен и исправлен и новая версия веб-сервера собрана. Таким образом, общее время устранения сбоя в работе сайта Plenty of Woofs составляет:

(Время на исправление бага) + (Время на создание нового релиза) + + 3 × (Время на обновление экземпляра)

Откаты

Если посмотреть на то, как происходит развертывание и куда уходит время, станет очевидно, что большая часть времени тратится на ожидание нового релиза:



Сарита и Арчи поняли, что когда развертывание приводит к сбою, не нужно оставлять сервис в неработающем состоянии и ждать исправления. Вместо этого можно сразу *откатиться* к предыдущей версии, которая, как известно, работает.

Поэтому если они производят развертывание версии 1.4.0 и оно завершается сбоем, вместо того, чтобы ждать версии 1.4.1 (то есть *исправленной*), можно вернуться к предыдущей версии (1.3.2). Как только ошибка будет исправлена и версия 1.4.1 станет доступна, сервис можно будет обновить с версии 1.3.2 до версии 1.4.1.

Откат = немедленное улучшение

Если команда Plenty of Woofs начнет применять стратегию отката при возникновении сбоев, время восстановления работоспособности сервиса резко сократится и будет равно времени, необходимому для отката до предыдущей версии при скользящем обновлении:

~~(Время на исправление бага) + (Время на создание нового релиза) +~~
+ 3 × (Время на обновление экземпляра)

Минуты

Время восстановления сокращается с нескольких дней до нескольких минут. Это сразу улучшает показатели DORA. *Время на восстановление работоспособности сервиса* уменьшится с нескольких дней или более до нескольких минут.

Метрика	Суперэффективные	Высокоэффективные	Среднеэффективные	Низкоэффективные
Время восстановления работоспособности сервиса	Менее одного часа	Менее одного дня	От одного дня до одной недели	Более 6 месяцев

Команда Plenty of Woofs сразу же перейдет из среднеэффективных в суперэффективные!

Сарита и Арчи немедленно сообщают о своем открытии остальным сотрудникам, и Plenty of Woofs вводит политику немедленного отката при возникновении сбоев. Это означает, что основной баг остается в коде и должен быть исправлен, но теперь это можно сделать спокойно в рабочее время, а не впопыхах, когда нужно как можно быстрее исправить рабочую версию.

Словарик

Чтобы эффективно использовать *откаты*, нужна стратегия. *Стратегия отката* — это автоматизированный, задокументированный и протестированный процесс, указывающий, как выполнять откат. К этому процессу следует относиться так же серьезно, как к любому другому процессу разработки или развертывания, если не более, потому что откат будет происходить в наиболее чувствительное для сервиса время (когда он неисправен и создает проблемы у всех пользователей). Автоматизация и сопутствующие тесты (и даже «пожарные учения» для отработки отката) позволят проводить процесс гладко в самых критических случаях.

А что насчет отката данных?

Когда развертывание включает изменения данных, откат становится немного сложнее. И что вполне естественно, большинство сервисов в той или иной форме содержат данные, схему взаимодействия с которыми необходимо обновлять по мере добавления новых функций и исправления багов.

К счастью, эта проблема решаема и не помешает использовать откаты. Чтобы обеспечить безопасное обновление (то есть развертывание) и откат по мере необходимости, потребуется внедрить несколько политик и рекомендаций в процесс:

- *Версионирование схем данных* — подобно тому, как в предыдущей главе мы рекомендовали управлять версиями продукта (и семантическое версионирование здесь тоже может пригодиться), каждое изменение схемы базы данных должно сопровождаться соответствующим изменением версии.
- *Создание скриптов обновления и отката для каждого перехода на новую версию* — каждое изменение схемы базы данных должно сопровождаться двумя скриптами (или другими средствами автоматизации): одним — для перехода от предыдущей версии к новой и вторым — для отката с новой версии к предыдущей. Если вы создаете их для каждого изменения версии, вы сможете легко откатиться назад, насколько это необходимо, применяя скрипты по очереди при откате назад или для перехода на более позднюю версию.
- *Обновление данных и сервисов по отдельности* — если объединять обновление данных и сервисов и выполнять их одновременно, откат назад станет проблематичнее и чаще будет приводить к ошибкам. Выполняйте обновление по отдельности. Когда вы вносите изменения, сервисы должны без ошибок обрабатывать как старые, так и новые версии данных. Добиться этого непросто, но это стоит сделать, чтобы упростить процесс и понизить риски.

Основная цель внедрения этих политик — осуществлять развертывание и откат изменений данных таким же образом, как для программ: проводить развертывание, когда требуется обновление, а если что-то пойдет не так, откатить все назад.

Аналогично, когда изменения функциональности требуют соответствующих изменений в структуре данных, можно обновить БД, добавив необходимые изменения. Опять же, постарайтесь не вносить эти изменения одновременно, чтобы откаты выполнялись независимо друг от друга.

В этой книге мы не будем подробно говорить о том, как работать с данными; чтобы узнать больше по этой теме, обратитесь к источникам, посвященным эффективному управлению данными.

Политика откатов в действии

Новая политика немедленного отката при возникновении сбоев (чтобы не ждать, пока команда создаст исправление) уже сэкономила Plenty of Woofs немало времени — и нервов! По мере внедрения этой политики Сарита и Арчи собирают данные и повторно анализируют метрику DORA «время восстановления работоспособности сервиса», чтобы подтвердить свою теорию о том, что это время сократится до минут и они попадут в категорию суперэффективных компаний. Изучая, что на самом деле происходит во время сбоя, они обнаружили, что общее время с момента начала сбоя до момента восстановления сервиса таково:

- Скользящее обновление трех экземпляров занимает 5–15 минут как при переходе к следующей версии (развертывание/обновление), так и при переходе к предыдущей (откат).
- Сбои возникают в начале скользящего обновления (после обновления первого экземпляра любой трафик, попадающий на этот экземпляр, начнет вызывать сбои).
- Для обнаружения сбоя требуется не менее 3 минут, а иногда и 10 минут.
- Как только становится ясно, что произошел сбой, инициируется откат. Сам откат представляет собой скользящее обновление всех трех экземпляров, и для его завершения потребуется еще 5–15 минут.

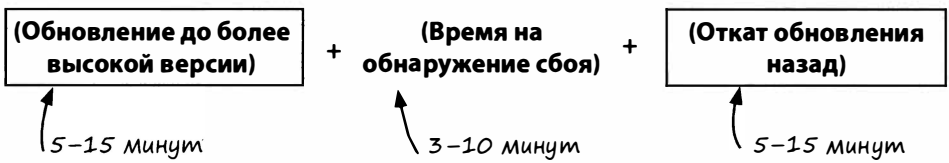
Таким образом, общее время от начала сбоя до решения проблемы составляет:

Таким образом, общее время от начала сбоя до его устранения составляет от 13 (5 + 3 + 5) до 40 (15 + 10 + 15) минут. Это означает, что метрика DORA «Время восстановления работоспособности сервиса» находится в интервале от 13 до 40 минут. Время на восстановление в 40 минут позволяет отнести компанию Plenty of Woofs к категории суперэффективных:

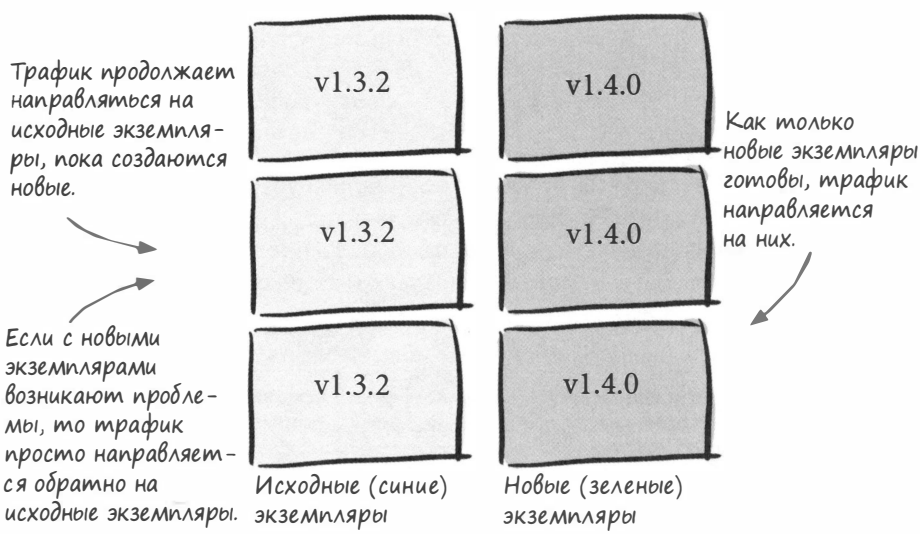
Метрика	Суперэффективные	Высокоэффективные	Среднеэффективные	Низкоэффективные
Время восстановления работоспособности сервиса	Менее одного часа	Менее одного дня	От одного дня до одной недели	Более 6 месяцев

Сине-зеленые развертывания

Хотя метрика времени восстановления работоспособности значительно улучшилась, 40 минут потенциально означают 40 минут ошибок на стороне пользователя, а между максимальным значением в 40 минут и верхним пределом в 1 час не такая большая разница. Если что-то в процессе замедлится (например, если добавить еще несколько экземпляров сервера), то команда снова станет просто высокоэффективной. Сарита предлагает сосредоточиться на самом времени выполнения скользящих обновлений и посмотреть, можно ли что-то улучшить:



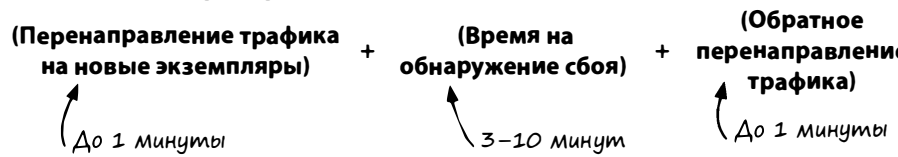
Сам процесс скользящего обновления добавляет от 6 до 20 минут ко времени простоя. Одним из способов сократить его является использование сине-зеленого развертывания вместо скользящего обновления. При *сине-зеленом развертывании* (также называемом *красно-черным развертыванием*) вместо обновления одного набора экземпляров по месту его размещения создается абсолютно новый набор экземпляров, который и обновляется. Только после того, как эти экземпляры будут готовы, трафик переключается с оригинальных экземпляров («синих», хотя на самом деле цвет значения не имеет) на новые («зеленые»).



Если с новыми экземплярами возникнут проблемы или произойдет сбой, то исправный набор экземпляров предыдущей версии будет продолжать работать. Откатиться назад будет так же просто, как переключить трафик на исходный набор экземпляров!

Быстрое восстановление при сине-зеленом развертывании

При сине-зеленом развертывании должно быть достаточно оборудования, чтобы одновременно запускать два полных набора экземпляров. Такой подход может оказаться нецелесообразным, если у вас собственное оборудование (если только у вас нет запасного). Но если вы пользуетесь услугами облачного провайдера, выделить дополнительные ресурсы довольно дешево и легко; за них нужно будет платить только в период развертывания. Сарита и Арчи оценили, сколько времени займет восстановление работоспособности сервиса, если Plenty of Woofs начнет использовать сине-зеленые развертывания вместо скользящих обновлений:



Время, затрачиваемое на скользящее обновление, теперь заменяется на время, необходимое для перенаправления трафика между экземплярами. Таким образом, общее время от начала сбоя до его устранения составляет от 5 (1 + 3 + 1) до 12 (1 + + 10 + 1) минут.

Сколько времени требуется для перенаправления трафика?

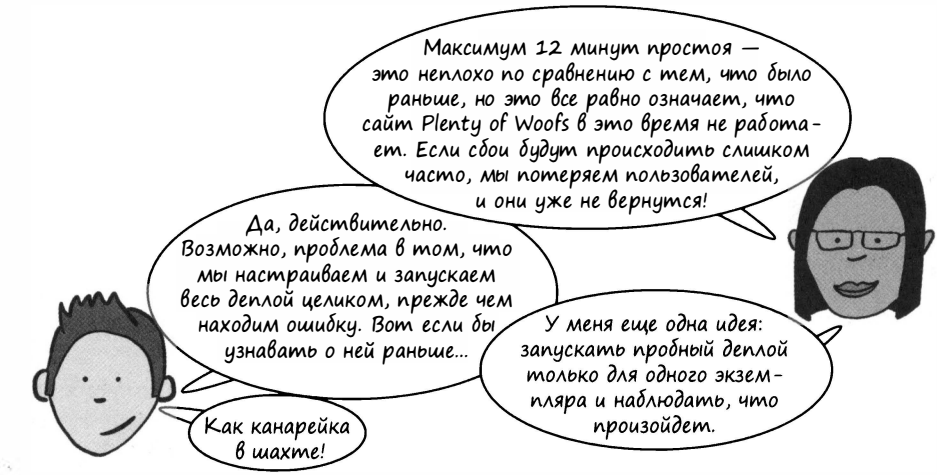
Грамотное перенаправление трафика занимает какое-то время. Грубый подход — прервать все соединения со старыми экземплярами в момент переключения, что в лучшем случае означает, что все пользователи, подключенные к этим экземплярам, увидят ошибку соединения, а в худшем — что прервется какой-то процесс, в результате чего какой-то функционал перестанет работать (проектирование приложений, позволяющее избежать подобных ситуаций, — это отдельная тема!). Один из способов корректного перенаправления трафика — позволить экземплярам *закончить обработку* в течение определенного времени, зависящего от того, как долго обычно обслуживаются запросы. (В Plenty of Woofs запросы выполняются максимум за несколько секунд, поэтому время ожидания установлено на 1 минуту.) Разгрузите экземпляры, позволив всем уже начатым запросам завершиться (не прерывайте их, если только они не завершатся за время ожидания!), и направьте новый трафик на новые экземпляры.

ВАЖНО

Политика откатов при возникновении сбоев на продакшен (чтобы не ждать исправлений и только затем переходить к новой версии) позволит быстро и безболезненно восстанавливать работоспособность продукта. А сине-зеленое развертывание, если у вас достаточно оборудования, сделает этот процесс еще быстрее и проще.

Быстрее и стабильнее с «канарейками»

Сине-зеленые развертывания вывели время восстановления Plenty of Woof на довольно хороший уровень, но Сарита задается вопросом, можно ли сделать еще лучше:



Сарита и Арчи определили еще одну подходящую стратегию развертывания: *канареечные развертывания* (также называемые *канареечными выкатываниями* или *канареечными релизами*). При канареечном развертывании обновляется один экземпляр (называемый *канарейкой*) и на него направляется небольшой процент трафика.

Выражение «канареечный релиз» пошло от канареек, которых спускали в угольные шахты, чтобы определять, есть ли там токсичные газы. Бедные канарейки!

Если «канарейка» в порядке, развертывание можно продолжить либо переключив весь трафик на экземпляры с новой версией, либо постепенно создавая все больше канареечных экземпляров и направляя на них все больше трафика, пока он не перестанет поступать на старые экземпляры. Если «канарейка» не работает, процесс можно остановить, а весь трафик переключить обратно на исходные экземпляры.

Требования к канареечным развертываниям

Чтобы канареечные развертывания были успешными, необходимо сделать следующее:

- *Распределить трафик между развертываниями* (например, настроив балансировщик нагрузки).
- *Измерить и определить успешность развертывания* (в идеале автоматически).
- *Отделить изменения данных от изменений функциональности* (см. предыдущую врезку «А что насчет отката данных?»).
- *Определить эффективный процент трафика, направляемого на «канарейку»*. Чтобы сигнал об успешности обновления был достоверным, на канареечный экземпляр должно направляться достаточно трафика, но с другой стороны, количество трафика должно быть минимальным, чтобы пользователи не испытали неудобств, если что-то пойдет не так.

Независимо от объема трафика, который вы получаете, лучше предположить, что нескольких секунд для сбора данных недостаточно. Многие метрики даже не измерить за это время (например, метрику «запросы в минуту»), а способ агрегирования этих метрик предполагает, что для четкого понимания происходящего нужно оценить несколько значений (например, усреднение значений за минуту позволяет сгладить разброс данных).

- *Определить время, необходимое для сбора данных с канареечного экземпляра.* Опять же, чтобы сигнал об успешности обновления был достоверным, необходимо собирать данные достаточно долго, иначе вывод об успешности обновления может быть ошибочным, но слишком длительный процесс сбора данных может замедлить реагирование при возникновении проблемы.

Как измерить и определить успешность развертывания?

Измерив работоспособность сервиса. Чтобы понять, что и как измерять, обратитесь к литературе по DevOps или по обеспечению надежности сайта. Книга «Site Reliability Engineering: How Google Runs Production Systems»[1] под редакцией Бетси Бейер (O'Reilly, 2016) описывает, например, *четыре золотых сигнала*, на которые следует обращать внимание при мониторинге распределенных систем (задержка, трафик, ошибки и насыщение). Чтобы автоматизировать сбор этих показателей и стратегии развертывания, применяйте существующие инструменты (например, Spinnaker, инструмент с открытым исходным кодом с поддержкой нескольких облачных провайдеров и сред) или создайте свой собственный. Создание собственного инструмента — самый дорогой вариант, а логика, необходимая для его поддержки, достаточно сложна, поэтому вам понадобится команда для его создания и обслуживания. Выбирайте этот путь, только если не можете найти подходящие инструменты (например, если вы создали свое собственное (проприетарное) облако).

[1] Бетси Бейер и др. «Site Reliability Engineering. Надежность и безотказность как в Google». Санкт-Петербург, издательство «Питер».

«Канарейки» и «конфигурация как код»

В главе 3 вы узнали, что важно использовать принцип «*конфигурация как код*» как можно чаще: рассматривать конфигурацию (и вообще все текстовые данные, необходимые для работы продукта) «как код», храня ее в системе контроля версий и применяя к ней непрерывную интеграцию. Но такая стратегия, как канареечное развертывание, требует внесения инкрементных обновлений в конфигурацию, используемую для запуска сервисов, по мере развертывания:

- обновить конфигурацию, описывающую экземпляры, включив новый экземпляр для запуска «канарейки» с новой версией продукта;
- обновить конфигурацию для маршрутизации трафика, направив часть трафика на канареечный экземпляр;
- если «канарейка» не работает, обновить конфигурацию, описывающую экземпляры, удалив «канарейку» и маршрутизацию трафика, направив весь трафик обратно на исходные экземпляры;
- если «канарейка» успешно работает, обновить конфигурацию так, чтобы в нее добавились все необходимые новые экземпляры, и переключить трафик на них.

Имеет ли смысл использовать принцип «конфигурация как код» при выполнении всех этих обновлений? Да! Продолжайте использовать этот принцип как можно чаще. Есть несколько способов заставить его работать с новыми стратегиями развертывания:

- По мере внесения изменений в конфигурацию фиксировать их в репозитории, где хранится конфигурация (используйте автоматизированный пайплайн с задачами, которые будут создавать и сливать PR-запросы в репозиторий с изменениями по мере развертывания).
- Фиксировать изменения в репозитории до того, как они будут развернуты в рабочей среде, и инициировать изменения в продакшен-конфигурации на основе изменений в коде. (Такой подход, особенно в контексте Kubernetes, часто называют *GitOps*.)

Подход GitOps может принести пользу и в другом отношении. Он позволяет использовать репозиторий конфигурации как шлюз для внесения изменений в рабочую версию продукта, гарантируя, что она всегда остается синхронизированной с конфигурацией в системе контроля версий и что все изменения в ней проходят проверку кода и CI. Подробнее см. ресурсы по GitOps и инструментам типа ArgoCD.

Канареечное развертывание параллельно с базовым

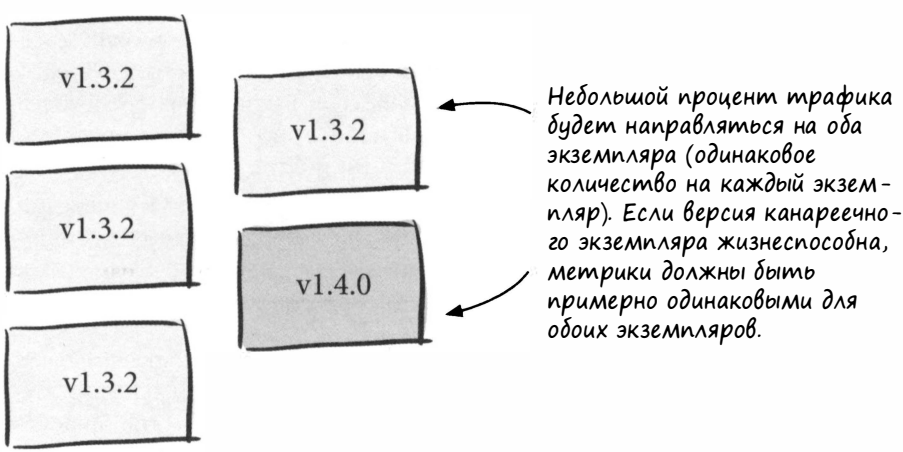
Чтобы представление об успешности обновления было точным, важно сравнивать сопоставимые понятия. При сравнении производительности новой и старой версии продукта необходимо, чтобы как можно больше переменных были одинаковыми, за исключением самого обновления.

В большинстве случаев при канареечном развертывании сравнение не совсем справедливо: например, неоправданно ожидать от совершенно нового, только что запущенного экземпляра такой же производительности, как от экземпляров, которые уже работают несколько часов или дней. Ниже перечислены факторы, которые чаще всего вызывают различия в производительности между канареечным и исходным экземплярами:

1. Только что запущенный экземпляр (например, до полной инициализации кэшей в оперативной памяти) нельзя сравнивать с экземпляром, который работает уже некоторое время.
2. Нельзя сравнивать обработку разных объемов трафика.
3. Работу в качестве отдельного экземпляра нельзя сравнивать с работой в составе более крупного кластера.

Все эти факторы, естественно, будут отличать канареечный экземпляр, который был запущен недавно, обрабатывает лишь небольшую часть общего трафика и не является частью кластера, достаточно большого, чтобы справиться с рабочей нагрузкой.

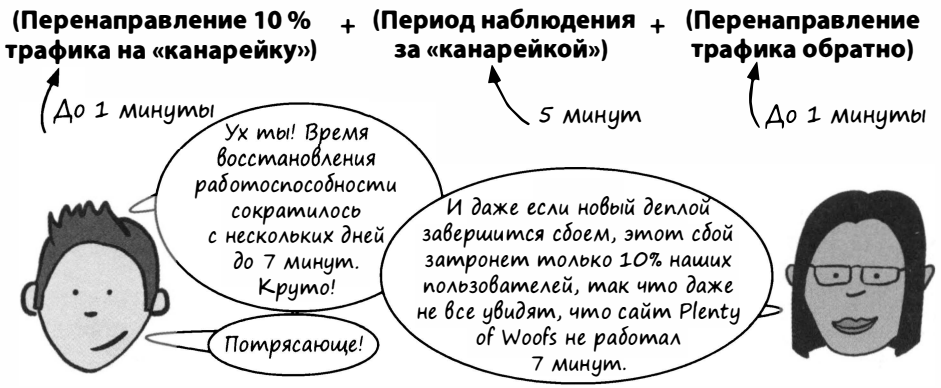
Так что же делать? Одним из решений является загрузка параллельно с канареечным экземпляром *базового экземпляра*, на котором работает предыдущая версия продукта. Затем, чтобы определить, было ли обновление успешным, сравните напрямую метрики «канарейки» с метриками базовой версии. Это поможет получить гораздо более точное представление об эффективности канареечного экземпляра.



Время восстановления работоспособности при канареечном развертывании

Сарита и Арчи рекомендуют команде Plenty of Woofs перейти на канареечные развертывания. К счастью, они находят инструмент, который им подходит (Deployaker, тот же инструмент, который использует команда Watch Me Watch в главе 3), и им не нужно создавать собственное решение. Они изучают время восстановления работоспособности сервиса после использования «канареек» в течение нескольких развертываний:

Deployaker — это вымышленный аналог Spinnaker ;)



Метрики DORA и «канарейки»

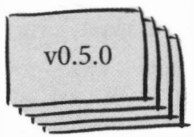
Если сбой затрагивает лишь небольшой процент пользователей и сервис можно быстро восстановить, отключив неисправный канареечный экземпляр, считать ли это сбоем продакшена? Метрики DORA и сопутствующая литература не слишком углубляются в этот вопрос. Они называют отказы «деградацией сервиса» или «ухудшением сервиса». Эти определения предполагают, что воздействие даже на небольшой процент пользователей считается сбоем продакшена и учитывается в метрике «частота сбоев при изменениях». Но вы вольны использовать любое определение сбоя. Важно не просто попасть в суперэффективную категорию, а реализовывать бизнес-ценность продукта. В методологии невозможно применять единый подход во всех сценариях, особенно когда речь идет о допустимости сбоев продакшена.

ВАЖНО

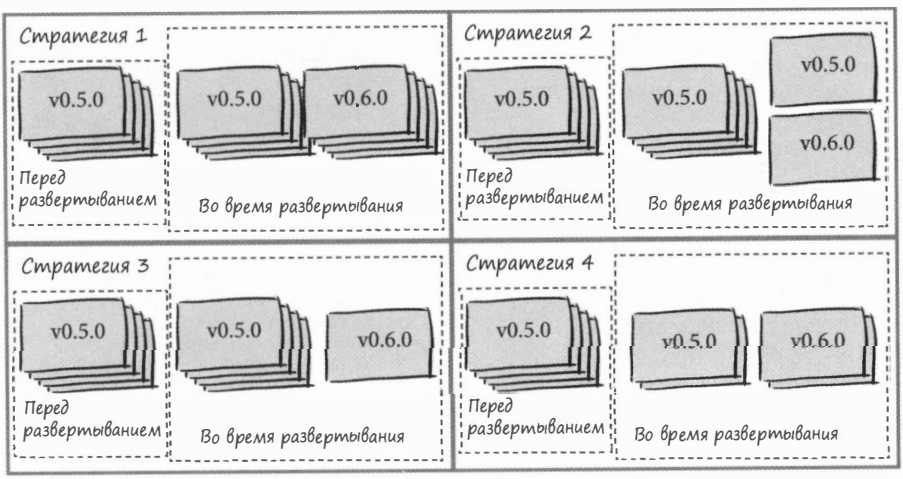
Использование канареечных развертываний снижает опасность, которую несут в себе неудачные развертывания, за счет сокращения количества пользователей, которых они затрагивают, а также за счет возможности быстро и просто сделать откат.

Ваш ход: определите стратегию

На каждой из следующих диаграмм показано обновление кластера из четырех экземпляров с использованием одной из четырех стратегий развертывания, которые мы только что рассмотрели (скользящее обновление, сине-зеленое развертывание, канареечное развертывание и канареечное развертывание параллельно с базовым). Сопоставьте каждую диаграмму с соответствующей стратегией развертывания.



Перед развертыванием: исходный кластер с четырьмя экземплярами работающей версии v0.5.0



Ответы

1. Стратегия 1: сине-зеленое развертывание
2. Стратегия 2: канареечное развертывание параллельно с базовым
3. Стратегия 3: канареечное развертывание
4. Стратегия 4: скользящее обновление

Увеличение частоты развертывания

Значительно улучшив метрику времени восстановления работоспособности сервиса, Сарита и Арчи вернулись к списку метрик DORA для Plenty of Woofs:

- *Частота развертывания* — один раз в неделю (*высокая*).
- *Время доставки изменений* — меньше одной недели (*высокий показатель*).
- *Время восстановления работоспособности сервиса* — примерно 7 минут (*наивысший показатель*).
- *Частота сбоев при изменениях* — одно из трех развертываний, или около 33 % (*низкий уровень*).

Сарита и Арчи предлагают остальным разработчикам перейти от еженедельных релизов к ежедневным. Канареечное развертывание имело успех (уровень стресса в команде понизился, и ей больше не нужно работать сверхурочно), поэтому все единодушно соглашаются.

Ранее Сарита и Арчи проанализировали 8 недель еженедельных развертываний и подсчитали, что 4 из 40 ежедневных развертываний за тот же период времени завершились бы сбоем:

- Неделя 1 — сбои в 1/5 ежедневных развертываний
- Неделя 2 — сбои в 0/5 ежедневных развертываний
- Неделя 3 — сбои в 2/5 ежедневных развертываний
- Неделя 4 — сбои в 0/5 ежедневных развертываний
- Неделя 5 — сбои в 0/5 ежедневных развертываний
- Неделя 6 — сбои в 0/5 ежедневных развертываний
- Неделя 7 — сбои в 1/5 ежедневных развертываний
- Неделя 8 — сбои в 0/5 ежедневных развертываний

В начале работы над улучшением развертывания сервиса Plenty of Woof они увидели, что за 8 недель три еженедельных развертывания завершились сбоем. Исходя из дней недели, когда обнаруживались баги, вызывающие сбои, они подсчитали, сколько сбоев могут вызвать ежедневные развертывания.

По мере того как Plenty of Woofs переходит к ежедневному развертыванию, они видят, что их прогнозы оказались довольно точными: 5 из 40, или 12,5 %, развертываний в течение следующих 8 недель завершались сбоями — и на устранение сбоев уходило примерно по 7 минут! Причем за это время никому не пришлось работать на выходных.

Метрики DORA при ежедневных канареечных развертываниях

Теперь команда Plenty of Woofs выполняет ежедневные канареечные развертывания, и метрики DORA выглядят следующим образом:

- *Частота развертывания* — ежедневно (*высокая*).
- *Время доставки изменений* — меньше одного дня (*высокий показатель*).
- *Время восстановления работоспособности сервиса* — 7 минут (*наивысший, «элитный» показатель*).
- *Частота сбоев при изменениях* — 12,5 % (*низкий уровень*).

Из таблицы метрик DORA для супер-, высоко-, средне- и низкоэффективных команд видно, что Plenty of Woofs заметно улучшила свое положение:

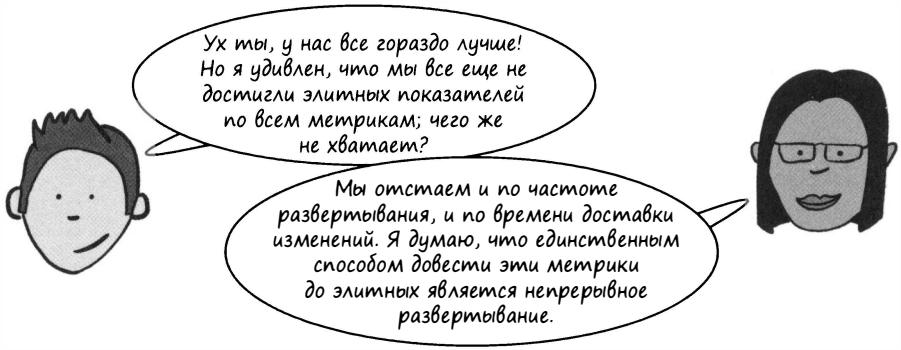
Метрика	Супер-эффективные	Высоко-эффективные	Средне-эффективные	Низко-эффективные
Частота развертывания	Несколько раз в день	От одного раза в неделю до одного раза в месяц	От одного раза в месяц до одного раза в полгода	Реже, чем раз в полгода
Время доставки изменений	Менее одного часа	От одного дня до одной недели	От одного месяца до полугода	Более полугода
Время восстановления работоспособности сервиса	Менее одного часа	Менее одного дня	От одного дня до одной недели	Более 6 месяцев
Частота сбоев при изменениях	0–15%	16–30%	16–30%	16–30%

Разработчики прошли путь от высокой, средней и низкой производительности до однозначно высоких показателей и даже до супервысоких в двух из четырех метрик!

ВАЖНО

При использовании надежной стратегии восстановления сервиса в случае неудачного развертывания (например, стратегии канареечного развертывания) частоту развертывания можно смело увеличить, а это положительно скажется и на других показателях!

Непрерывное развертывание



Сарита предполагает, что для достижения суперэффективных (элитных) показателей по всем метрикам DORA необходимо начать выполнять *непрерывное развертывание*. Так ли это?

Давненько я подробно не рассказывала о непрерывном развертывании (с самой первой главы!). Его путают с непрерывной доставкой и тоже часто называют *CD* (в этой книге аббревиатура *CD* используется исключительно для обозначения *непрерывной доставки*). *Непрерывное развертывание* — это методика программной разработки, при которой:

> **выпуск релиза рабочей версии продукта для пользователей осуществляется автоматически при каждом коммите.**

При непрерывном развертывании каждый коммит, внесенный в систему контроля версий, запускает процесс развертывания. В компании Plenty of Woofs развертывание происходит один раз в день, но каждое развертывание может содержать несколько коммитов, то есть все коммиты, которые были слиты и добавлены в главную ветку веб-сервера в этот день.

Две метрики DORA, которые в Plenty of Woofs пока не дотягивают до суперэффективных, — это частота развертывания и время доставки изменений, которые тесно связаны и непосредственно влияют друг на друга:

Метрика	Суперэффективные	Высокоэффективные
Частота развертывания	Несколько раз в день	От одного раза в неделю до одного раза в месяц
Время доставки изменений	Менее часа	От одного дня до одной недели

Передача изменений в рабочий продукт менее чем за час требует частоты развертывания несколько раз в день (предполагая, что каждый день происходит слияние нескольких изменений).

Чтобы стать суперэффективной, Plenty of Woofs должна выпускать развертывания еще чаще, например каждый час. Но в этом случае они окажутся такими частыми, что с тем же успехом можно использовать непрерывное развертывание (которое, как вы увидите ниже, упрощает поддержку кода в состоянии готовности к релизу).

Когда использовать непрерывное развертывание

Непрерывное развертывание имеет свои преимущества, но подходит не для каждого проекта. Его не обязательно использовать, чтобы наладить непрерывную доставку.

Возможно, из главы 1 вы помните, что цель CD — достичь состояния, при котором можно выпускать релизы в любое время, и автоматизировать этот процесс (чтобы выпустить релиз «было так же просто, как нажать кнопку»). Если вы это сделали, у вас, по сути, есть все необходимое для непрерывного развертывания.

Но не всегда стоит заходить так далеко. Важно проанализировать проект и решить, подходит ли вам непрерывное развертывание. Чтобы выполнять его, проект должен соответствовать следующим условиям:

- *Допускается определенный процент (небольшой) невыполненных запросов к сервису.* Безопасность непрерывного развертывания зависит от способности обнаруживать и быстро устранять сбои в рабочем продукте, однако для некоторых программных продуктов цена сбоев слишком высока. Конечно, неоходимо минимизировать последствия сбоя, но иногда риск, связанный с допущением сбоя, слишком велик (например, если сбой может повлиять на здоровье человека).
- *Нормативные требования не препятствуют его проведению.* В зависимости от предметной области, продукт, возможно, должен соответствовать нормативным требованиям, которые делают невозможным выпуск релиза при каждом изменении.
- *Исследовательское тестирование перед релизом не требуется.* Этот вид тестирования должен блокировать выпуск релизов только в том случае, если сбои связаны с достаточно высоким риском (подробнее об этом ниже), но если блокировка все же необходима, то она настолько замедлит релиз, что непрерывное развертывание окажется просто невозможным.
- *Перед релизом не требуется явное одобрение.* Выпуски релиза могут требовать явного одобрения руководства. Старайтесь этого избегать (например, придумайте, как автоматизировать все, что должно утверждать руководство), но если это требование сохранится, выполнять непрерывное развертывание не получится.
- *Релизы не требуют аппаратных изменений.* Если обновления продукта требуют соответствующих обновлений оборудования (например, если вы работаете над встроенным ПО), вы, скорее всего, не сможете использовать непрерывное развертывание.

Если один или несколько из этих факторов исключают возможность непрерывного развертывания, не расстраивайтесь! На нем свет клином не сошелся, и улучшить процессы CD, сделать их гладкими и удобными в работе можно и без него (о чем говорится далее в этой книге).

Обязательные этапы контроля качества

Зачастую внедрить непрерывное развертывание мешает требование проводить исследовательское или QA-тестирование (QA, Quality Assurance — обеспечение качества) перед выпуском продукта на продакшен. Обычно в этом процессе задействовано несколько программных сред:

1. PR проверяются, и CI верифицирует изменения.
2. Изменения сливаются в код.
3. Обновленный продукт развертывается в *тестовой среде*.
4. QA-аналитики изучают продукт в тестовой среде и ищут ошибки.
5. Только после этапа QA продукт может быть развернут в рабочей среде.

При работе с несколькими средами старайтесь использовать для них один метод автоматического развертывания. Использование разных методов облегчает проникновение багов, не проверенных в этих средах.

Словарик

Исследовательское тестирование (exploratory testing) — это процесс ручной проверки продукта на наличие непредвиденных проблем. Тестирование с участием человека бесценно при выявлении таких проблем. Какими бы полезными ни были автоматизированные тесты, они, скорее всего, будут выявлять только те проблемы, которые вы уже предвидели заранее (и систематизировали в виде тестов). Чтобы выявить действительно неожиданные ситуации, при которых что-то пойдет не так, нужны люди, которые будут проводить эксперименты с продуктом.

Словарик

Среды (они же *среды выполнения*, runtime environments), которые здесь обсуждаются, — это машины, на которых выполняется продукт. Например, в *рабочей* среде (*продакшен*, сокращенно prod) с продуктом взаимодействуют конечные пользователи. Каждый разработчик может иметь свою собственную среду *разработки* (development, сокращенно dev), в которой он осуществляет развертывание продукта по мере работы над ним. Самые распространенные типы сред — промежуточная (иначе называемая предпродакшен, или стейдж), тестовая, рабочая (иначе называемая продакшен) и среда разработки: *staging, test, prod, dev*.

Контроль качества и непрерывное развертывание

Наличие этапа контроля качества (или любого другого этапа *ручного подтверждения*), который блокирует развертывание продукта, не позволит осуществлять непрерывное развертывание и будет определять максимально достижимую верхнюю границу частоты развертывания. Если у вас есть такое ограничение, обратите внимание на следующее:

- *Какова цена сбоя?* Опять же, если цена сбоя в процессе эксплуатации слишком высока (например, жизнь человека!), имеет смысл проявить максимальную осторожность и все же включить этап QA.
- *Проводит ли группа контроля качества исследовательское тестирование?* Если да, то один из вариантов — продолжать это тестирование параллельно, но при этом не блокировать релизы, что позволит использовать преимущества обоих подходов.
- *Возможно ли автоматизировать тестирование?* Если специалисты, проводящие ручную проверку, при поиске проблем используют чек-лист ранее замеченных признаков сбоя, скорее всего, этот процесс можно автоматизировать. А если его можно автоматизировать, то его можно выполнять как часть пайплайна CD.

В общем, чем больше вы сможете автоматизировать, тем меньше сред и специалистов вам потребуется и тем больше людей смогут делать то, что у них получается лучше всего, а не просто следовать чек-листу.

И всегда остерегайтесь FUD (страх, неуверенность и сомнения)! Часто этап QA делают обязательным только потому, что люди боятся ошибиться. Но если вы можете уменьшить влияние ошибок с помощью безопасных стратегий развертывания и если в процессе эксплуатации продукта допустимо появление некоторых недочетов, то вам, вероятно, не понадобится этап QA. Не позволяйте страху вам мешать.

Поддержание кода в состоянии готовности к релизу

Еще одна веская причина применять непрерывное развертывание, если это возможно, — использовать его для поддержания кода в состоянии готовности к релизу. Стратегия немедленного отката хороша с точки зрения поддержания стабильности развертывания, но после отката код, вызвавший сбой, остается в базе. Это означает, что сразу после отката код уже непригоден для релиза (технически он был таким и до развертывания, но узнать об этом было невозможно). Цель CI — поддерживать такое состояние, чтобы:

безопасно вносить изменения в продукт в любое время.

Как же после отката вернуться в состояние готовности к релизу? Непрерывное развертывание действительно с этим справляется: если развертывание выполняется после каждого изменения и однажды приводит к сбою, то легко отследить изменение, которое его вызвало. А если пойти еще дальше, то можно автоматически отменить изменение или, по крайней мере, автоматически создать PR для отмены изменения, а проводить анализ и принимать решение будет человек.

В отсутствие CD развертывания, скорее всего, будут включать множество изменений, и после отката кто-то должен будет просмотреть все эти изменения, решить, какое из них вызвало сбой, и определить, как вернуть все обратно или исправить ошибку. В течение этого времени код будет оставаться непригодным для релиза.

ВАЖНО

У непрерывного развертывания много преимуществ, но оно годится не на все случаи жизни, и не стоит ожидать, что его можно будет применить в любом проекте.

Суперэффективность согласно метрикам DORA

Компания Plenty of Woofs решает попробовать непрерывное развертывание. Разработчики настроили его запуск из системы контроля версий, которая инициирует канареечное развертывание с помощью Deployaker каждый раз, когда PR сливается в код веб-сервера.

И они довольны результатами! Их метрики DORA на данный момент выглядят так:

- *Частота развертывания* — несколько раз в день (*элитная*).
- *Время доставки изменений* — после внесения изменений запускается пайплайн выпуска, и развертывание завершается менее чем за 1 час (*элитное*).
- *Время восстановления работоспособности сервиса* — 7 минут (*элитное*).
- *Частота сбоев при изменениях* — 12,5 % (*низкий уровень*).

Теперь команда Plenty of Woofs считается суперэффективной по всем показателям:

Метрика	Суперэффективные
Частота развертывания	Несколько раз в день
Время доставки изменений	Менее одного часа
Время восстановления работоспособности сервиса	Менее одного часа
Частота сбоев при изменениях	0–15%

Развертывания теперь проходят настолько гладко и просто, что большинство сотрудников быстро начинают относиться к ним как к обычным процессам. Это гораздо лучше, чем эмоциональное выгорание, грозившее команде, когда Сарита и Арчи начинали работу!

Заключение

Еженедельные развертывания в компании Plenty of Woofs вызывали у разработчиков стресс и эмоциональное выгорание. Первая естественная реакция на такую ситуацию — проводить развертывания реже. Но руководствуясь метриками DORA, члены команды сразу поняли, что на самом деле лучше проводить развертывания чаще, а не реже.

Внедрение стратегии отката изменений вместо немедленного исправления ошибок дало им большую свободу действий. После этого они изучили более сложные стратегии, такие как сине-зеленые и канареечные развертывания.

В конце концов компания Plenty of Woofs решила внедрить непрерывное развертывание, тем самым попав в категорию суперэффективных (элитных) команд согласно DORA-метрикам и добившись того, что развертывание превратилось из события, которого все боялись, в обычную работу.

Итоги

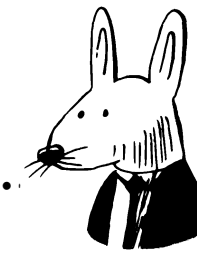
- Частота сбоев при изменениях — одна из двух метрик DORA для стабильности, показывающая, как часто развертывания на продакшен приводят к сбоям сервиса.
- Время восстановления работоспособности сервиса — вторая метрика DORA для стабильности, определяющая, сколько времени требуется, чтобы вывести сервис из нерабочего состояния.
- Более частое развертывание снижает степень риска при каждом конкретном развертывании.
- Немедленный откат при возникновении проблем в работе продукта позволяет быстро и безболезненно провести восстановление, спокойно устраняя основную проблему.
- Сине-зеленые и канареечные развертывания снижают негативное влияние сбоев на пользователей и обеспечивают быстрое восстановление.
- После внедрения таких процессов, как сине-зеленое и канареечное развертывание, можно смело увеличивать частоту развертываний.
- У непрерывного развертывания много преимуществ, но оно годится не на все случаи жизни, и не стоит ожидать, что его можно будет применить в любом проекте.

Далее...

В следующей главе мы углубимся в вопросы, связанные с реализацией пайплайнов. Вы на практике узнаете, как быстро проделать путь от абсолютного новичка до эксперта непрерывной доставки.

В заключительной части мы рассмотрим общие концепции непрерывной доставки.

Часть 4
Реализация непрерывной доставки

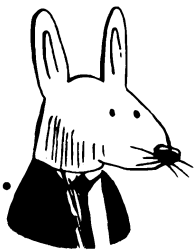


В главе 11 мы вернемся к элементам непрерывной доставки, которые изучались в предыдущих главах, и узнаем, как эффективно применять их в проектах, реализуемых с нуля, и в уже существующих проектах.

В главе 12 основное внимание уделено «рабочей лошадке», часто лежащей в основе любой автоматизации непрерывной доставки: shell-скриптам. Вы увидите, как применять передовые методики, которые мы использовали для остальной части кода, к скриптам, обеспечивающим безопасную и корректную доставку программного продукта.

В главе 13 представлена общая структура автоматизированных пайплайнов, которые необходимы для поддержки непрерывной доставки. Пайплайны, которые мы рассмотрим, иллюстрируют функционал систем автоматизации CD, требуемый для обеспечения эффективности работы пайплайнов.

11 | Стартовые наборы: с нуля до CD



В этой главе

- ✓ Определение основных элементов эффективного пайплайна CD
- ✓ Как найти и добавить недостающие элементы в существующих пайплайнах CD
- ✓ Как создать с нуля эффективный пайплайн CD для нового проекта
- ✓ Как добавить полезную автоматизацию CD к уже существующим проектам и при этом не решать все проблемы сразу

Разобраться, с чего начать работу с непрерывной доставкой, может быть непросто, особенно если у вас вообще отсутствует автоматизация, если вы делаете что-то совершенно новое или если у вас накопилась куча унаследованного кода. Эта глава поможет вам избежать трудностей и покажет, с чего начать и как быстро получить максимальную отдачу от пайплайнов CD независимо от того, приступаете ли вы к работе с чистого листа или вам приходится иметь дело с унаследованным кодом, которому уже 20 лет.

Стартовые наборы: обзор

В этой главе мы рассмотрим, как усовершенствовать CD для проекта, независимо от того, на каком этапе вы начинаете. Глава состоит из трех разделов:

Словарик

Проекты *greenfield*, или *проекты, создаваемые с нуля*, являются совершенно новыми и поэтому предполагают большую свободу действий. Их противоположность — *старые* (или *унаследованные)* проекты (иногда называемые проектами *brownfield*), которые существуют достаточно долго, поэтому в них накапливается много кода и вносить в них крупные изменения становится проблематично.

1. *Краткий обзор.* Сначала мы вспомним задачи, которые вы найдете в любом пайплайне CD и которые мы уже подробно рассматривали. Мы обсудим, как они могут сочетаться друг с другом в прототипах пайплайнов выпуска и непрерывной интеграции. Полезно знать, что представляет собой пайплайн CD, но совсем другое дело — знать, как применять его в проектах. Поэтому остальная часть главы будет посвящена тому, как усовершенствовать пайплайны CD в двух очень разных, но очень распространенных случаях.
2. *Проект с нуля.* Первый тип проектов, которые мы рассмотрим, — это совершенно новые проекты (или проекты greenfield), в которых кода еще практически нет и можно с самого начала применить передовые методики. Для примера мы возьмем стартап под названием Gulpy, который работает всего пару месяцев, поэтому кое-какой код уже написан, но не так много, чтобы его было сложно изменять. Мы подробно разберем весь путь от полного отсутствия автоматизации CD до полноценного набора задач и пайплайнов.
3. *Старый (унаследованный) проект.* Каждый проект, создаваемый с нуля, рано или поздно становится старым, поэтому велика вероятность, что проект, над которым вы сейчас работаете, именно таков. В этих проектах, скорее всего, кода и автоматизации так много, что их модернизация может оказаться непосильной. Мы рассмотрим компанию Rebellious Hamster и ее унаследованный код и увидим, как инкрементное улучшение CD приносит быстрый результат.

Универсальные задачи пайплайна CD

Независимо от того, над каким программным проектом вы работаете, в полноценный пайплайн CD входят, как правило, одни и те же основные задачи. (Однако их количество и порядок выполнения могут быть разными.) В главах 1 и 2 мы бегло ознакомились с этим стандартным набором задач:

- *Линтинг* — наиболее распространенная форма статического анализа в пайплайнах CD.
- Юнит-тесты и интеграционные тесты — это разновидности *тестов*.
- Чтобы использовать большую часть программных продуктов, необходимо произвести их *сборку*.
- Многие продукты перед их использованием требуют *публикации*.
- Чтобы обновить работающий сервис для использования нового артефакта, нужно произвести *развертывание*.

На протяжении всей книги мы сталкивались с линтингом, тестированием, сборкой и развертыванием. В главе 6 вы познакомились с типами задач тестирования, которые должен содержать пайплайн:

- юнит-тесты;
- интеграционные тесты;
- сквозные тесты.

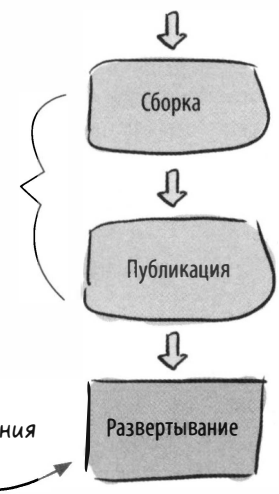
В главе 2 эти задачи были отнесены к категориям *шлюзов* и *преобразований* кода. Вот какие шлюзы и преобразования составляют полнофункциональный пайплайн CD:

Задачи шлюзования составляют часть пайплайна CI. Даже в них, вероятно, присутствуют преобразования, проверяющие, что элементы, проходящие через шлюз, соответствуют тому, что вы действительно выпускаете (см. главу 7).

Прототип пайплайна выпуска

В главах 9 и 10 рассматривались автоматизации сборки и развертывания, а также их инициализация. Предположим, что вы собираетесь запускать пайплайны сборки и развертывания отдельно от пайплайна CI (но это необязательно и даже нежелательно, но вы узнаете об этом подробнее в главе 13, посвященной реализации пайплайнов). Тогда ваш пайплайн выпуска будет содержать только задачи *преобразования* и выглядеть примерно так:

Событие, запускающее пайплайн, будет зависеть от выбранной стратегии развертывания (см. главу 10) или стратегии выпуска релиза (если развертывание не требуется, см. главу 9). Если вы осуществляете непрерывное развертывание или непрерывный релиз, то этот пайплайн должен запускаться автоматически после завершения пайплайна CI сразу после слияния (см. далее). В остальных случаях его можно запускать по запланированному событию или автоматически, например пометив коммит новой версией релиза.



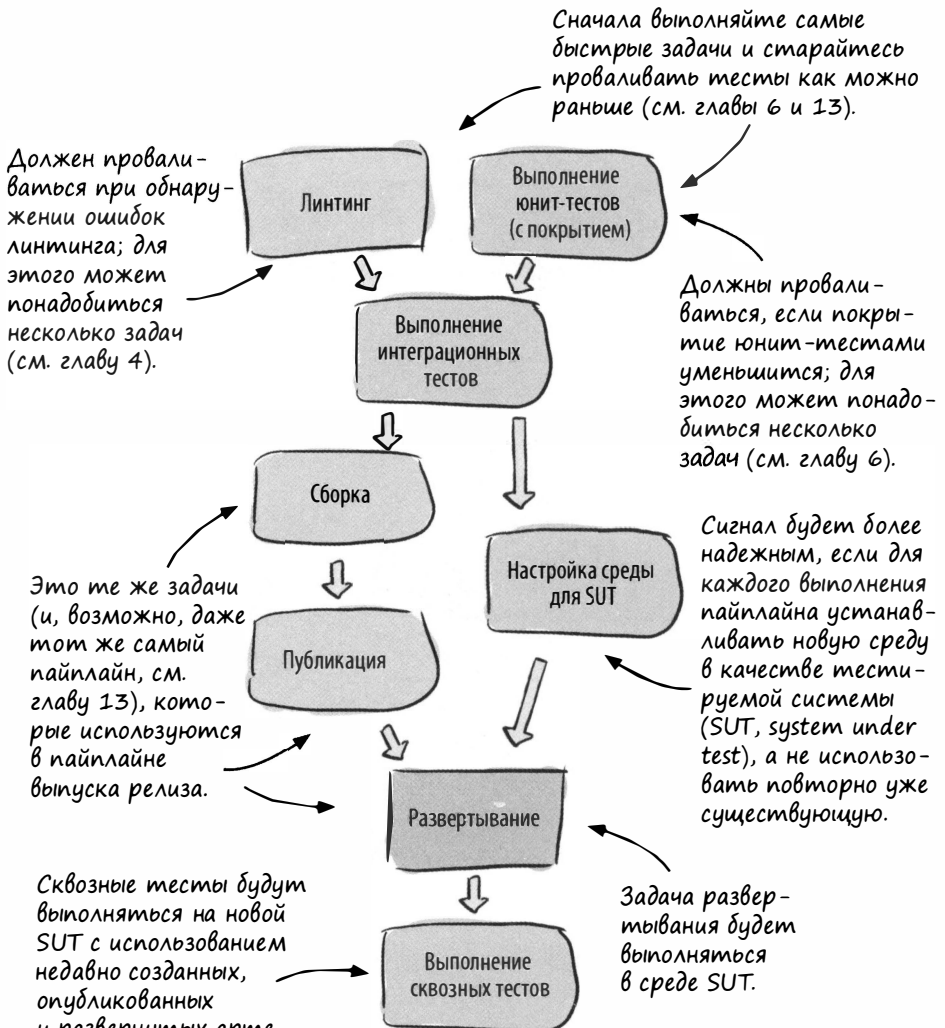
В зависимости от того, что вы создаете и какие инструменты используете, задачи сборки и публикации могут быть объединены в одну.

Эта задача развертывания может делать и мало, и много. В главе 10 объяснялось, как сделать развертывание легким и безопасным; вы можете встроить логику развертывания непосредственно в задачу самостоятельно, но более вероятно, что эта задача будет содержать обращение к какой-либо системе развертывания, в которой эта сложная логика уже реализована.

В развертывании нуждаются только продукты, которые вы предоставляете пользователям путем размещения и запуска. Пайплайны для других продуктов, например для библиотек, не будут включать задачи развертывания, но вам, вероятно, все равно понадобится пайплайн для сборки и публикации продуктов.

Прототип пайплайна CI

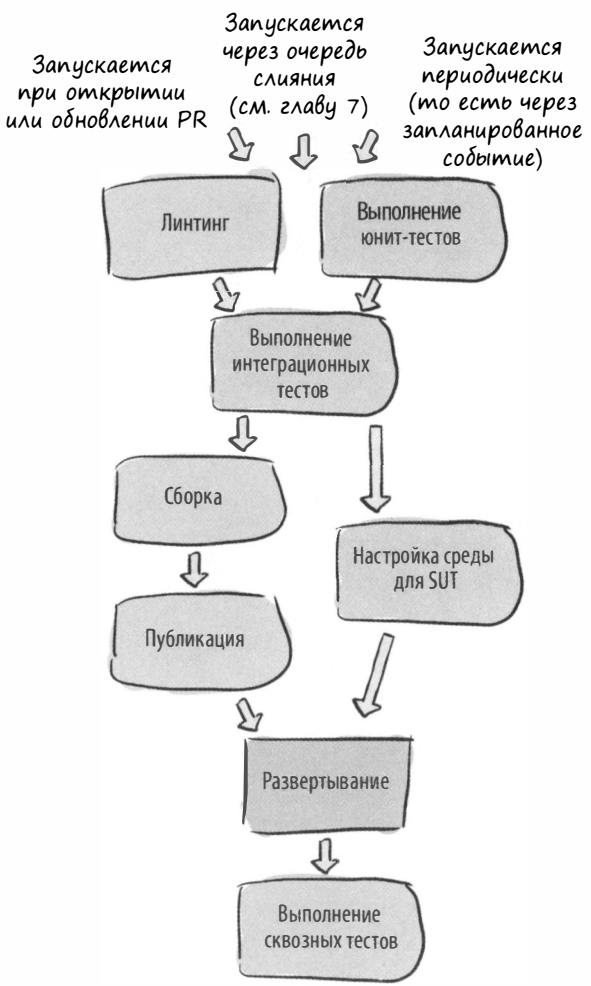
Пайплайн CI должен содержать задачи *шлюзования*. Но как было показано в главе 7, очень важно также включить в него задачи (или в идеале даже пайплайны) сборки и развертывания, чтобы убедиться, что тестируемый объект максимально приближен к версии для продакшен. Поэтому этот пайплайн, скорее всего, будет содержать задачи *преобразования* и будет выглядеть примерно так:



В главе 7 приведен пример работы этого прототипа пайплайна непрерывной интеграции для сайта CoinExCompare.

Пайплайны и их инициализация

Разобраться с тем, что включить в пайплайн CI, — только полбеды; его еще нужно вовремя запустить. В главе 7 мы рассматривали, в каких точках жизненного цикла изменений необходимо запускать части пайплайнов CI, чтобы предотвращать появление багов. Если к схеме пайплайна CI добавить информацию об инициализации, то полная картина пайплайнов CD будет выглядеть примерно так:



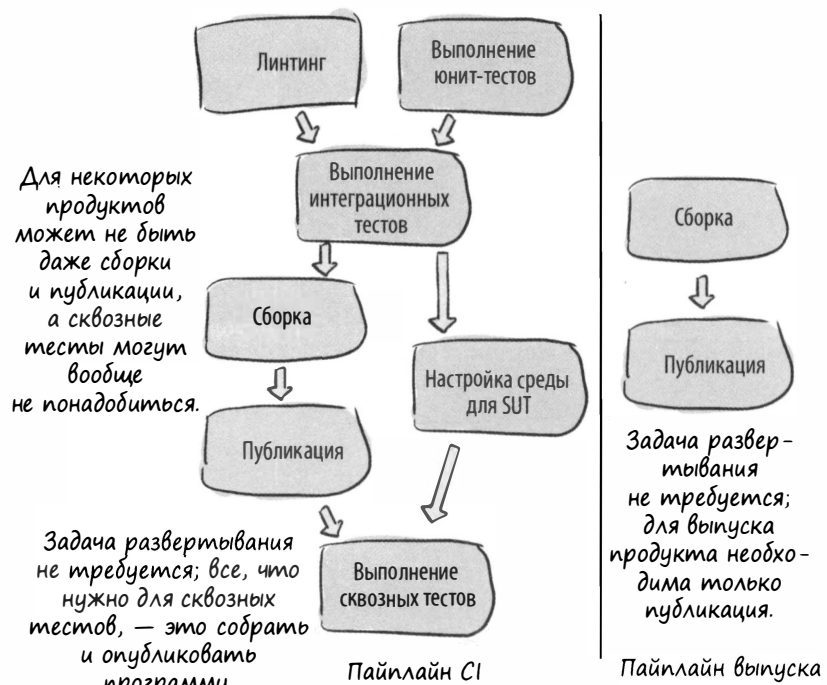
Это то, к чему вы должны стремиться, но способ, которым вы к этому придете (и вообще, нужны ли вам все этапы? Вспомните из главы 1, что CD — это всего лишь методика), будет зависеть от проекта, над которым вы работаете.

Что делать, если у меня не сервис?

Примеры пайплайнов в этой главе приведены для ПО, которое предоставляется в виде сервиса и поэтому должно быть где-то развернуто. В главе 1 мы изучили, какие программные продукты можно поставлять (библиотеки, двоичные файлы, конфигурации, образы и сервисы).

Самое большое отличие между пайплайнами, которые запускаются для ПО, предоставляемого как сервис, и ПО, которое таковым не является (например, для библиотек и двоичных файлов), заключается в задаче развертывания. Большинство задач CI универсальны: линтинг и различные виды тестирования можно применять ко всем программным продуктам, даже к конфигурации.

Если вы поставляете инструменты и библиотеки, вы можете не включать задачу развертывания ни в пайплайн CI, ни в пайплайн выпуска:



Если ваш продукт предназначен для использования как сервис (но вы не предоставляете под него хостинг), пайплайн CI все равно должен включать задачу развертывания (поскольку развертывание необходимо для полного тестирования), а пайплайн выпуска — нет. Что представляет собой доставка для каждого вида продуктов, то есть какие элементы нужно включать в пайплайн, а какие нет, смотрите в таблице главы 1 (с. 41).

Проект с нуля: переходим к CD

Освежив в памяти основные концепции пайплайнов CD, рассмотрим, как они работают в проектах, находящихся на различных этапах жизненного цикла: новых и унаследованных проектах. Для разработки пайплайнов CD и автоматизации процессов работа с нуля — идеальный сценарий.

Когда код небольшой, легко принимать масштабные решения, которые будут применяться везде (например, вводить новые правила линтинга). И чем раньше вы наладите автоматизацию соблюдения политик (например, минимальное покрытие юнит-тестами — 80 %), тем больше шансов, что эти требования будут соблюдаться в проекте по мере его роста.

В проектах с нуля можно вносить совсем небольшие коррективы в процесс работы, которые принесут огромную пользу в будущем, когда проект будет считаться унаследованным, — что неизбежно.

Мы рассмотрим принцип, который одинаково применим ко всем проектам, но сначала разберемся, что он означает, если начинать с чистого листа:

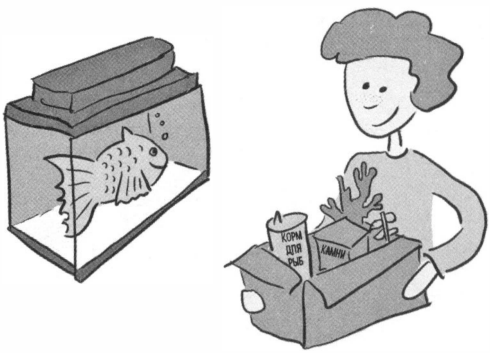
Получать как можно больше сигналов как можно быстрее.

Когда мы начинаем работу без автоматизации, нам нужно выяснить, какие элементы пайплайна CD дадут самый полезный сигнал быстрее всего, и отталкиваться от них.

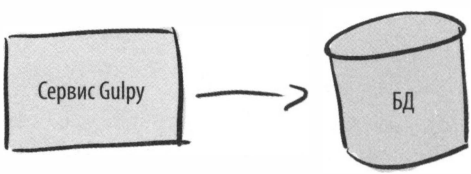
В главе 5 мы уже говорили о *сигнале* и *шуме* в связи с тестами, а в главе 7 подробно обсудили, в каких местах пайплайна требуются сигналы о появлении багов. По мере улучшения (или создания с нуля) пайплайны CD приносят максимальную отдачу, если стараться увеличить количество получаемых от них сигналов, то есть информации, которая нам нужна, чтобы поддерживать код в состоянии, пригодном для релиза, и осуществлять релизы по мере необходимости.

Проект Gulpy

Я начну с рассмотрения стартап-проекта greenfield под названием Gulpy. Проект предназначен для аквариумистов и призван упростить онлайн-заказ корма и разных принадлежностей.



Проекту Gulpy всего несколько месяцев, поэтому в нем совсем немного кода и разработчики еще не создали пайплайны CD или автоматизацию. Его архитектура довольно проста, всего один сервис и одна база данных:



На текущий момент создан только код главной страницы, он позволяет пользователю завести аккаунт и ничего больше (пока!) и находится только в одном репозитории.

Стоит ли команде Gulpy сразу создавать более сложную архитектуру?

В главе 3 Сара и Саша разработали более сложную архитектуру для своего стартапа, прежде чем начать писать какой-либо код. Подход Сары и Саши, как и подход Gulpy (который ориентирован на то, чтобы что-то заработало быстро), верен; их подход, вероятно, несколько более перспективен. Но главное, оценить сроки и цели проекта и принять решение, подходящее именно вам.

Проект greenfield: с нуля до CD

Определяя, какие средства автоматизации CD следует добавить в проект, неважно, новый он или унаследованный, руководствуйтесь одной целью:

> **Получать как можно больше сигналов как можно быстрее.**

Концепция соотношения *сигнал — шум*, в частности максимизация сигнала и минимизация шума, применима к CI в целом:

> **Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.**

Часть CI, относящаяся к проверке, — это *сигнал*, который вы ищете: сигнал о том, что код пригоден для релиза и что любые изменения не повлияют на это состояние. Поставленная цель предполагает, что при выборе CD для проекта сначала лучше сосредоточиться на улучшении CI. Это логично, поскольку *прежде чем выпускать релизы быстрее и чаще, необходимо убедиться, что это безопасно*!

Для greenfield-проекта можно реализовать автоматизацию пайплайна CD путем постепенного добавления задач CD в следующем порядке:

1. Создайте начальную автоматизацию и добавьте задачу, чтобы убедиться, что код пригоден к *сборке*.
2. Обеспечьте высокое качество кода, добавив *линтинг* как можно раньше.
3. Исправьте все ошибки линтинга, чтобы код стал идеально чистым.
4. Начните проверять функциональность (и писать более чистый код) с помощью *юнит-тестов*.
5. Измерьте *покрытие тестов*, чтобы сразу же понять, что отсутствует в юнит-тестах.
6. Добавьте тесты, которых не хватает для достижения целей покрытия.
7. После настройки начальной CI приступайте к созданию пайплайна выпуска релизов, добавив логику для *публикации* создаваемого кода.
8. Добавьте в пайплайн выпуска релизов *автоматизацию развертывания*.
9. Когда большинство базовых элементов готово, можно добавлять *интеграционные* и *сквозные тесты*.

Вы можете полностью изменить порядок шагов. Просто помните, что главное — получать как можно больше сигналов как можно быстрее. Именно поэтому этапы, требующие больше времени (например, настройка сквозного тестирования), оставлены на потом; если вы займетесь ими сразу, то сигнал вы получите не скоро.

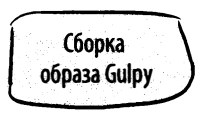
Первый шаг: выполняется ли сборка?

Первым делом необходимо проверить, что код можно собрать в том виде, в каком он должен использоваться. Если код нельзя собрать, скорее всего, с ним не получится проделать и другие операции: тестирование, линтинг и, конечно же, релиз.

Сигнал пайплайна CI о возможности сборки сразу же позволит выполнять дальнейшие действия по реализации пайплайна CD, вот почему именно с этого стоит начать работу с совершенно новым кодом:

1. Создайте начальную автоматизацию и добавьте задачу, проверяющую, что код пригоден к *сборке*.

Код проекта Gulpy написан на Python, и разработчики запускают свой сервис в контейнере Docker, поэтому самая первая задача, которую они будут использовать (и начало их пайплайна CI), — это задача, выполняющая сборку образа контейнера.



Однозадачный пайплайн Gulpy пока не представляет особого интереса, но он будет развиваться!

Начавшись так скромно, проект Gulpy прошел путь от полного отсутствия автоматизации CD до первого пайплайна CD!

Автоматизация сборки — это уже достижение!

Как уже говорилось в главе 1, самые ранние системы CD выполняли одну простую задачу: создание программ. Остальная автоматизация (тестирование, линтинг, развертывание и т. д.) появилась позже. Именно поэтому системы CD часто называют *системами сборки*, а задачи в пайплайнах CD по-прежнему принято называть *сборками* (или *билдами*). Так что не стоит недооценивать наличие автоматизации, которая будет производить сборку за вас, ведь раньше это было огромным достижением!

Выбор системы CD

Перечень функций распространенных систем CD см. в приложении А.

Подобно тому как начинать проект с исходной точки (то есть с минимально жизнеспособного продукта, или MVP), которая определит все дальнейшие действия, — это отличный способ заложить основу для добавления будущих функций, так и установление простой начальной цели для пайплайна CD, например осуществления сборки, станет основой для вашего продукта и одновременно позволит начать работу над автоматизацией.

В проекте Gulpy уже существует исходный (однозадачный) пайплайн, но он не принесет компании никакой пользы, если не будет запущен и не начнет работать. Однако прежде чем что-либо настраивать, разработчики должны выбрать систему CD. Чтобы сузить круг поиска, необходимо ответить на два важных вопроса. Вот первый из них:

- Вы собираетесь использовать существующую систему CD или создавать собственную?

Скорее всего, вы захотите использовать существующую систему, чтобы сэкономить на стоимости (и сложности) создания и поддержки собственной. Если же у вас есть особые требования к способу сборки, тестирования и развертывания кода, которые не поддерживаются существующими системами CD, то скорее всего, вам не избежать создания собственной системы. (К счастью, это маловероятно на столь раннем этапе проекта, если только вы не работаете в сфере, где существуют особые нормативные требования, о которых вы уже знаете.) Исходя из того, что вы выбрали вариант с существующей системой, возникает второй вопрос:

- Если вы используете существующую систему, вы собираетесь размещать ее на стороннем ресурсе или на своем?

Лучше всего (особенно на начальном этапе) выбрать первый вариант. Это позволит легко начать работу и не потребует оплаты услуг специалистов по настройке и обслуживанию системы.

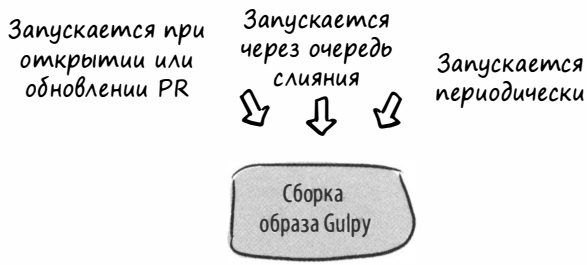
Для многих проектов исходный код будет закрытым. Чтобы использовать существующую систему CD для работы с закрытым кодом, необходимо либо настроить безопасный доступ к системе, либо разместить на хостинге свой собственный экземпляр системы.

Собственную систему CD не обязательно создавать с нуля. В приложениях вы найдете системы CD, существующие в виде платформ и строительных блоков, которые можно использовать для создания собственных систем. По мере масштабирования проекта может оказаться удобнее поддерживать собственную систему CD, чтобы легко накладывать ограничения на проекты и создавать именно ту автоматизацию, которая подходит для вашего бизнеса и клиентов.

Настройка начальной автоматизации

Команда Gulpy настраивает начальный пайплайн CD, который создает образ контейнера Gulpy и запускается тремя событиями (подробнее об этих событиях и их важности читайте в главе 7):

- при открытии или обновлении PR;
- когда PR готов к слиянию и контролируется очередью слияния;
- периодически, каждый час.



Разработчики Gulpy хотят сохранить исходный код закрытым, но им не требуется создавать свою систему CD. Они уже используют приватные (частные) репозитории на GitHub, поэтому они решили воспользоваться GitHub Actions для автоматизации.

Чтобы создать свой (однозадачный) пайплайн в GitHub Actions и настроить триггеры, им нужно лишь создать файл рабочего процесса (ci.yaml) и закоммитить его в своем репозитории в каталоге .github/workflows:

```
name: Gulpy Continuous Integration
on:
  pull_request:
 push:
    branches:
   -gh-readonly-queue/main/**
  schedule:
 -cron: '0 * * * *'
```

Указывает GitHub Actions срабатывать при обновлении PR.

Указывает GitHub Actions, что очередь слияния должна запускать этот рабочий процесс перед слиянием.

Указывает GitHub Actions срабатывать каждый час.

```
jobs:
  build:
  ...
```

Рабочий процесс (запущенный тремя предыдущими событиями) имеет одно задание под названием «сборка». Подробности здесь опущены, но он может быть описан здесь или содержать ссылку на задачу GitHub, определенную в другом месте.

Пайплайн CD запущен и работает! Еще многое предстоит добавить, но теперь команда сразу же получит сигнал, если в репозиторий будет внесено изменение, делающее код непригодным к релизу (посредством триггеров PR и очереди слияния), или (с помощью периодических тестов) если в процесс сборки закралась неопределенность, в результате чего сборка завершилась сбоем (см. главу 9).

Состояние кода: линтинг

Настроив автоматизацию и проверив, что код можно собрать, можно переходить к самому коду. На такой ранней стадии проекта бóльшая часть кода еще не написана, поэтому сейчас самое время привести его в порядок и обеспечить условия для поддержания высоких стандартов на всем жизненном цикле проекта.

В главе 4 вы убедились, насколько полезным может быть линтинг: он не только сохраняет код чистым и последовательным, но и выявляет серьезные баги. Вы также заметили, что проводить линтинг унаследованного кода может быть сложно и что на данном этапе достичь идеала вряд ли получится, а идеал заключается в следующем:

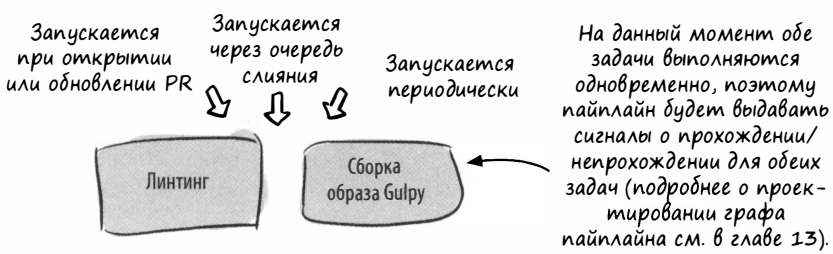
При анализе кода линтер не выдает сообщения об ошибках.

Однако при работе с новым кодом это вполне реально. И если вы с самого начала настроите автоматизацию так, чтобы она поддерживала это состояние, вы легко его сохраните.

Чем позже вы добавляете линтинг, тем сложнее обеспечить его регулярное применение, поэтому для нового проекта это отличная возможность повысить качество кода. Вот почему при настройке пайплайнов CD для проектов с нуля второй шаг должен быть таким:

2. Обеспечьте высокое качество кода, добавив *линтинг* как можно раньше.

Команда Gulpy добавляет автоматизацию линтинга в свой пайплайн:



Как только разработчики начинают выполнять задачу линтинга, она выявляет в их коде ряд нарушений. Они решают исправить их немедленно, поэтому для начала настраивают пайплайн так, чтобы задача линтинга не запускалась, пока они устраняют проблемы, делая третий инкрементный шаг для создания пайплайна CD:

3. Исправьте все ошибки линтинга, чтобы код стал идеально чистым.

```
jobs:
  lint:
    continue-on-error: true
  build:
  ...
```

Предотвращает сбой всего рабочего процесса при невыполнении этой задачи. Приведя код в порядок, разработчики смогут удалить эту опцию, чтобы будущие изменения блокировались до тех пор, пока они не станут соответствовать требованиям линтинга.

Состояние кода: юнит-тесты

Теперь у проекта Gulpy есть пайплайн, который проверяет возможность осуществления сборки кода, а также требования линтинга для поддержания согласованности кода (и выявления типичных багов). Далее следует обеспечить проверку самой функциональности, то есть бизнес-логики (и всего кода, поддерживающего ее), которая на самом деле является ядром любого программного проекта. Функциональность программного продукта проверяется с помощью тестов, а самыми быстрыми с точки зрения создания (и выполнения) являются юнит-тесты.

На ранней стадии проекта уже могут быть какие-то юнит-тесты, но даже если их нет, полезно на будущее добавить автоматизацию их выполнения (что мы и сделаем далее). Автоматизация позволит получать обратную связь сразу же после добавления тестов:

Словарик

Бизнес-логика — это главная причина, по которой мы вообще пишем программы! Именно правила, которые мы воплощаем в коде, делают наши библиотеки и сервисы действительно значимыми. Продукт остается востребованным у пользователей, поскольку для них важна заключенная в нем логика, и если мы создаем его ради получения прибыли, именно эта логика обеспечивает его *коммерческую ценность*.

4. Начните проверять функциональность (и писать более чистый код) с помощью *юнит-тестов*.

Чтобы писать эффективные юнит-тесты, код должен быть структурирован так, чтобы его можно было тестировать, а это обычно означает сильную связность (high cohesion), слабую связанность (loose coupling) и прочие полезные свойства. Без юнит-тестов проще не обращать внимания на структуру кода, и добавлять юнит-тесты становится все труднее и труднее.

В проекте Gulpy еще нет юнит-тестов, поэтому задача их запуска, которую добавляют разработчики, сразу же проходит (тесты не найдены = тесты не провалены). Теперь их пайплайн CD выглядит следующим образом:

Интеграционные и сквозные тесты также чрезвычайно полезны, но мы не будем добавлять их сразу, потому что их разработка и выполнение требуют больше времени. Не говоря уже о том, что чем раньше вы внедрите юнит-тесты, тем легче будет обеспечить высокое покрытие и поддерживать его. Помните: большинство тестов должны быть модульными!

Состояние кода: покрытие

После автоматизации запуска юнит-тестов (она будет активироваться при их добавлении) целесообразно добавить измерение покрытия кода тестами, которое на данный момент составляет 0 %! Отсутствие тестов означает отсутствие покрытия. Поскольку проект новый и кода для покрытия не так много, самое время сосредоточиться на достижении желаемого уровня покрытия. Тогда в дальнейшем вам останется только поддерживать этот уровень! Поэтому следующий шаг для Gulpy будет таким:

> Установка любого произвольного порога означает, что время от времени вы будете писать бесполезные тесты, например, если вам требуется покрытие 80 %, а у вас только 79,5 %. Но простота необходимой автоматизации стоит пары лишних тестов.

5. Измерьте *покрытие тестов*, чтобы сразу же понять, что отсутствует в юнит-тестах.

В главе 6, когда Шридхар добавил измерение покрытия для сайта Dog Picture Website, ему пришлось создать логику для отслеживания уровня покрытия, чтобы этот уровень не снижался. Это эффективный способ постепенно наращивать покрытие в старом проекте, но в совершенно новых проектах, в которых практически нет кода, можно определить произвольный порог и сразу же написать тесты, необходимые для достижения этого порога. Разработчики обновляют задачу юнит-тестирования, чтобы в ней также измерялось покрытие и выдавалась ошибка, если оно меньше 80 %:

Задача, конечно же, сразу проваливается и блокирует внесение любых новых изменений. Затем разработчики Gulpy добавляют юнит-тесты, пока не достигнут своей цели покрытия в 80 %:

6. Добавьте тесты, которых не хватает для достижения целей покрытия.

Они временно снимают требование покрытия 80 %, пока добавляют тесты. В противном случае им пришлось бы создать один большой PR со всеми тестами и включить его обратно в главную ветку, как только все тесты будут готовы.

> ***Что считать достаточным порогом покрытия?***
>
> Достаточный порог покрытия зависит от свойств кода. Начните с покрытия 80 % или около того и посмотрите, как пойдет. Это достаточно мягкое условие, допускающее, что какие-то строки кода останутся непокрытыми, если их тесты не слишком важны, и достаточно жесткое, чтобы обеспечить покрытие большинства строк. В дальнейшем откорректируйте это значение в большую или меньшую сторону.

Что дальше: публикация

Теперь у проекта Gulpy есть пайплайн с базовыми элементами CI, которые гарантируют, что код работает (то есть пригоден для выпуска) и что компания соблюдает поставленные цели для линтинга и покрытия юнит-тестами. Этого достаточно, чтобы разработчики были уверены в достижении исходной цели непрерывной интеграции:

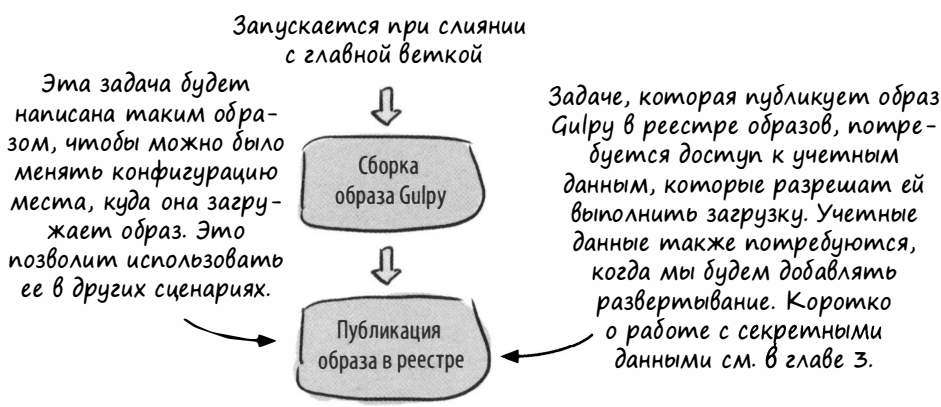
> **Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.**

Чувствуя себя достаточно уверенными (пока!) в своей CI, они могут перейти к остальной части пайплайна CD, начав работать с образом контейнера, который они уже создали:

7. После настройки начальной CI приступайте к созданию пайплайна выпуска релизов, добавив логику для *публикации* создаваемого кода.

Поскольку разработчики не ограничены в выборе методики развертывания, они решают сразу же воспользоваться непрерывным развертыванием. Они создают отдельный пайплайн, который использует ту же задачу сборки, что и пайплайн CI, и запускается при успешном слиянии с главной веткой (в главе 13 вы узнаете больше о компромиссах, связанных с использованием нескольких пайплайнов):

В некоторых инструментах для создания артефактов сборка и выгрузка объединены в одну команду.



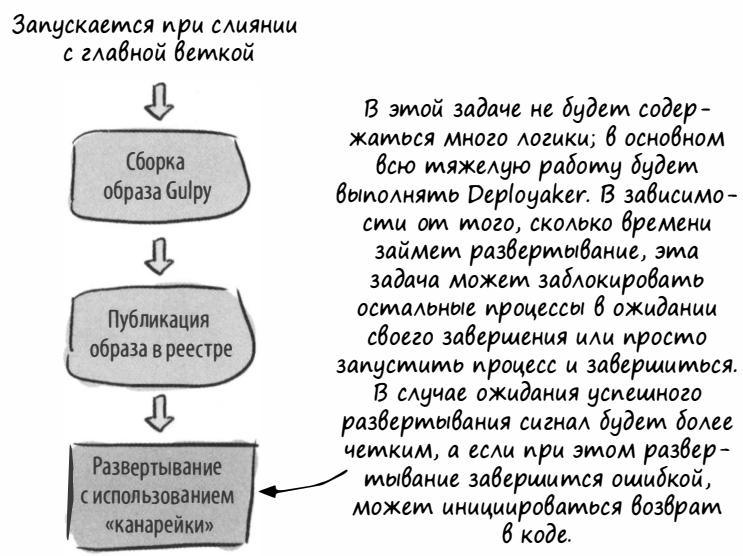
Развертывание

Следующим шагом для команды Gulpy является автоматизация развертывания. Разработчики уже провели несколько развертываний, но они делали их вручную, запуская скрипт, который обновляет экземпляры их сервиса, работающие в популярном облаке RandomCloud. Итак, следующий шаг:

8. Дополните пайплайн выпуска релизов *автоматизацией развертывания*.

Наличие уже готового скрипта для выполнения части процесса развертывания может упростить его автоматизацию. Однако на таком раннем этапе жизненного цикла продукта проще проявлять гибкость в выборе решений для развертывания, и то, что вы начинаете работу с чистого листа, может не иметь большого значения, особенно если вы решите использовать существующий инструмент, который выполнит автоматизацию развертывания за вас (см. главу 10 о видах стратегий развертывания, которые вам, возможно, стоит рассмотреть).

Разработчики Gulpy решили реализовывать канареечные развертывания. Они выбрали Deployaker, популярный инструмент для автоматизации стратегий развертывания. Кроме того, они планируют использовать непрерывное развертывание, поэтому обновляют существующий пайплайн выпуска, добавив задачу, которая вызывает Deployaker для запуска канареечного развертывания:



На этой ранней стадии проекта также уместно внедрить и опробовать непрерывное развертывание. С одной стороны, это создаст отличный задел для дальнейшего развития проекта, не подвергая его особому риску, поскольку этап достаточно ранний, а с другой — даст возможность отложить принятие решения о том, подходит ли такое развертывание для проекта, на более поздний срок.

Расширяем тестирование

Теперь у проекта Gulpy есть все необходимое: пайплайн CI, который проверяет сборку, линтинг и юнит-тесты (с 80 %-ным покрытием), и пайплайн выпуска, который использует непрерывные канареечные развертывания. Последний этап разработки этих пайплайнов — составление сценария тестирования. Вспомните пирамиду тестов из главы 6:

Большинство тестов должны составлять юнит-тесты, но интеграционные и сквозные тесты тоже полезны! Сейчас в проекте Gulpy есть только юнит-тесты.

Какими бы полезными ни были юнит-тесты, они все равно могут многое упустить и их нужно дополнять сквозными и/или интеграционными тестами, которые тестируют отдельные блоки совместно:

9. Когда большинство базовых элементов готово, можно добавлять *интеграционные* и *сквозные тесты*.

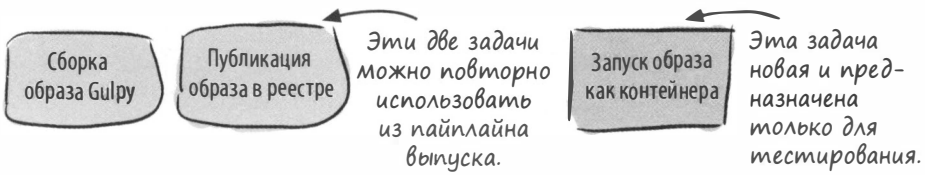
Подготовку и запуск этих тестов лучше сделать последним этапом настройки исходных пайплайнов CD по следующим причинам:

- Подготовка и выполнение этих тестов — и настройка, необходимая для запуска сквозных тестов, — требуют гораздо больше времени.
- Для запуска этих тестов можно повторно использовать часть или всю логику сборки, публикации и развертывания.

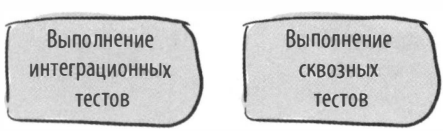
Задачи для интеграционных и сквозных тестов

При разработке сквозных тестов инженеры Gulpy анализируют свой пайплайн выпуска и определяют, что можно применить повторно. Они должны использовать задачи сборки и публикации в неизменном виде, передавая им различные параметры имени создаваемого образа, чтобы указать, что он предназначен только для тестирования (например, `gulpy-v0.2.3-testing`).

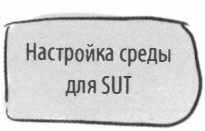
Для задачи развертывания им предстоит решить, использовать ли Deployaker для развертывания тестов. У этого инструмента есть одно преимущество — он предоставляет возможность протестировать конфигурацию развертывания. Но — и это большой недостаток — для использования Deployaker при тестировании требуется отдельный экземпляр, и разработчики не смогут просто использовать такую же логику при разработке. Поэтому они решают написать новую задачу для развертывания, которая напрямую запускает сервис, работающий как контейнер.



Разработчики создают набор сквозных тестов, которые выполняются на работающем экземпляре сайта Gulpy, как это сделал бы клиент, и подготавливают несколько интеграционных тестов.



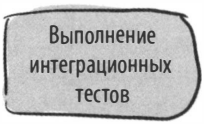
Чтобы запустить сквозные тесты, им необходимо еще одно: место для развертывания и тестирования, то есть среда для тестируемой системы (SUT, system under test). Команда Gulpy создает задачу, которая будет выполнять инициализацию виртуальной машины для запуска контейнера:



Возможно, вам вообще не потребуется настраивать отдельную тестируемую систему (например, вы можете запустить контейнер непосредственно в рамках сквозного тестирования). На самом деле это зависит от дизайна системы и количества компонентов, которые необходимо задействовать при тестировании. Подробнее об этом можно узнать из литературы, посвященной тестированию.

Завершаем пайплайн CI

Для запуска интеграционных тестов разработчикам Gulpy нужно добавить в существующий пайплайн CI всего одну задачу, которая запускает эти тесты. Эти тесты выполняются немного дольше, чем юнит-тесты, но не требуют специальной настройки:



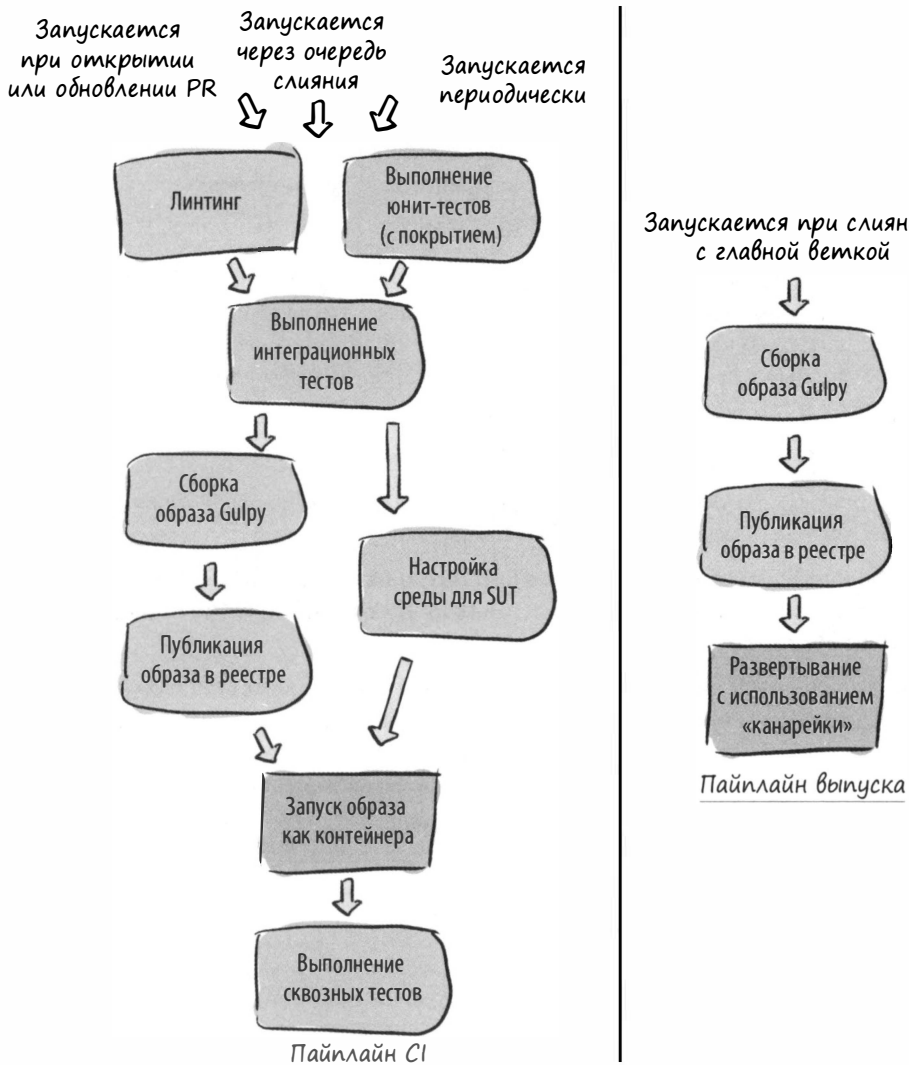
Добавить сквозные тесты в пайплайн CI немного сложнее. Для этого нужен целый пайплайн задач:


```
jobs:
  setup-sut:
    ...
  build:
    ...
  publish:
     needs: build
    ...
  run-container:
     needs: [setup-sut, publish]
    ...
  end-to-end-tests:
     needs: run-container
    ...
```

GitHub Actions использует ключевое слово needs для определения зависимостей и упорядочивания заданий в рамках рабочего процесса. Подробнее об определении сложных графов см. в главе 13.

Окончательные пайплайны проекта Gulpy

Собрав все части воедино, команда Gulpy сформировала следующие два пайплайна, каждый из которых запускается отдельно:



Возможно, по мере развития проекта и добавления новых функций будут вноситься коррективы, но в целом эти пайплайны вполне должны удовлетворить потребность команды в CD. Это идеальные образцы, к которым необходимо стремиться при разработке пайплайнов (плюс-минус несколько задач в зависимости от целей проекта и используемых инструментов).

Унаследованный проект: переход к CD

На самом деле проекты с нуля запускаются нечасто, поэтому вам вряд ли выпадет удача реализовать все необходимые элементы, пока кода еще немного. Вместо этого вы, скорее всего, будете иметь дело со *старым (унаследованным)* проектом.

Каждый greenfield-проект в какой-то момент становится старым. Когда именно наступает этот момент, точно сказать нельзя, но одним из признаков устаревания проекта является увеличение объема работы для добавления блокирующих задач CD, таких как те, которые мы только что рассматривали: например, требование 80 %-ного покрытия кода или требование прохождения линтинга. Как вы уже поняли из предыдущих глав, для унаследованных проектов реализовать подобные рекомендации сложнее.

Подход, который мы обсудили, все равно полезен, поскольку он определяет элементы, необходимые во всех проектах, в том числе унаследованных. Однако порядок добавления этих элементов немного отличается, и скорее всего, вы не сможете осуществить все свои задумки (и это нормально!).

Независимо от того, работаете ли вы со старыми или с новыми проектами, цель CD в них одинакова:

> **Получать как можно больше сигналов как можно быстрее.**

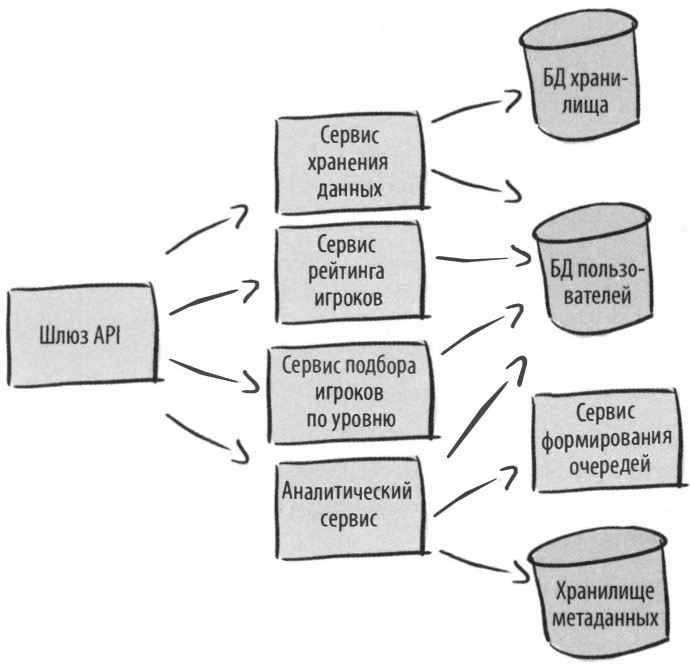
И опять же, начать стоит с нерперывной интеграции, поскольку она закладывает основу для реализации остальных компонентов:

> **Непрерывная интеграция — это процесс, при котором объединение изменений кода происходит постоянно и каждое изменение проверяется при внесении.**

Если до сих пор CI пренебрегали, она может быть особенно актуальной для унаследованного кода — кто знает, в каком он состоянии! Без CI очень трудно понять, что произойдет, если начать добавлять автоматизацию релизов и выпускать их чаще и быстрее. Имеет смысл перед этим убедиться в жизнеспособности кода.

Проект Rebellious Hamster

Теперь рассмотрим, как модернизировать CD для унаследованного проекта, на примере проекта Rebellious Hamster. Эта компания предоставляет серверные услуги для видеоигр. У нее огромная кодовая база, которая создавалась последние пять лет, но автоматизация CD отсутствует — или, по крайней мере, команда Rebellious Hamster не настраивала ее последовательно для каждого проекта. С чего же начать?



Код, поддерживающий эти сервисы, находится в нескольких репозиториях:

Каждый репозиторий в большей или меньшей степени принадлежит отдельной команде, поэтому в одних репозиториях CD не настроена вообще, а в других иногда присутствуют тестирование и автоматизация. Между проектами нет согласованности, и у каждой команды свои стандарты.

Первый шаг: установление приоритетов для инкрементных целей

Работа с унаследованными проектами может оказаться вам не по плечу, если вы установите слишком высокую планку. Лучше поставьте перед собой несколько инкрементных целей, которые можно начать воплощать сразу же и сразу же получать результат, чтобы, остановившись в любой момент, знать, что вы продвинулись по сравнению с начальной точкой. И помните, что всех целей достигать не обязательно! Две широкие области, требующие улучшения, соответствуют двум принципам CD:

- выпускать релизы ПО безопасно и в любое время;
- делать выпуск максимально просто, буквально одним нажатием кнопки.

В первую очередь обеспечьте возможность доставлять свой продукт безопасно в любое время, то есть вы должны всегда знать, что произошел сбой:

1. Автоматизируйте процесс, чтобы понимать, пригоден ли код к сборке.
2. Изолируйте части кода, которые вы хотите улучшить, от частей, на которые не стоит тратить силы, следуя парадигме «разделяй и властвуй» (на исправление всех элементов, скорее всего, уйдет слишком много сил, а отдача в результате будет невелика).
3. Добавьте тесты, включая измерение покрытия.

Как только вы почувствуете, что получаете достаточно сигналов о том, что можно безопасно выпускать релизы чаще или, по крайней мере, быстрее, сделайте следующее:

- Решите, будете ли вы автоматизировать существующие процессы или начнете с нуля в рамках нового подхода (например, используя сторонний инструмент).
- Если вы выбрали первое, автоматизируйте процессы постепенно, по частям.
- При переходе на сторонний инструмент разработайте безопасные эксперименты с минимальным воздействием для перехода от текущих процессов к новому инструменту.

К счастью, любая модернизация — это улучшение. Даже если вы никогда не продвинетесь дальше небольших инкрементных изменений, вы все равно улучшите свой механизм непрерывной доставки!

Куда же делся линтинг?

Как вы могли заметить, выше говорится о тестировании, но ничего нет о линтинге. При работе с новым кодом я рекомендовала сначала добавить линтинг в основном потому, что это очень легко сделать, когда кода мало, и это будет мотивировать на написание последовательного кода. Однако добавить линтинг в унаследованный код гораздо сложнее (см. главу 4), и быстрее пользу принесут тесты и автоматизация развертывания. Поэтому для унаследованного кода линтинг часто становится вишенкой на торте, которую можно добавить позже, если останется время.

В первую очередь «болевые точки»

Решая, с чего начинать поэтапное улучшение механизма CD, определите «болевые точки» процесса. Это может повлиять на порядок действий. (И помните, как уже говорилось в главе 6, если существует достаточно «болезненный» процесс, выполняйте его чаще!)

Определение «болевых точек» также помогает мотивировать команды к совместной работе над созданием согласованных пайплайнов CD. Акцент на уменьшении «головной боли» команд может сплотить их вокруг общей цели. Кроме того, решив в первую очередь самые крупные проблемы, вы уже останетесь в плюсе, даже если у вас закончится время и вы не успеете реализовать все намеченные цели CD.

Что вы имеете в виду под «болевыми точками»?

Сложно дать точное определение, но можно считать, что «*болевые точки*» в процессах CD — это то, чего стараются избегать, потому что это проблемы, которые трудно решать. Например, это затормозит разработку функций, потребует сверхурочной работы или это что-то, что регулярно откладывается. Зачастую лучший способ обнаружить «болевые точки» — найти действия, которые выполняются реже всего: например, развертывание только раз в три месяца может быть признаком того, что в нем есть «болевые точки». Близко связанное с этим понятие, описанное в книге разработчиков Google «Site Reliability Engineering. Надежность и безотказность как в Google»[1], — это *рутина*, то есть повторяющиеся ручные операции, не имеющие особой ценности, которые часто являются источником «головной боли» при реализации CD.

Что делать, если привлечь все команды не получается?

Вы наверняка хотите, чтобы все ваши команды занялись улучшением CD, но скорее всего, согласятся на это не все — по крайней мере, сразу. В этом случае стоит начать с одной команды или проекта и на их примере продемонстрировать, что вы предлагаете (и это все равно будет полезно, даже если вы не сможете убедить всех). Наглядное воплощение концепции и ее преимуществ может стать самым убедительным аргументом. Еще лучше, если вы сможете подкрепить его метриками: соберите метрики по «болевым точкам» и покажите, как с модернизацией CD они улучшаются (метрики DORA, рассмотренные в главах 8 и 10, отлично подойдут для начала).

Новые проекты — отличное начало

В основном вы, конечно, будете работать с унаследованными продуктами, но время от времени обязательно должны будут появляться новые проекты. Новые проекты — прекрасный повод продемонстрировать все преимущества лучших практик и установить стандарты, которые унаследованные проекты смогут постепенно перенимать. Относитесь к новому проекту как к «непаханому полю» и устанавливайте высокие стандарты (включая линтинг) с самого начала!

[1] Бетси Бейер и др. «Site Reliability Engineering. Надежность и безотказность как в Google». Санкт-Петербург, издательство «Питер».

«Болевые точки» в проекте Rebellious Hamster

«Болевые точки» в проекте Rebellious Hamster иллюстрируют подход, описанный выше. Их «болезненность» ярко выражена в отношении разработчиков к развертыванию:

- Развертывания происходят редко и нерегулярно (не чаще чем раз в 3 месяца).
- Все сервисы развертываются одновременно.
- Перед развертыванием проводится авральное тестирование в промежуточной среде, в ходе которого обнаруживается множество багов.
- Баги после развертывания оказываются нормой, и после развертывания все долго и лихорадочно их исправляют.

Поскольку много багов обнаруживается после развертывания, развертывания проводят редко. И хотя «болевая точка» связана с развертыванием, источник проблем кроется не в процессах развертывания, а в состоянии развертываемого кода:

- Перед развертыванием кода в репозитории команда практически не получает сигналов о том, безопасно ли оно.
- Перед развертыванием команда не получает сигналов, гарантирующих, что многочисленные сервисы Rebellious Hamster могут успешно интегрироваться друг с другом.
- Сами развертывания выполняются полностью вручную, но эта проблема ничто по сравнению с «головной болью» по устранению всех багов, обнаруживаемых после развертывания.

Если работать с настоящим источником «головной боли» в проекте Rebellious Hamster, то имеет смысл следовать общему подходу к внедрению CD в унаследованные проекты: начать с модернизации CI и перейти к улучшению автоматизации развертывания, когда это станет безопасным.

Всегда узнавать о сбоях

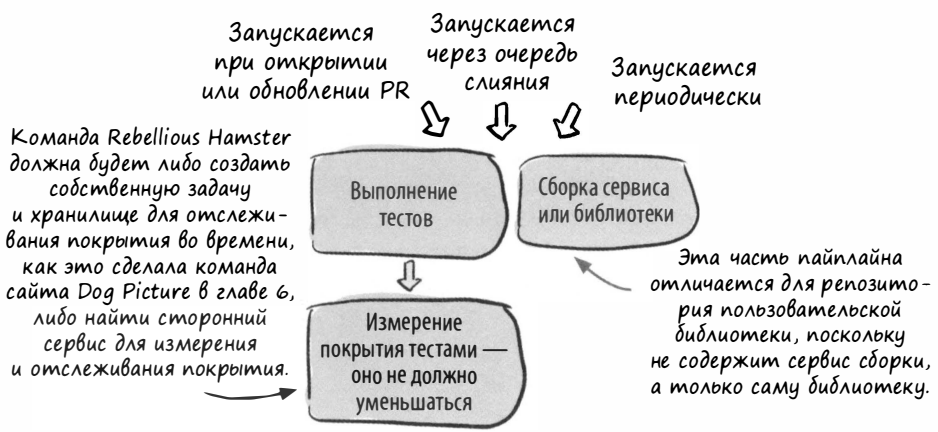
Компания Rebellious Hamster решает отладить CI и начинает так:

1. Автоматизирует процесс, чтобы понимать, пригоден ли код к сборке.
2. Изолирует части кода, которые необходимо улучшить, от частей, на которые не стоит тратить силы, следуя парадигме «разделяй и властвуй».
3. Добавляет тесты, включая измерение покрытия.

Разработчики определяют следующие исходные цели для всех репозиториев:

- Получать информацию о том, что можно успешно проводить сборку сервиса или библиотеки в репозитории.
- Измерять покрытие кода тестами и останавливать пайплайн CD, если оно снижается.

Они создают начальный пайплайн с параметрами, позволяющими повторно использовать его для разных репозиториев (подробнее см. главу 13), который выполняет сборку каждого сервиса (или, в случае библиотеки учетных записей пользователей, создает библиотеку), запускает юнит-тесты, если они существуют, и выдает ошибку, если покрытие юнит-тестами снижается:



Результаты покрытия в каждом репозитории довольно сильно различаются:

- Репозитории аналитики и пользовательских библиотек вообще не имеют покрытия.
- Покрытие API-шлюза и хранилища составляет менее 40 %.
- Покрытие в репозиториях рейтинга и подбора игроков уже выше 60 %.

Несмотря на разные отправные точки, если покрытие перестанет снижаться, со временем оно естественным образом начнет увеличиваться в каждом проекте.

Изолируем код и добавляем тесты

Достигнув поставленных целей и начав получать сигналы о сбоях содержимого из каждого репозитория, разработчики Rebellious Hamster вернулись к рекомендованному пути оптимизации CI унаследованного проекта:

1. ~~Автоматизируйте процесс, чтобы понимать, пригоден ли код к сборке.~~
2. Изолируйте части кода, которые необходимо улучшить, от частей, на которые не стоит тратить силы, следуя парадигме «разделяй и властвуй».
3. Добавьте тесты, ~~включая измерение покрытия~~.

Они переходят к улучшению тестов, увеличивая базовое покрытие, но только там, где это стоит делать. Они также решают добавить первоначальный сквозной тест, поскольку многие баги обнаруживаются только после развертывания всех сервисов вместе. Для начала они выбирают репозитории с минимальным покрытием:

- Репозитории аналитики и пользовательских библиотек вообще не имеют покрытия.
- Покрытие API-шлюза и хранилища составляет менее 40 %.

Не пытаясь достичь 70–80 %+ покрытия для всего кода в каждом репозитории, они ищут в репозиториях код, который действительно регулярно обновляется. Некоторые пакеты и библиотеки не менялись годами, поэтому их не трогают (об изоляции кода, к которому вообще не планируется применять стандарты CI, см. в главе 4).

Для написания хороших юнит-тестов часто требуется рефакторинг кода. В унаследованном коде он может быть очень затратным, поэтому разработчики Rebellious Hamster добавляют простые юнит-тесты, где это возможно, и время от времени проводят интеграционные тесты, когда рефакторинг не стоит усилий. Это приносит достаточный результат и позволяет соблюдать подход «оставь код чище, чем он был до тебя», при котором код становится лучше, чем был, что облегчит постепенное проведение рефакторинга в будущем.

На этот раз мы сразу вкладываемся в сквозные тесты?

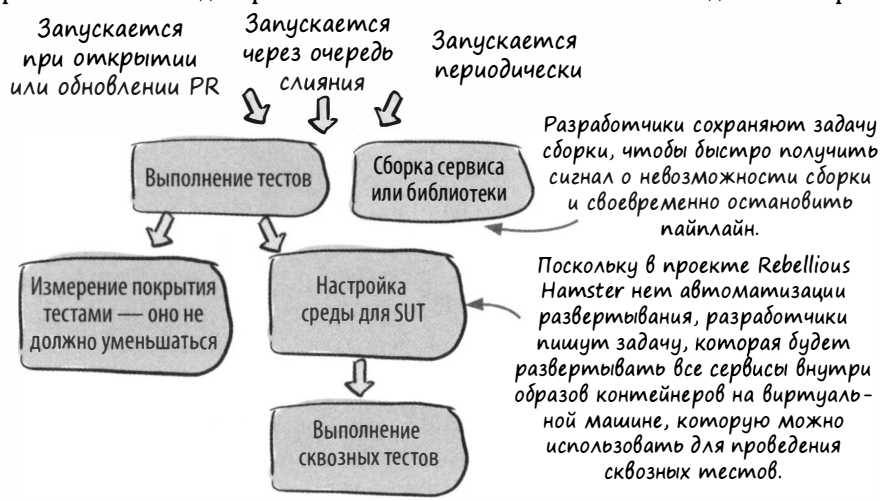
В проектах «с нуля» мы оставляли сквозные тесты напоследок. В унаследованных проектах, возможно, стоит заняться ими раньше, потому что добавлять юнит-тесты на этом этапе сложнее (и, возможно, их уже достаточно), а даже один сквозной тест, если таковых раньше вообще не было, может дать действительно ценный сигнал. В частности, в проекте Rebellious Hamster нет тестов, проверяющих совместную работу всех сервисов до развертывания. Сквозной тест поможет компании гораздо раньше получить сигнал о том, смогут ли сервисы успешно работать вместе.

Добавление большего количества тестов в унаследованный пайплайн

Изолируя код в репозиториях API-шлюза, аналитики, хранилища и пользовательской библиотеки, который не меняется, и увеличивая покрытие остального кода (используя интеграционные тесты, когда юнит-тесты требуют значительного рефакторинга), команда Rebellious Hamster увеличивает покрытие тестами в репозиториях. Теперь покрытие кода, который часто обновляется, во всех репозиториях превышает 60 %.

Кроме того, разработчики пишут несколько сквозных тестов, которые охватывают всю развернутую систему, со всеми сервисами. Это довольно значительная работа, поскольку у них до сих пор вообще не было автоматизации развертывания. Чтобы понять, как осуществить развертывание каждого сервиса, требуется приложить много усилий, не говоря уже о том, чтобы понять, как автоматизировать процессы, чтобы быстро проводить тестирование в пайплайне.

Однако они скоро убедились, что их усилия оправдались. Новый сквозной тест за первые несколько недель работы выявил несколько багов во взаимодействии сервисов.



Как настроить SUT для такого сквозного теста?

Разобраться, с чего начать написание задачи, которая будет осуществлять развертывание продукта (для сквозного тестирования), может оказаться чрезвычайно сложно, особенно если у вас нет автоматизации, на которую можно было бы ориентироваться. Для некоторых продуктов это может быть вообще нецелесообразным, и в этом случае имеет смысл сначала улучшить развертывание, а затем уже вернуться к сквозным тестам. Стоит также рассмотреть использование образов контейнеров. Если вы сможете создать образ контейнера для каждого сервиса, вы сможете запустить все эти образы вместе на одной машине, создав тестовую среду, которая не очень похожа на рабочую, но может быть достаточной, чтобы находить очевидные баги во взаимодействии сервисов.

Увеличение степени автоматизации развертывания

Команда Rebellious Hamster создала надежный пайплайн CI и может переходить к улучшению процесса развертывания:

- Оценить возможности сторонних инструментов развертывания: решить, следует ли автоматизировать существующие процессы или лучше начать с нуля в рамках нового подхода.
- Автоматизировать существующие процессы постепенно, по частям.
- При переходе на сторонний инструмент разработать безопасные эксперименты с минимальным воздействием для перехода от текущих процессов к новому инструменту.

В первую очередь нужно решить, стоит ли сразу переходить на стороннее решение для развертывания или сначала автоматизировать и улучшить текущие процессы. Переход на сторонний инструмент может быть оправдан, если вы хотите попробовать более продвинутые методы развертывания, такие как сине-зеленые и канареечные (см. главу 10). Разработчики Rebellious Hamster, возможно, когда-нибудь и захотят сделать это, но сейчас они выбирают второй вариант. Следующие шаги помогут им постепенно перейти от ручных процессов к автоматизированным:

1. Задокументируйте все ручные операции.
2. Преобразуйте эту документацию в один или несколько скриптов или инструментов.
3. Сохраните скрипты и инструменты в системе контроля версий (то есть внедрите принцип «конфигурация как код»).
4. Создайте автоматизированный запуск развертываний (например, веб-интерфейс с кнопкой, срабатывающей на событие слияния).

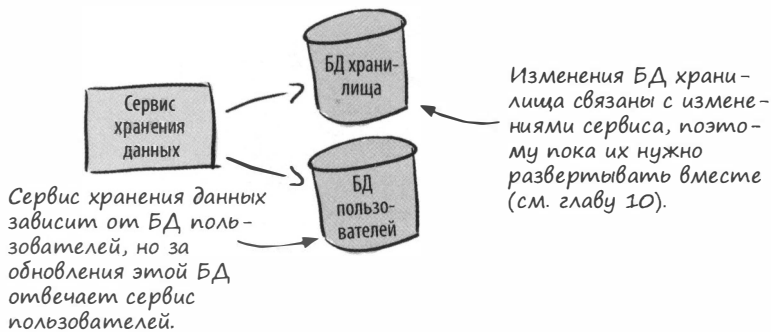
Текущий процесс включает ручную установку двоичных файлов на компьютеры, и разработчики решают (особенно на волне успеха контейнерного подхода в сквозных тестах), что в рамках этой модернизации они начнут упаковывать двоичные файлы в образы контейнеров и запускать их как контейнеры.

Что, если все-таки сразу использовать сторонний инструмент?

Вы не прогадаете, решив попробовать, и это справедливо для большинства изменений, которые вы, возможно, захотите внести при автоматизации развертывания. Для отработки новой техники развертывания (стороннего инструмента или новой автоматизации) выберите проект с наименьшим риском (еще лучше, если это будет совершенно новый проект). Когда вы поймете, как реализовать новый механизм в выбранном проекте, вы сможете внедрять его и в остальных проектах.

Создание пайплайна выпуска

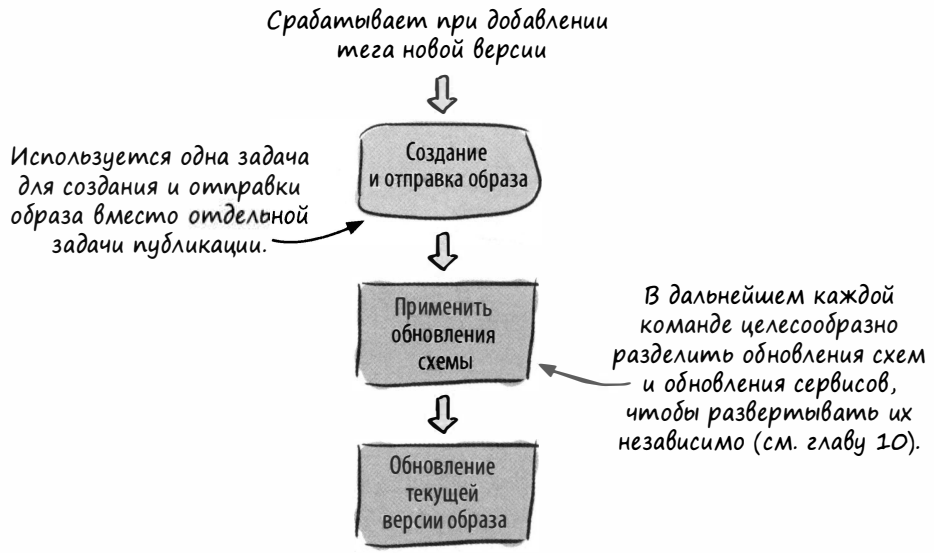
Команда Rebellious Hamster решает начать с сервиса хранения данных, поскольку его архитектура похожа на архитектуру большинства других сервисов, а также он создает определенные сложности, связанные с тем, что он опирается на базу данных:



1. *Документирование.* Члены команды сервиса хранения данных создают документ, в котором подробно описывают процесс развертывания. Сначала сервис хранения собирается в виде двоичного файла, который загружается в реестр артефактов Rebellious Hamster. Затем один из разработчиков вручную применяет все необходимые обновления схемы базы данных к БД хранилища и вручную же обновляет двоичный файл, установленный на виртуальной машине, на которой работает сервис.
2. *Преобразование документации в скрипты.* Команда сервиса хранения данных создает три скрипта: один для сборки сервиса хранения в образ контейнера и загрузки его в реестр образов, другой — для обновления схемы базы данных и, наконец, третий — для обновления образа на виртуальной машине до последней версии.
3. *Внедрение принципа «конфигурация как код».* Все три скрипта фиксируются в системе контроля версий в том же репозитории, что и код сервиса хранения данных и схемы БД.
4. *Создание автоматизированного запуска развертываний.* Разработчики создают пайплайн, который выполняет скрипты, и добавляют триггер для запуска пайплайна в ответ на событие — появление в репозитории новой версии, снабженной тегом. Это позволит автоматизировать процесс запуска, но при этом сохранить ручной контроль.

Пайплайн выпуска Rebellious Hamster

Пайплайн, который создает команда сервиса хранения данных, в итоге выглядит так:



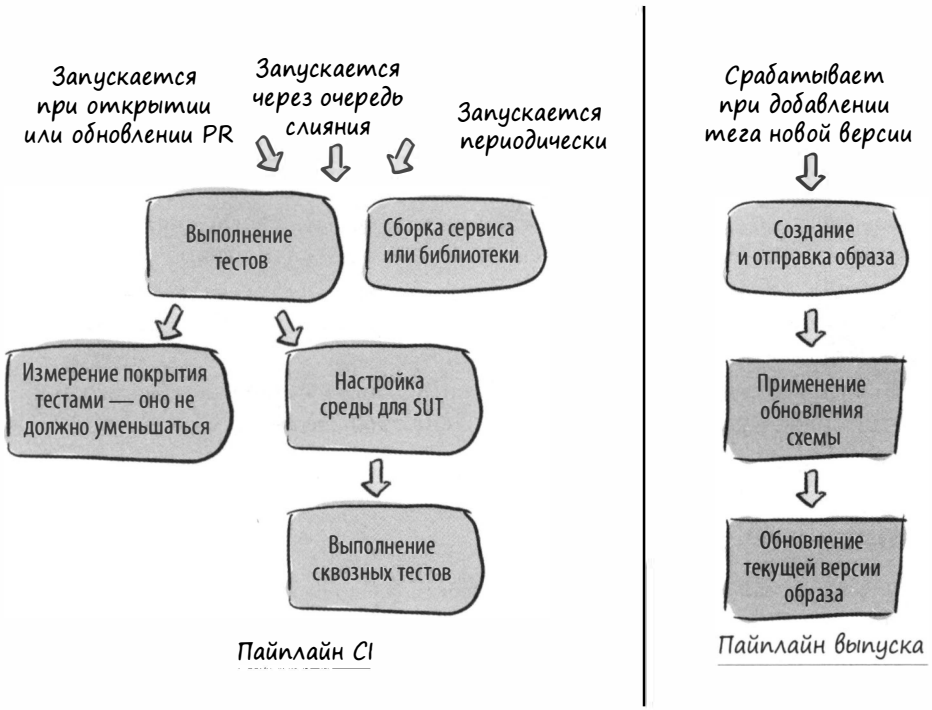
Этот же пайплайн выпуска можно использовать во всех репозиториях Rebellious Hamster с некоторыми изменениями для сервисов, не привязанных к базе данных.

Стоит ли автоматизировать развертывание и вносить большие изменения одновременно?

Команда Rebellious Hamster решила перейти к развертыванию на основе образов контейнеров и одновременно автоматизировать развертывание. Более медленный и поэтапный подход заключается в том, чтобы автоматизировать имеющиеся операции и только потом переходить к использованию образов контейнеров. Выбор подхода зависит от степени допустимого риска: менять несколько процессов одновременно более рискованно, но позволяет быстрее достичь цели.

Окончательный вид пайплайнов Rebellious Hamster

Теперь для каждого репозитория в проекте Rebellious Hamster существует два пайплайна, запускающихся отдельно. Благодаря более высокой степени автоматизации стало возможным проводить независимое развертывание каждого сервиса, и команда Rebellious Hamster может безопасно экспериментировать с более частыми развертываниями, а затем и с более сложными стратегиями развертывания.



При добавлении CD в унаследованные проекты важно учитывать особенности проекта и принять тот факт, что создаваемые пайплайны не будут идеальными и не смогут включить все, что вы хотите. Например, в пайплайнах выше существует несоответствие между тем, как происходит сборка и развертывание сервисов на продакшен, и тем, как они собираются и развертываются для сквозного тестирования.

Это можно будет постепенно исправить в будущем, но сначала обратите внимание, насколько улучшилось состояние проектов. Даже если разработчики Rebellious Hamster не будут модернизировать эти пайплайны еще несколько лет, они могут быть совершенно уверены, что теперь их код всегда готов к релизу, а развертывание стало проще чем когда-либо.

Заключение

Как будут развиваться проекты Gulpy и Rebellious Hamster? Непрерывную доставку можно совершенствовать постоянно! В пайплайнах всегда будет что-то, что можно улучшить; проще всего взглянуть на базовые пайплайны в начале главы и поискать недостающие элементы, или, возможно, они выяснятся в будущем.

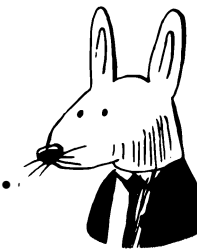
Итоги

- Базовыми элементами эффективных пайплайнов CD (одного или нескольких) являются линтинг, юнит-тесты, интеграционные тесты, сквозные тесты, сборка, публикация и развертывание.
- В коде, разрабатываемом с нуля, устанавливайте высокие стандарты качества сразу, чтобы их было удобно поддерживать (и корректировать) в течение всего срока жизни проекта.
- Работая с унаследованным кодом, лучше всего начать с улучшения CI для кода, который действительно часто меняется, а не пытаться исправить все сразу.
- Примите тот факт, что вы никогда не сможете реализовать все, и это нормально! Улучшение пайплайнов CD все равно того стоит, и даже небольшие изменения могут принести заметную пользу.
- Продолжайте модернизировать CD, обращая внимание на «болевые точки», и правильно расставляйте приоритеты. Чем дольше вы откладываете решение проблемы, тем серьезнее она становится!

Далее...

В следующей главе мы познакомимся с одним из основных строительных блоков пайплайна CD, который мы еще подробно не рассматривали. Этому компоненту редко уделяется должное внимание, и это минимальный скрипт.

Скрипты — это тоже код | 12



В этой главе

- Разработка задач с сильной связностью (cohesion) и слабой связанностью (coupling), которые будут использованы в пайплайнах
- Использование подходящего языка в нужное время, чтобы писать надежные и удобные в сопровождении пайплайны CD
- Выявление компромиссов между написанием задач с применением языков shell-скриптов (таких, как bash) и языков общего назначения (таких, как Python)
- Поддержка работоспособности и удобства сопровождения пайплайнов CD путем применения концепции «конфигурация как код» к скриптам, задачам и пайплайнам

Если внимательно проанализировать пайплайны и задачи, то можно заметить, что в их основе всегда лежат скрипты. Иногда кажется, что непрерывная доставка — это просто множество тщательно организованных скриптов, в частности bash-скриптов.

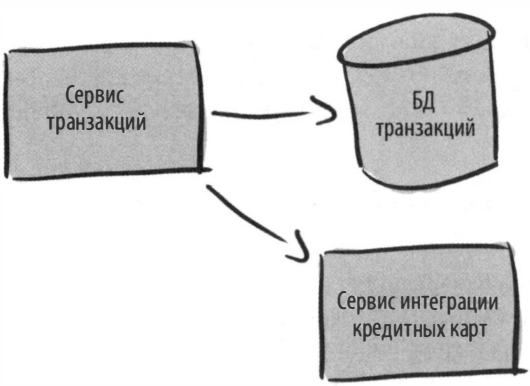
В этой главе мы рассмотрим концепцию «конфигурация как код» немного подробнее и убедимся, что можно применять ее к скриптам, которые мы используем для определения логики CD внутри задач и пайплайнов. Это часто означает необходимость перехода от языка shell-скриптов, такого как bash, к более универсальному языку. Давайте разберемся, как работать со скриптами CD как с кодом.

Онлайн-банк Purrfect Bank

PurrfectBank — это онлайн-банк, ориентированный на рынок банковских услуг для кошек. Владельцы кошек заводят счета для своих питомцев, выплачивают им содержание и позволяют совершать онлайн-покупки с помощью кредитной карты Purrfect Bank.

Ладно, на самом деле покупки совершают не кошки; очевидно, что кошки не умеют пользоваться компьютерами или кредитными картами. Но с Purrfect Bank их хозяева могут делать это понарошку.

Команды разработчиков Purrfect Bank разделены на подразделения, которые в основном работают независимо. В последнее время у подразделения платежей возникли проблемы с пайплайнами CD. Подразделение платежей отвечает за два сервиса: сервис транзакций и сервис интеграции кредитных карт.

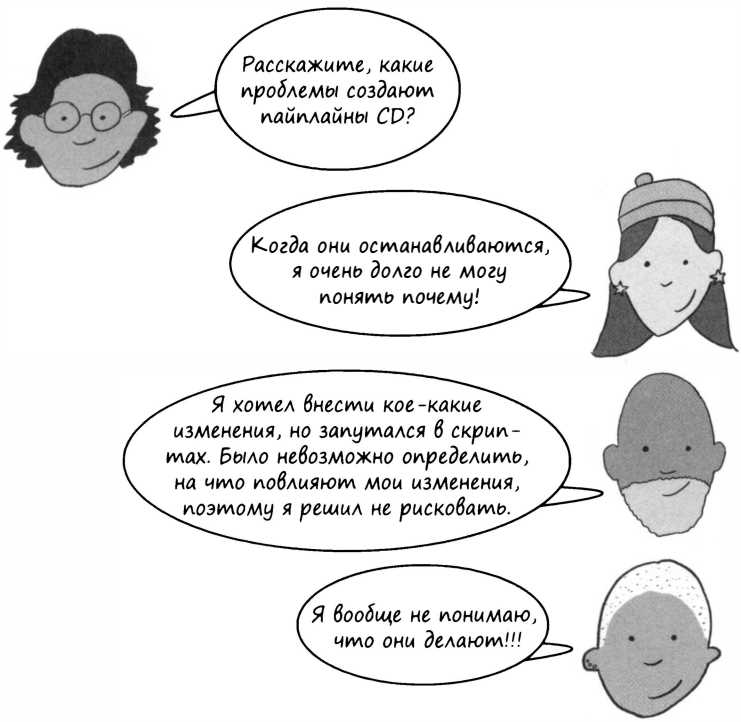


Сервис транзакций опирается на базу данных и использует сервис интеграции кредитных карт, чтобы отправлять запросы провайдерам кредитных карт, с которыми сотрудничает Purrfect Bank.

Проблемы непрерывной доставки

С сервисом транзакций и сервисом интеграции кредитных карт работают отдельные команды, и в последнее время их члены жалуются, что пайплайны CD замедляют их работу, особенно пайплайны CI. Некоторые даже предлагали вообще от них избавиться!

Лоренцо, технический руководитель подразделения платежей в Purrfect Bank, серьезно относится к этим жалобам и пытается найти решение проблем, с которыми сталкиваются команды, не отказываясь полностью от CI.



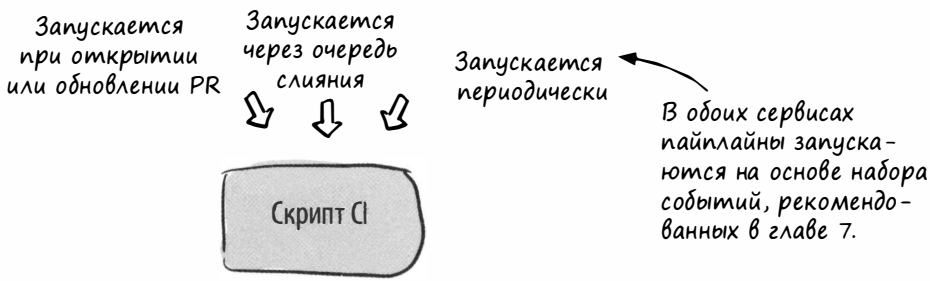
Лоренцо резюмирует проблемы с пайплайнами CD:

- Их трудно отлаживать.
- Они непрозрачны.
- Разработчики не решаются вносить в них изменения.

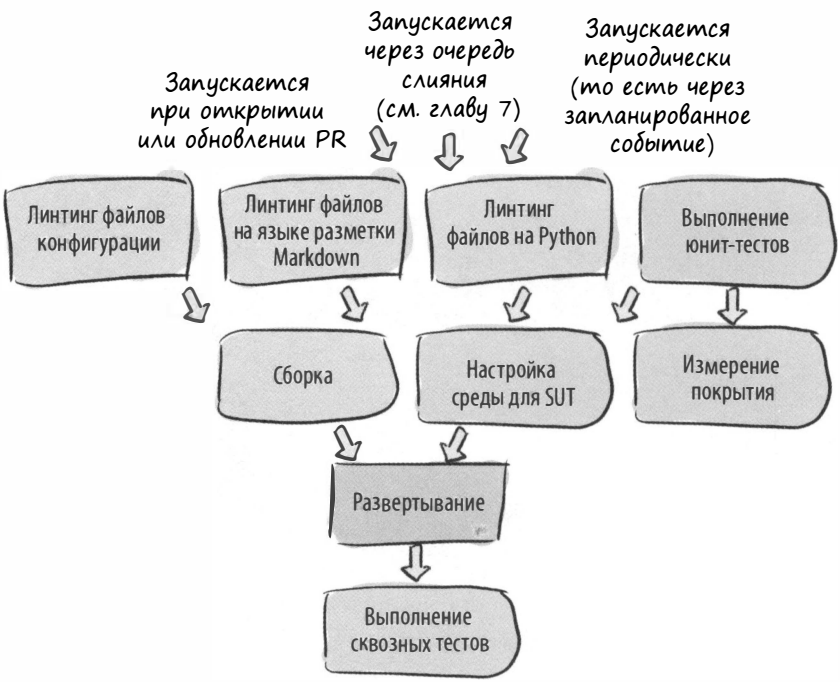
В общем, по мнению Лоренцо, эти пайплайны *сложно использовать* и *трудно поддерживать*.

Схема непрерывной доставки в онлайн-банке Purrfect Bank

Чтобы понять, почему с пайплайнами CD подразделения платежей так много проблем, Лоренцо проводит их анализ. Есть два пайплайна, один для сервиса транзакций и один для сервиса интеграции кредитных карт. Пайплайн сервиса транзакций — это на самом деле просто одна гигантская задача, которая запускает bash-скрипт:



Пайплайн сервиса кредитных карт выглядит немного лучше и состоит из нескольких задач:



Оба этих пайплайна, хотя и имеют разную структуру, опираются на общую библиотеку bash-скриптов.

Bash-библиотеки подразделения платежей

Команды подразделения платежей в Purrfect Bank размещают свой код в следующих репозиториях:

Репозиторий CI-скриптов содержит ряд bash-скриптов, включая следующие:

- linting.sh;
- unit_tests.sh;
- e2e_tests.sh.

Два других репозитория содержат копии репозитория CD-скриптов (которые регулярно обновляются при изменении репозитория CD-скриптов), и их задачи импортируют эти библиотеки. Например, начало bash-библиотеки linting.sh выглядит так:

```
#!/usr/bin/env bash
set -xe
function lint() {
        local command=$1
        local params=$2
        shift 2
        for file in $@; do
                echo "${command} ${params} ${file}"
                ${command} ${params} ${file}
        done
}
function python_lint() {
        lint "python3" "-m pylint" $@
}
```

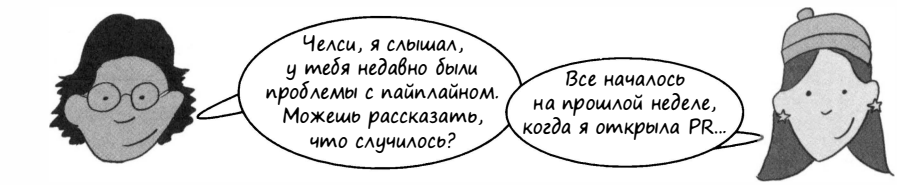
Функции для выполнения конкретных видов линтинга вызывают обобщенную функцию lint().

Стоит ли вообще заводить bash-библиотеку?

Хороший вопрос! Некоторые атрибуты командной оболочки bash делают ее не совсем подходящей для создания повторно используемых библиотек — мы подробнее поговорим об этом позже. А пока достаточно сказать, что это, возможно, не лучший выбор, но если вам требуются общие bash-скрипты для задач и проектов, можно использовать и такой подход. Стоит ли? Наверное, нет — вскоре мы к этому вернемся!

Пайплайн сервиса транзакций

Лоренцо решает начать с сервиса транзакций. Весь пайплайн представляет собой одну задачу и один большой скрипт, который ее поддерживает. У Лоренцо есть ощущение, что это и есть источник проблем команды сервиса транзакций, но он хочет проверить свои подозрения.



Вторник
- Я открыла PR с новой функцией «Подарочная карта».
- Пайплайн почти сразу же выдал ошибку.
- Немного повозившись, я поняла, что линтинг нашел мою ошибку, и я ее исправила.
- Пайплайн снова выдал ошибку, на этот раз через несколько минут.
- После небольшого анализа я поняла, что нарушила работу кое-каких юнит-тестов, и исправила их.
- Пайплайн снова дал сбой, на этот раз более чем через полчаса после исправления.

Среда
- Мне потребовалось еще больше времени, чтобы понять, что нужно обновить способ сборки сервиса, чтобы адаптировать его к новой функции. Исправить удалось только на следующий день.

Четверг
- Почти через час после этого, разумеется, пайплайн снова дал сбой! Оказалось, что нужно было обновить сквозной тест, но к этому моменту я возилась с пайплайном уже несколько дней и была жутко зла!

Выслушав Челси, Лоренцо понял, что это подтверждает его теорию: *сигнал*, который Челси получила в качестве отклика от пайплайна при сбое, не содержал достаточно информации, и ей потребовался дополнительный анализ, например просмотр журнала выполнения (лог-файлов). Поскольку пайплайн состоял из одной огромной задачи, единственный сигнал, который могла получить Челси, — это либо «*пайплайн прошел*», либо «*пайплайн не прошел*», и ей приходилось копаться в деталях, чтобы понять, что это значит.

> Подробнее о *сигналах* и *шуме* читайте в главе 5, а обо всех точках жизненного цикла изменений, где необходимы сигналы для выявления багов, — в главе 7.

Избавляемся от одного большого скрипта

Лоренцо ставит перед сервисом транзакций цель: перейти от текущего состояния пайплайна, когда подается только один сигнал (либо «*пайплайн прошел*», либо «*пайплайн не прошел*»), к состоянию, когда пайплайн может выдавать несколько отдельных сигналов. В ситуации, с которой столкнулась Челси, это сэконо́мит много времени на анализ и позволит сразу понять, где ошибка, и сосредоточить усилия в нужном месте. Выясняя это, Лоренцо случайно открыл эмпирическое правило для определения границ задачи:

> **Для каждого сигнала, который требуется от пайплайна, должна быть предусмотрена отдельная задача.**

Лоренцо изучает текущий скрипт и определяет сигналы, которые были бы полезны команде:

```
#!/usr/bin/env bash

set -xe

source $(dirname ${BASH_SOURCE})/linting.sh
source $(dirname ${BASH_SOURCE})/e2e_tests.sh
source $(dirname ${BASH_SOURCE})/unit_tests.sh

config_file_lint $@
markdown_lint $@
python_lint $@

run_unit_tests
measure_unit_test_coverage

build_image "purrfect/transaction" "image/Dockerfile"
setup_e2e_sut
deploy_to_e2e_sut "purrfect/transaction"
run_e2e_tests
```

Все библиотеки bash должны быть включены в этот скрипт.

Каждый вызов функции в этом скрипте может стать отдельной задачей, генерирующей отдельный сигнал.

Реальные bash-скрипты, скорее всего, будут выглядеть гораздо сложнее, чем этот. Этот скрипт сильно упрощен для целей данной книги.

> ***Какие-то из описанных процессов наверняка должны выполняться параллельно?***
>
> Очень верно подмечено: мало того что все это делается в рамках одной большой задачи, так еще и каждый вызов функции блокируется предыдущим, хотя иногда для этого нет никаких причин. В новом пайплайне Лоренцо обновит часть кода, чтобы он выполнялся параллельно; подробнее об этом см. также в главе 13.

Принципы хорошо спроектированной задачи

Цель Лоренцо — перейти от одного-единственного сигнала (либо «*пайплайн прошел*», либо «*пайплайн не прошел*») к отдельным задачам для каждого полезного сигнала, следуя правилу:

> **Для каждого сигнала, который требуется от пайплайна, должна быть предусмотрена отдельная задача.**

В дополнение к этому правилу существуют и другие принципы грамотного проектирования задач. Полезно относиться к задачам как к разработке функций. Хорошо продуманные задачи обладают следующими свойствами:

- *Сильная связность* (хорошо выполняют одну задачу).
- *Слабая связанность* (могут использоваться повторно и объединяться с другими задачами).
- Имеют *четко определенные интерфейсы с понятными действиями* (вход и выход).
- *Делают ровно столько, сколько нужно* (не слишком мало, но и не слишком много).

Как и в случае с написанием чистого кода, разработчики по-разному могут интерпретировать эти принципы, однако чем больше опыта разработки задач вы приобретаете, тем лучше у вас это получается. На самом деле главное — помнить об этих принципах и стараться их применять, а не стремиться к созданию идеальной задачи. Тем не менее есть признаки, которые говорят о том, что задача, возможно, слишком раздута:

- Вы замечаете, что дублируете части задачи в других задачах (например, копируете логику загрузки результатов в несколько задач).
- Задача содержит логику оркестрации (например, циклическое обращение к входным данным для выполнения одного и того же действия несколько раз или проверка завершения одного действия перед запуском другого).

Эти функции лучше подходят для пайплайна; задачи должны формировать логику сильной связности и слабой связанности, а пайплайны — организовывать комбинации этой логики.

Разделение гигантской задачи на части

Анализируя большую задачу, которая составляла весь пайплайн сервиса транзакций, Лоренцо выделил девять отдельных сигналов, которые он хотел бы получать от отдельных задач (которые, в свою очередь, могут быть объединены и управляться самим пайплайном):

1. `config_file_lint`
2. `markdown_lint`
3. `python_lint`
4. `run_unit_tests`
5. `measure_unit_test_coverage`
6. `build_image`
7. `setup_e2e_sut`
8. `deploy_to_e2e_sut`
9. `run_e2e_tests`

Разделив эту задачу на части совместно с командой сервиса транзакций, Лоренцо получил задачу для каждого из предыдущих вызовов функций:

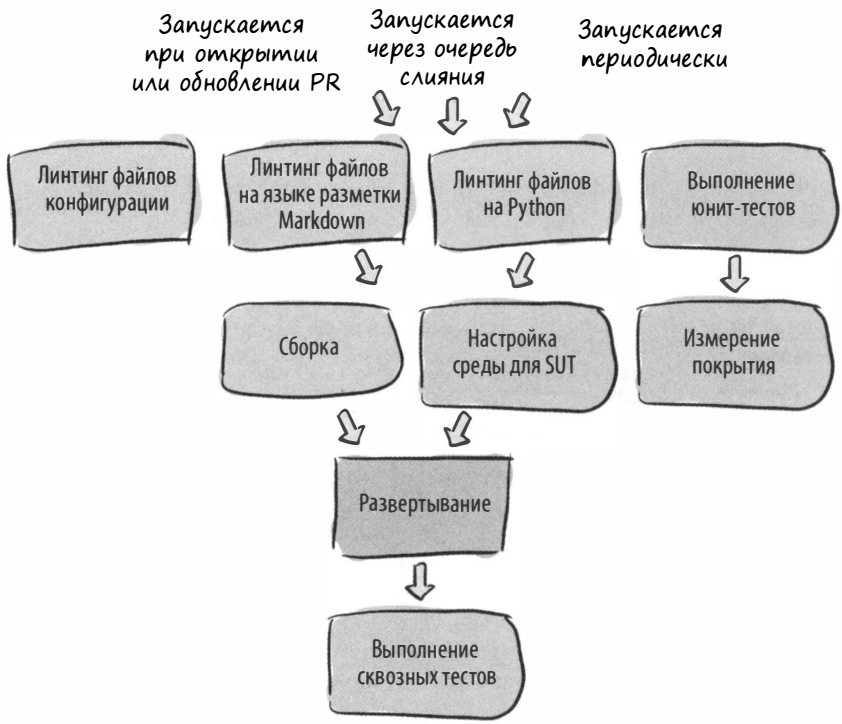
Чтобы минимизировать количество багов, вносимых при разбиении на эти задачи, Лоренцо постарался сохранить их как можно ближе к оригиналу. Он старается (пока) не менять базовые библиотеки bash, насколько это возможно, поэтому создаваемые им задачи очень похожи на предыдущий большой bash-скрипт; они просто делают меньше. Например, вот как выглядит задача линтинга файлов на Python:

```
#!/usr/bin/env bash
set -xe

source $(dirname ${BASH_SOURCE})/linting.sh
source $(dirname ${BASH_SOURCE})/e2e_tests.sh
source $(dirname ${BASH_SOURCE})/unit_tests.sh

python_lint $@
```

На этом начальном этапе все библиотеки bash все еще импортируются в скрипт задачи Python-линтинга, потому что эти библиотеки могут содержать важные побочные эффекты, которые требуются для функции python_lint. Благодаря инкрементному подходу, когда изменения минимальны, переход к новой практике работы станет простым и плавным и ее можно будет постепенно улучшать.

Обновленный пайплайн сервиса транзакций

Эти задачи кажутся Лоренцо знакомыми, и он понимает, что эти же задачи используются в пайплайне сервиса интеграции кредитных карт. Немного поработав совместно с двумя командами, он обновил эти два пайплайна и сделал их практически одинаковыми: они имеют одну и ту же форму и используют в основном одни и те же задачи. Теперь оба пайплайна выглядят примерно так:



Теперь команда сервиса транзакций не запускает одну гигантскую задачу, а обе команды подразделения платежей работают одинаково: их пайплайны CD генерируют несколько сигналов, по одному на каждую задачу, причем каждая задача обращается к библиотекам bash, которые они совместно используют через репозиторий скриптов CD.

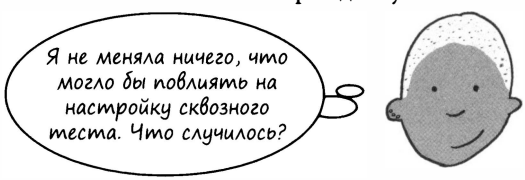
ВАЖНО

Хорошо продуманные задачи похожи на хорошо разработанные функции. Они обладают сильной связностью, слабой связанностью, имеют четко определенные интерфейсы и делают ровно столько, сколько нужно. А благодаря инкрементному подходу перейти от задач со слишком большим количеством действий к хорошо продуманным задачам становится гораздо легче.

Отладка bash-библиотек

Две команды подразделения платежей теперь используют практически одинаковый пайплайн, но это не означает, что Лоренцо решил все проблемы. Команда сервиса интеграции кредитных карт, чей пайплайн более отлажен, по-прежнему регулярно сталкивается со множеством проблем в нем.

Недавно у Лулу, разработчицы из этой команды, возник неприятный баг в пайплайне CD, когда она попыталась открыть PR. Она занималась простым исправлением багов и с удивлением увидела, что задача `Set up SUT Environment` (Настройка среды для SUT) завершилась на ее PR с ошибкой прав доступа.



Чтобы разобраться в проблеме, Лулу изучает скрипт, который использует задача `Set up SUT Environment`:

```
#!/usr/bin/env bash

set -xe

SERVICE_ACCOUNT='cc-e2e-service-account'
source $(dirname ${BASH_SOURCE})/e2e_tests.sh

setup_e2e_sut
```

Она точно ничего здесь не меняла, и логика кажется ей верной: установить переменную среды, которая управляет учетной записью сервиса, используемого функцией `setup_e2e_sut`, а затем импортировать и вызвать функцию. Итак, она открывает в bash-библиотеке функцию `setup_e2e_sut` и видит следующее:

```
unset SERVICE_ACCOUNT
```

Переменная среды SERVICE_ACCOUNT, которую задача устанавливала непосредственно перед загрузкой setup_e2e_sut.sh, сбрасывается при каждой загрузке файла.

```
function setup_e2e_sut() {...
```

Оказывается, в файл e2e_tests.sh была добавлена строка, которая создает побочный эффект загрузки библиотеки/скрипта bash: она намеренно сбрасывает переменную среды, которая управляет учетной записью сервиса, используемой сквозными тестами.

> ***Что такое учетная запись сервиса?***
>
> Детали тут не важны, но при высокоуровневом программировании во многих приложениях (и особенно на облачных платформах) пользователи могут определять *учетные записи сервисов*, которым могут быть предоставлены определенные права доступа. В нашем примере для проведения сквозных тестов нужно предоставить учетную запись сервиса, чтобы настроить необходимую инфраструктуру.

Исследуем баг в bash-библиотеке

С точки зрения Лулу, этот побочный эффект в bash-библиотеке e2e_tests.sh является багом. Она сообщает об этом Лоренцо, который попросил членов команды рассказывать ему обо всех проблемах в работе пайплайна. Лоренцо начинает выяснять, что произошло, изучая сообщение от Аджая из команды сервиса транзакций, которое касается коммита с изменением:

> **Аджай**
>
> Убедитесь, что сквозные тесты начинаются с нуля.
>
> Когда я попытался обновить нашу задачу среды SUT для настройки нескольких сред, я понял, что можно случайно использовать уже установленные переменные среды. Этот коммит обновляет библиотеку, стирая все необходимые переменные среды и начиная с нуля.

Анализируя пайплайн сервиса транзакций, Лоренцо понимает, что задача `Set up SUT environment`, которую использует команда, немного отличается от той, которую использует команда сервиса интеграции кредитных карт:

```
#!/usr/bin/env bash

set -xe

source $(dirname ${BASH_SOURCE})/e2e_tests.sh
SERVICE_ACCOUNT='cc-e2e-service-account'

setup_e2e_sut
```

В этой задаче переменная среды устанавливается после импорта библиотеки e2e_tests.sh, поэтому побочный эффект, вызванный ее импортом, не вызывает проблем.

Другой способ свести к минимуму подобные проблемы — сделать так, чтобы обе команды выполняли абсолютно одинаковые задачи. Лоренцо стремится к этой цели, но пока не достиг ее.

Почему бы вместо этого не передавать учетную запись сервиса в качестве параметра?

Это справедливое замечание: если бы функция `setup_e2e_sut` брала учетную запись сервиса в качестве параметра, а не ожидала установки переменной среды, этой проблемы можно было бы избежать. В нашем примере мы полагаемся на переменные среды. Одним из недостатков скриптов bash является то, что они часто зависят от переменных среды, таких как эта, особенно если несколько функций должны использовать одно и то же значение. Кроме того, bash-скрипты часто вызывают программы, которые сами ожидают, что эти переменные среды будут установлены.

Почему появился этот баг?

Лоренцо понимает, почему для Аджая это изменение кажется логичным, если он учитывает только то, как служба транзакций использует библиотеку e2e_tests.sh. Но он хочет подробнее разобраться, почему никто не догадался о более серьезных последствиях этого изменения. Лоренцо видит, что проверку PR с коммитом Аджая проводила Челси, поэтому он встречается с Аджаем и Челси, чтобы обсудить, что пошло не так.

Лоренцо поблагодарил Аджая и Челси за то, что они рассказали все как есть. Он понял, что они честно старались убедиться, что изменения не вызовут проблем, но все-таки их не избежали, и вот почему:

- При обновлении кода было трудно отследить его использование — и не было тестов, которые показали бы связанные с ним ограничения.
- А поскольку не было тестов и понятного способа их добавления, Аджай не смог включить автоматизированную проверку изменений.
- При анализе кода было очень трудно понять последствия изменений. Не было тестов, которые можно было бы запустить, и проверяющий столкнулся с теми же проблемами, что и автор, поскольку не мог изучить вопрос со всех сторон и определить, на какие задачи и пайплайны может повлиять это изменение.

Для чего нужен bash

Проблемы, с которыми сталкивается команда, особенно отсутствие тестов, заставляют Лоренцо засомневаться, действительно ли bash лучше всего подходит для использования в репозитории CD-скриптов. Хотя он видел, что bash часто используется в пайплайнах CD, он понимает, что никогда не задумывался о том, что это такое и насколько он хорош.

Чтобы разобраться в этом, нужно понять, что такое bash, для чего он подходит, а для чего нет. Bash, сокращение от *Bourne Again Shell*, был создан как замена *Bourne Shell* (или *sh*). Оба они являются своего рода *командными оболочками* (англ. — shell) — текстовыми интерфейсами операционных систем.

Поэтому хотя разработчики часто считают, что bash — просто язык скриптов, на самом деле это нечто большее: это язык, используемый для включения интерфейса в операционную систему. Авторы bash-скриптов могут использовать следующее:

- *конструкции языка bash* (такие, как операторы `if`, циклы `for` и функции, которые вы определяете в своих скриптах);
- *встроенные команды bash* (функции, доступные как часть оболочки, например `echo`);
- *программы*, которые bash умеет находить (исполняемые программы, доступные в каталогах, в которых bash настроен на поиск через переменную среды `PATH`).

Bash и другие распространенные языки shell-скриптов (например, sh и PowerShell) предназначены для управления командами операционной системы. Они действительно хороши, когда вам нужно только запустить команду или программу (особенно если таких команд, разработанных в среде Unix, несколько), чтобы легко получать выходные данные одной команды и передавать их в другую. Иногда в общем говорят, что bash хорош для «конвейерной передачи» (piping stuff around, передача выходных данных одной команды в другую).

> ***В чем заключается философия Unix?***
>
> Не буду вдаваться в детали, но стоит поискать и почитать литературу на эту тему. Вкратце, это набор принципов для определения модульных программ, которые действительно хорошо работают вместе, и этот набор использовался при разработке Unix.

Задачи CD часто сводятся к вызову программы или команды и отчету о результатах (например, вызов Python для запуска юнит-тестов), поэтому понятно, почему языки shell-скриптов так часто появляются в задачах CD и пайплайнах.

> ***Итак, если использовать sh вместо bash, получится ли избежать проблем?***
>
> Замечания, сделанные по поводу bash, в равной степени справедливы для всех распространенных языков shell-скриптов (например, если вы часто работаете с Windows, вы наверняка будете работать с PowerShell), и ко всем этим языкам следует относиться с одинаковым вниманием.

Когда bash не подходит

У bash есть свои сильные стороны, и поскольку он особенно хорошо подходит для задач CD (вызова программ и команд), начинать с него лучше всего. Однако со временем, когда задачи CD растут и усложняются, bash начинает проигрывать. О том, что bash — уже не лучший инструмент, говорят следующие признаки:

- Длина скриптов превышает несколько строк.
- Появляются скрипты, включающие множество условий и циклов.
- Логика становится достаточно сложной, и для нее требуются тесты.
- Возникает логика, которую требуется совместно использовать в различных скриптах, например, с помощью библиотек.

Отсутствие тестирования и должной поддержки определения повторно используемых функций — основные причины, почему bash начинает сыпаться при усложнении логики. Даже простые функции bash крайне ограничены в том, что они могут поддерживать (например, вы не можете возвращать данные из них, только коды выхода, и вместо этого должны полагаться на переменные среды или потоковую обработку, чтобы получить от них значения). Не существует надежных механизмов для версионирования и распространения библиотек этих функций.

Если вы видите, что в скриптах CD появляются какие-то или все указанные проблемы, не стоит пытаться заставить bash делать то, в чем он не слишком хорош, лучше обратиться к языку программирования общего назначения (например, к Python).

Языки общего назначения отлично работают там, где bash не справляется. В частности, они предназначены для поддержки определения хорошо прописанных, повторно используемых функций и библиотек. Эти функции и библиотеки могут поддерживаться тестами и версионированными релизами, как и любое другое ПО.

Большой плюс в том, что любую программу, написанную на языках общего назначения, можно вызвать из bash. Поэтому вопрос надо ставить не об использовании *либо* bash, *либо* языка общего назначения, а скорее об использовании их обоих с пониманием того, в каких ситуациях лучше подходит один, а в каких — другой.

> Вы не стали бы писать на bash рабочие приложения, так зачем же писать код, который вы используете для поддержки разработки, исключительно на bash? Код — это код!

Простота кода и стоимость его сопровождения

Одна из причин, по которой bash так часто встречается в пайплайнах CD, — это распространенный, но довольно вредный подход к разработке: чрезмерная оптимизация в целях упрощения написания кода, а не снижения затрат на его поддержку. Bash-скрипты просты в написании, и когда вам нужно их обновить, добавить там-сям пару строк не составит труда. Но затраты на сопровождение этих простых строк могут быть огромными по сравнению с единовременными издержками на переход к языку с лучшей поддержкой требуемой логики.

Безопасность и скрипты

Использовать скрипты в пайплайнах CD может быть опасно. Как уже говорилось, bash, в частности, предоставляет интерфейс к базовой операционной системе (ОС). Это часто означает, что bash-скрипты имеют широкий доступ к содержимому ОС и могут стать объектом атаки злоумышленников.

Распространенность bash-скриптов в системах CD означает, что в задачах и пайплайнах CD очень часто определяются переменные среды для важной информации, необходимой для выполнения задач, включая конфиденциальные данные, такие как информация об аутентификации. Например, представьте, что в bash-скрипт добавлена переменная среды, содержащая секретный токен:

```
MY_SECRET_TOKEN=qwerty012345qwerty012345qwerty012345
```

Скрипт bash может делать с этим токеном все что угодно, например записать его во внешнюю конечную точку:

```
curl -d $MY_SECRET_TOKEN https://some-endpoint
```

CD-системы, такие как GitHub Actions, предоставляют дополнительные механизмы для работы с секретными данными; например, обычно GitHub Actions сохраняет секрет, а затем в рабочем процессе вы можете привязать его к переменной среды. Код выше может быть таким:

```
- name: Use my secret token
  env:
    MY_SECRET_TOKEN: ${{ secrets.MY_SECRET_TOKEN }}
  run: |
    curl -d $MY_SECRET_TOKEN https://some-endpoint
```

Опасность в том, что если злоумышленник сможет изменить скрипты, он получит полную свободу доступа к этим переменным среды (и любым другим, доступным в операционной системе) и сможет делать с ними все, что захочет.

Один из самых известных примеров этого — атака на инструмент покрытия кода Codecov в апреле 2021 года. Один из злоумышленников смог получить доступ и изменить bash-скрипт, который Codecov использовал для отправки отчетов о покрытии кода. Этот bash-скрипт был загружен и применялся во всех конвейерах CD, которые использовали Codecov для создания отчетов о покрытии. Злоумышленнику осталось только добавить одну строку bash, которая загружала все переменные среды, доступные в контексте скрипта, на удаленную конечную точку. А поскольку конфиденциальная информация скриптам часто предоставляется через переменные среды, это открывало злоумышленнику доступ ко многим системам, и пользователям Codecov пришлось повозиться и повторно вводить все учетные данные, которые могли быть раскрыты. (Узнать больше об этой атаке можно на сайте Codecov по ссылке https://about.codecov.io/security-update/).

Этот случай наглядно показывает, почему стоит быть осторожными со скриптами, которые мы пишем в пайплайнах CD. Подумайте, кто может изменить скрипт и к чему тем самым получить доступ.

Сравнение языков shell-скриптов и языков общего назначения

Функция	Языки shell-скриптов	Языки общего назначения
Базовое управление потоком (например, операторы if, циклы for, функции)	Да. Они поддерживают функции управления потоком; функции в bash не так удобно использовать повторно, как в языках общего назначения	Да
Различные типы данных	Часто очень ограниченно. Теоретически bash поддерживает строки, массивы и целые числа, но все они хранятся как строки	Да
Вызов других программ	Да. Легко, с поддержкой цепочки ввода и вывода между программами	Да, но не так просто. Обычно для этого необходимо обратиться к библиотеке, которая инициирует новый процесс программы и обрабатывает взаимодействие с ним
Тесты	Нет. Почти не поддерживаются как часть самого языка	Да
Отладка	Часто ограниченна. Лишь ограниченное количество инструментов поддерживает отладочные скрипты	Да. Для отладки языка общего назначения часто создаются вспомогательные инструменты, например поддержка IDE для установки точек останова
Версионированные библиотеки и пакеты	Нет	Да
Поддержка синтаксиса IDE	Да	Да
Защита от вредоносных модификаций (см. предыдущую страницу)	Нет. Для утечки большого количества данных достаточно одной строки в bash	Нет. Языки общего назначения также можно использовать для извлечения конфиденциальных данных. Однако шансы, что будут использоваться передовые методики (например, проверка, хорошо продуманный код), выше, что потенциально снижает риск

Словарик

Языки программирования общего назначения были созданы для использования в различных ситуациях и областях, в отличие от языков, созданных для конкретных узкоспециализированных целей. В этой главе они сравниваются с языками shell-скриптов (предназначенными для вызова из оболочки операционной системы). Примерами языков общего назначения являются Python, Go, Ruby и C.

От языка shell-скриптов к языку общего назначения

Лоренцо решает, что поддержка bash-библиотек в репозитории скриптов CD приносит больше неудобств, чем пользы, и разрабатывает план переноса этих библиотек с bash на другой язык. Он предлагает команде три варианта, как это сделать, выбрав Python в качестве языка для более сложной логики, которая может появиться в будущем:

- Преобразовать функции bash в независимый инструмент, написанный на Python.
- Преобразовать функции bash в библиотеки Python, которые можно вызывать из отдельных инструментов или Python-скриптов.
- Преобразовать функции bash в повторно используемые задачи.

> ***Как определять и распределять повторно используемые задачи?***
>
> Ответ зависит от используемой системы CD. Если вы работаете с GitHub Actions, это делается путем определения собственных пользовательских задач Actions. Как эта функция работает в других системах CD, см. в приложении А.

Если следовать этому плану, репозиторий скриптов CD будет содержать повторно используемые версионированные инструменты, библиотеки и задачи, а не кучу bash-скриптов без указания версий. Лоренцо изучает каждый bash-файл в репозитории и функции, которые он содержит, чтобы решить, какие из них лучше перенести на другой язык:

- linting.sh: **`python_lint`**
- linting.sh: **`markdown_lint`**
- linting.sh: **`config_file_lint`**
- unit_tests.sh: **`run_unit_tests`**
- unit_tests.sh: **`measure_unit_test_coverage`**
- e2e_tests.sh: **`build_image`**
- e2e_tests.sh: **`setup_e2e_sut`**
- e2e_tests.sh: **`deploy_to_e2e_sut`**
- e2e_tests.sh: **`run_e2e_tests`**

В случаях с linting.sh и unit_tests.sh Лоренцо обнаружил, что содержимое функций bash на самом деле просто вызывает программы и сообщает о результатах, поэтому имеет смысл сохранить их в bash. Он предлагает создать для них задачи в репозитории скриптов CD и вызывать как есть из пайплайна CD каждого сервиса.

В e2e_tests.sh логика становится все более сложной (о чем Лулу, Аджай и Челси узнали не понаслышке), особенно функция `setup_e2e_sut`, поэтому Лоренцо предлагает преобразовать ее в версионированную библиотеку Python, а командам создать и поддерживать инструмент для настройки сквозных тестов, использующих ее.

План перехода

Лоренцо предлагает следующий план перехода от библиотек bash к повторно используемым задачам:

1. Преобразовать функции в файлах linting.sh и unit_tests.sh в многоразовые задачи. Они по-прежнему будут использовать bash, но пайплайны сервиса транзакций и сервиса интеграции кредитных карт не будут совместно использовать bash в качестве библиотек, а смогут ссылаться на определения задач как таковых (нет необходимости совместно использовать bash или копировать и вставлять определения).
2. Чтобы подготовиться к преобразованию функций из bash в Python, следует создать исходную пустую библиотеку Python и автоматизировать создание релизов версий (см. главу 9 о создании версий). А затем сделать то же самое для всех новых инструментов, которые планируется создать.
3. Для каждой функции, которая должна быть преобразована из bash в Python, следует:
   а) решить, в какой пакет поместить функцию, и либо создавать новый пакет, либо добавить ее в уже существующий;
   б) переписать функцию на Python;
   в) добавить тесты для этой функции.

По мере постепенного обновления функций Лоренцо также обновляет пайплайны, которые их использовали, чтобы учесть новые версии, либо обновляя пайплайн для работы с новыми задачами, либо обновляя существующие задачи, чтобы они могли использовать Python-скрипты (и импортировать новые библиотеки), либо обновляя существующие задачи, чтобы вместо них использовать новый комплексный инструмент на основе Python.

Следует ли хранить все скрипты CD в одном репозитории?

Работайте со скриптами как с любым другим кодом: как с рабочим кодом бизнес-логики. Вначале работа с одним репозиторием может упростить задачу, но старайтесь, чтобы он в результате не превратился в беспорядочную кучу случайных функций. Использование отдельных репозиториев также может облегчить версионирование каждой библиотеки и задачи в отдельности; в противном случае присваивать номера версий в репозитории может быть сложнее. Например, тег версии `v0.1.0` будет применяться ко всему коду в репозитории, который может состоять из нескольких задач, библиотек и инструментов.

От библиотеки bash к задаче с bash-скриптом

Одна из первых функций bash, за которую берется Лоренцо, — это функция, выполняющая линтинг Python. Функция вызывалась из пайплайна службы транзакций следующим образом:

```
#!/usr/bin/env bash
set -xe

source $(dirname ${BASH_SOURCE})/linting.sh
source $(dirname ${BASH_SOURCE})/e2e_tests.sh
source $(dirname ${BASH_SOURCE})/unit_tests.sh

python_lint $@
```

Чтобы понять весь код, необходимый для ее работы, Лоренцо изучает файлы linting.sh, e2e_tests.sh и unit_tests.sh, которые используются этим скриптом, и обнаруживает, что имеет значение только код из linting.sh, а именно две функции, `lint()` и `python_lint()`:

```
function lint() {
        local command=$1
        local params=$2
        shift 2
        for file in $@; do
                ${command} ${params} ${file}
        done
}
function python_lint() {
        lint "python" "-m pylint" $@
}
```

Функция lint() в библиотеке bash нужна, чтобы цикл можно было повторно использовать в нескольких функциях линтинга. Теперь этот цикл будет повторяться в каждой из них, зато больше не нужно возиться с библиотеками bash — это стоит небольшого количества повторов.

Цель Лоренцо — превратить функциональность линтинга в компактный автономный bash-скрипт, который можно использовать внутри задачи, не прибегая к внешним библиотекам. Если бы функция `python_lint()` включала в себя логику функции `lint()`, то после выполнения всех шагов получилось бы следующее:

```
function python_lint() {
        for file in $@; do
                python -m pylint ${file}
        done
}
```

Половина строк в предыдущей функции lint() была просто обработкой аргументов функции, поэтому Лоренцо нужно было просто добавить сюда цикл.

Таким образом, чтобы инкапсулировать всю функциональность линтинга в один автономный bash-скрипт, нужно всего лишь сделать так:

```
#!/usr/bin/env bash
set -xe
for file in $(find ${{ inputs.dir }} -name "*.py"); do
        python -m pylint $file
done
```

Оказалось, что большинство исходных библиотек bash вообще не нужны для линтинга: необходима только возможность перебирать некоторые файлы и вызвать команду Python lint.

Многоразовый bash-скрипт внутри задачи

Лоренцо выполнил рефакторинг логики линтинга Python в один автономный bash-скрипт:

```
#!/usr/bin/env bash
set -xe
for file in $(find ${{ inputs.dir }} -name "*.py"); do
        python -m pylint $file
done
```

Но он не хочет передавать этот bash-скрипт напрямую в пайплайны CD службы транзакций и кредитных карт. Он хочет поместить его в задачу, которую можно написать один раз и ссылаться на нее во всех пайплайнах.

Поскольку для автоматизации CD онлайн-банка Purrfect Bank используется GitHub Actions, мы создадим задачу GitHub в репозитории CD-скриптов, которая будет функционировать как повторно используемая и выглядеть так:

```
name: Python Lint
description: Runs Python lint on the directory
inputs:
  dir:
    description: "The directory containing files to lint"
    default: "."
runs:
  using: "composite"
  steps:
  -shell bash
    run: |
      pip install pylint
      for file in $(find ${{ inputs.dir }} -name "*.py"); do
        python -m pylint $file
      done
```

Использование директории в качестве аргумента позволяет применять эту задачу независимо от того, где находятся файлы.

Такой синтаксис указывает GitHub Actions на то, что содержимое раздела run должно выполняться в оболочке bash.

Лоренцо необходимо установить Pylint, чтобы он был доступен.

Теперь скрипт Лоренцо встроен непосредственно в GitHub Actions.

Теперь оба пайплайна могут ссылаться на задачу GitHub непосредственно в репозитории скриптов CD, что позволяет им повторно использовать задачу следующим образом:

```
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Python lint
      uses: purrfectbank/cd/.github/actions/python-lint@v0.1.0
```

Определив линтинг Python как задачу GitHub, мы сделали его повторно используемой задачей, для которой можно выпускать релизы и присваивать версии.

Переход от bash к Python

Затем Лоренцо переходит к файлу e2e_tests.sh. Функции `build_image`, `run_e2e_tests` и `deploy_to_e2e_sut` достаточно просты, поэтому он создает для каждой из них многоразовую задачу GitHub. Но последняя bash-функция (`setup_e2e_sut`) сложна, так что он решает создать для нее версионированную, тестируемую библиотеку на Python и инструмент Python, который можно использовать для ее запуска.

Сначала он создает пустую библиотеку Python в репозитории CD под названием end-to-end («сквозная»). Он также настраивает автоматизацию для создания версионированных релизов, отправляемых во внутренний реестр артефактов компании (точно такую же автоматизацию разработчики применяют для любой из библиотек, используемых в коде на продакшен).

Внутри этой библиотеки он создает пакет под названием setup и инструмент командной строки под названием `purrfect-e2e-setup`, который вызывает эту библиотеку. Самое интересное, что он может добавить тесты для каждой функции, а также для инструмента в пакет setup! После всех этих обновлений структура репозитория CD-скриптов выглядит так:

```
.github/
  actions/
    config-lint.yaml
    markdown-lint.yaml
    python-lint.yaml
    unit-test.yaml
    coverage.yaml
    build-image.yaml
    run-e2e-tests.yaml
    deploy-to-e2e.yaml
e2e/
  setup/
    test/...
  purrfect-e2e-setup.py
  setup.py
  requirements.txt
  README.md
README.md
```

Каждый из них представляет собой задачу GitHub, на которую можно ссылаться из других рабочих процессов онлайн-банка Purrfect Bank, расположенного на GitHub.

Логика функции setup_e2e_sut теперь распределена по нескольким функциям в пакете setup, каждая из которых имеет юнит-тесты и управляется с помощью командной строки к файлу purrfect-e2e-setup.py, который, в свою очередь, определяет и проверяет аргументы командной строки.

ВАЖНО

shell-скрипты идеально подходят для объединения в пайплайн отдельных программ, но когда логика начинает разрастаться, лучше перейти на другой язык и/или механизм совместного использования.

Задачи как код

После того как Лоренцо закончил работу, задачи, используемые двумя командами подразделения платежей, выглядят так:

Каждая задача определяется в версионированной задаче (посредством GitHub Actions). Каждая задача содержит минимум bash и может повторно использоваться в нескольких пайплайнах. Пайплайны сервиса транзакций и сервиса интеграции кредитных карт используют все эти задачи.

Репозиторий CD содержит тестируемые библиотеки и инструмент командной строки, чтобы настраивать среду для сквозного тестирования. В каждом проекте подразделения платежей создается своя собственная задача, которая вызывает инструмент командной строки, потому что у каждого из них разные потребности (например, сервису интеграции кредитных карт требуется запускать фиктивные экземпляры нескольких провайдеров кредитных карт).

Пайплайны, определенные для двух сервисов, практически идентичны, и теперь они оба поддерживаются библиотеками и скриптами, которые обрабатываются как код: хранятся в системе управления версиями, тестируются, проверяются и версионируются.

Лоренцо смог решить все проблемы с пайплайнами, и команды больше не собираются отказываться от CI:

- ~~*Их трудно отлаживать.*~~ Каждый сбой теперь четко связан с конкретной задачей, bash используется по минимуму, а сложная логика поддерживается хорошо написанными инструментами и библиотеками, которые сообщают о значимых ошибках.
- ~~*Они непрозрачны.*~~ Объем bash в задачах минимален, при использовании библиотек и инструментов исходный код хорошо детализируется и подкрепляется тестами, что значительно облегчает понимание кода.
- ~~*Разработчики не решаются вносить в них изменения.*~~ Логика, реализованная в bash, не слишком сложна, а если возникнет необходимость ее усложнить, то ее легко преобразовать в библиотеки и инструменты Python. Существующие инструменты и библиотеки сопровождаются тестами, а проверять код стало намного проще.

Пайплайны, которые было *сложно использовать* и *поддерживать*, стали такими же простыми в использовании и поддержке, как и все остальное ПО.

Скрипты CD — это тоже код

Одна из причин, по которым подразделение платежей в Purrfect Bank оказалось в сложной ситуации, заключается в том, что разработчики не обрабатывали код CD так же, как и весь остальной код. Бизнес-логика, реализованная в сервисах транзакций и интеграции кредитных карт, соответствовала лучшим практикам: она была тщательно продумана, протестирована, проверена и версионирована. Но к коду пайплайнов и задач CD применялись другие стандарты. Разработчикам просто нужно было, чтобы он работал.

Как и с любым другим кодом, такой подход поначалу может быть довольно эффективным, но по мере увеличения сложности он начинает давать сбои, и возникают проблемы с поддержкой, что значительно замедляет работу. Вспомните определение CI:

> ***Постоянный процесс внесения изменений в код,* когда каждое изменение проверяется *при добавлении его к уже имеющимся и проверенным изменениям.***

В главе 3 было показано, как этот принцип применяется ко *всем текстовым данным, из которых состоит продукт*, в том числе к конфигурации. Чтобы в полной мере использовать принцип «конфигурация как код», осталось признать, что *код, определяющий способ интеграции и доставки программного продукта, сам по себе также является его частью.*

Лоренцо и команда Purrfect Bank усвоили важный урок:

- Бизнес-логика — это код.
- Тесты — это код.
- Конфигурация — это код.
- *Пайплайны, задачи и скрипты CD — это тоже код.*

ВАЖНО

Код — это код. Применяйте ко всему коду, который вы пишете, те же практики, которые вы применяете к бизнес-логике.

ВАЖНО

Применяйте принцип «как код» (например, конфигурация как код, пайплайны как код) ко всему коду, включая задачи, пайплайны и скрипты CD.

Заключение

Когда команды подразделения платежей онлайн-банка Purrfect Bank только начали создавать свои пайплайны CD, bash хорошо выполнял свои функции. Но в какой-то момент bash перестал удовлетворять всем потребностям, а команды все равно продолжали его использовать. Это привело к тому, что пайплайны CD стало сложно поддерживать и эксплуатировать, и возник вопрос, есть ли от них хоть какая-то польза.

Лоренцо удалось привести пайплайны CD в надлежащее состояние, разбив большие bash-скрипты на хорошо продуманные дискретные задачи и устранив их зависимость от больших разросшихся библиотек bash. Большая часть логики осталась в bash, но теперь она уже определялась грамотно спроектированными, версионированными задачами, а более сложная логика была преобразована в инструменты и библиотеки Python, поддерживаемые тестами.

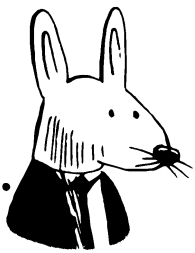
Итоги

- Задачи, которые выполняют слишком много, не могут генерировать хорошие сигналы.
- Грамотно спроектированные задачи похожи на правильно разработанные функции. Они обладают сильной связностью, слабой связанностью, имеют четко определенные интерфейсы и делают ровно столько, сколько нужно.
- Языки shell-скриптов, такие как bash, отлично подходят для связи ввода и вывода из нескольких программ, но языки общего назначения лучше справляются с более сложной логикой, обеспечивая удобство сопровождения.
- Код — это код. Применяйте ко всему коду, который вы пишете, те же практики, которые вы применяете к бизнес-логике.
- Код пайплайна CD является частью общего кода программного продукта.

Далее...

В последней главе будут рассмотрены различные способы организации задач в рамках пайплайна, а также функции системы CD, которые понадобятся для создания максимально эффективных пайплайнов.

13 Реализация пайплайнов



В этой главе

- ✓ Как добиться, чтобы необходимые действия выполнялись независимо от сбоев
- ✓ Ускорение пайплайнов за счет параллельного выполнения задач вместо последовательного
- ✓ Обеспечение повторного использования пайплайнов путем их параметризации
- ✓ Как оценивать компромиссы при принятии решения о том, сколько пайплайнов использовать и что поместить в каждый из них
- ✓ Как определить, какие функции необходимы в системе CD для представления графов пайплайнов

Добро пожаловать в последнюю главу книги «Грокаем Continuous Delivery». В этой главе мы рассмотрим общую структуру пайплайнов непрерывной доставки, а также особенности, на которые следует обращать внимание в системах CD, чтобы эффективно структурировать пайплайны.

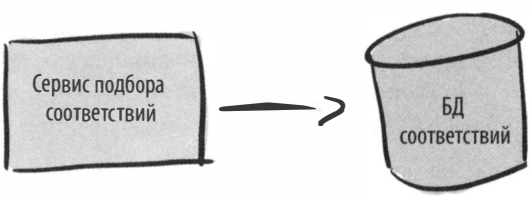
Большое внимание мы уделим системам CD, оптимизированным под повторное использование. Этот подход основывается на общем принципе «конфигурация как код», в частности на использовании передовых методик разработки при написании пайплайнов CD, как и при написании любого другого кода.

Проект PetMatch

Чтобы понять важность конкретной реализации пайплайна, рассмотрим пайплайны, используемые в компании PetMatch. Эта компания предоставляет услугу поиска и подбора питомца тем, кто хочет завести домашнее животное. Уникальный собственный алгоритм подбора соответствий помогает будущему владельцу подобрать самого подходящего питомца.

Компания выросла из стартапа и работает чуть более восьми лет. Все это время она уделяла большое внимание автоматизации CD, поэтому создала множество пайплайнов для различных сервисов.

Однако в последнее время разработчики сервиса подбора соответствий начали сомневаться в полезности использования пайплайнов. Архитектура самого сервиса относительно проста: вся бизнес-логика содержится в рабочем сервисе Python, который предоставляет REST API[1] остальной части стека PetMatch, и поддерживается базой данных:



Этот сервис имеет обширную автоматизацию и тестирование, реализованные в нескольких пайплайнах CD, но они выполняются медленно, и разработчики чувствуют, что не получают необходимую им информацию вовремя. Фактически автоматизация CD их подводит!

[1] REST API — это способ взаимодействия сайтов и веб-приложений с сервером. API (Application Programming Interface — программный интерфейс приложения) — это код, который позволяет двум приложениям обмениваться данными с сервера. REST (Representational State Transfer — передача состояния представления) — это способ создания API с помощью протокола HTTP. — *Примеч. пер.*

Пайплайны CD сервиса подбора соответствий

Сервис подбора соответствий имеет три отдельных пайплайна, и разработчики взаимодействуют с каждым из них совершенно по-разному:

- *Пайплайн CI* запускается при каждом PR перед слиянием.
- *Пайплайн сквозного тестирования* запускается один раз в сутки ночью.
- *Пайплайн выпуска релиза* запускается, когда команда готова выпустить очередной релиз (примерно один раз в несколько недель).

Команда в целом довольна пайплайном тестирования CI. Он выполняется для каждого PR менее чем за 5 минут и генерирует полезный сигнал.

Однако два других пайплайна не так хороши. В пайплайне сквозного тестирования есть несколько проблем:

- Поскольку он запускается ночью, сотрудники узнают о сбое только на следующий день. Приходится заниматься поиском причин сбоя, выяснять, кто виноват, а затем откатываться назад и исправлять то, что уже было внесено в общий код. Это напрягает.
- Выполнение тестов только раз в день означает, что код редко бывает готов к выпуску. И команда не может уверенно перейти к непрерывному развертыванию.
- Причина, по которой пайплайн выполняется ночью, — его продолжительность: полное выполнение этого пайплайна занимает более часа, и никто в команде не хочет ждать так долго во время итераций PR.
- Последняя проблема заключается в том, что сквозные тесты создают тестовую среду, но если тесты не завершаются успешно, они не подчищают за собой, а это означает, что со временем на тестирование расходуется все больше и больше ресурсов и кому-то приходится чистить все вручную.

Проблемы с пайплайном выпуска можно свести к одному пункту:

- Проблемы с сервисом и конфигурацией обычно выявляются этим пайплайном. Но к сожалению, он запускается, только когда команда уже приготовилась к выпуску релиза, поэтому выпуски часто прерываются и откладываются, пока разработчики спешно выискивают и устраняют обнаруженные проблемы.

Рассмотрим эти проблемы под несколько иным углом и попробуем найти общие закономерности в двух проблемных пайплайнах.

Проблемы пайплайнов CD

Можно ли выделить общие закономерности в этих двух пайплайнах? Взглянем на них еще раз. Это проблемы с пайплайном сквозного тестирования:

- Разработчики узнают о сбое только на следующий день после слияния.
- Код не готов к выпуску.
- Работает слишком медленно, чтобы запускать его при каждом PR.
- Не подчищает за собой, если происходит сбой.

Сигнал приходит с опозданием.

К этому приводит задержка сигнала об ошибке.

Низкая скорость — это проблема.

Это либо баг, либо ошибка в самом пайплайне.

А у пайплайна релизов только одна серьезная проблема:

- Обнаруживает проблемы в коде, которые были внесены давно.

Еще один случай, когда сигнал приходит с опозданием.

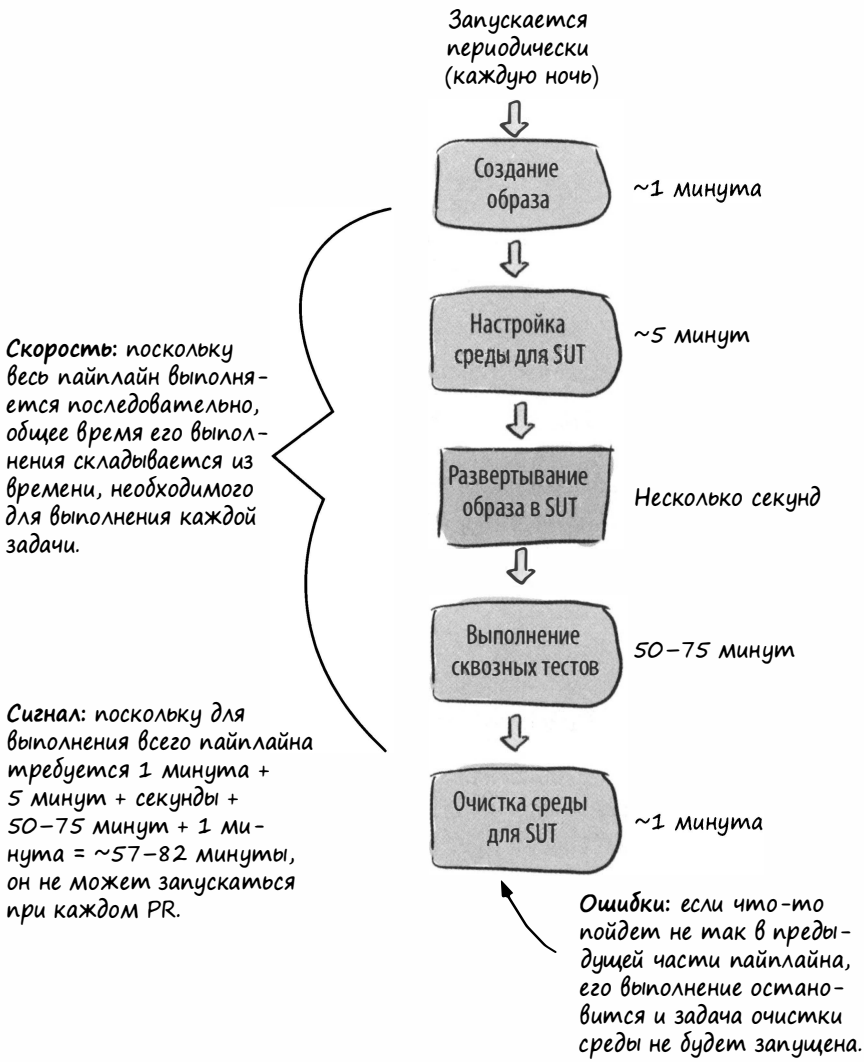
Эти проблемы можно разделить на три категории:

- *Ошибки* — когда пайплайны не делают того, что должны, например, сквозные тесты оставляют после себя тестовую среду в ненадлежащем состоянии.
- *Скорость* — низкая скорость выполнения пайплайна сквозного тестирования не позволяет команде запускать его, когда это необходимо.
- *Сигнал* — оба пайплайна по причине неподходящего времени их запуска генерируют сигналы слишком поздно, уже после того, как произошло слияние кода (иногда спустя довольно длительное время).

Далее мы рассмотрим, как решить каждую из этих проблем во всех пайплайнах сервиса подбора соответствий.

Пайплайн сквозного тестирования

Начнем с пайплайна сквозного тестирования сервиса подбора соответствий. У этого пайплайна существуют проблемы из всех трех категорий: *ошибки*, *скорость* и *сигнал*. Текущий пайплайн выглядит так:



Пайплайн сквозного тестирования и ошибки

Первая проблема, которую необходимо решить, связана с *ошибками* в пайплайне сквозного тестирования, а именно с тем, что задача очистки среды для тестирования запускается только в том случае, если остальная часть пайплайна прошла успешно. Большинство команд в компании PetMatch используют GitHub Actions, и вот как выглядит рабочий процесс GitHub Actions для сервиса подбора соответствий:

```
name: Run System tests
on:
  schedule:
  - cron: '0 23 * * *'
jobs:
  build-image: ...
  setup-sut:
    needs: build-image
    outputs:
      env-ip: ${{ steps.provision.outputs.env-ip }}
    ...
  deploy-image-sut:
    needs: setup-sut
    ...
  end-to-end-tests:
    needs: [setup-sut, deploy-image-sut]
    env:
      SUT_IP: ${{ needs.setup-sut.outputs.env-ip}}
    ...
  clean-up-sut:
    needs: [setup-sut, end-to-end-tests]
    ...
```

Рабочий процесс запускается каждый вечер в 23:00 UTC (Universal Coordinated Time — Всемирное координированное время).

Ключевое слово needs обеспечивает очередность выполнения каждого задания и позволяет последующим заданиям использовать выходные данные предыдущих заданий, от которых они зависят.

В большинстве заданий требуется выполнить setup-sut, поскольку им необходимо использовать IP-адрес среды, предоставляемый этим заданием.

Поскольку задание clean-up-sut запускается после каждого второго задания, сбой любого из предшествующих ему заданий означает, что оно не запустится.

ВАЖНО

Позволяя задачам выпускать выходные данные, которые другие задачи пайплайна могут использовать в качестве входных, можно разрабатывать задачи как отдельные юниты с сильной связностью, слабой связанностью и возможностью повторного использования в разных пайплайнах.

Словарик

GitHub Actions использует *рабочие процессы* (workflows) для того, что в этой книге называется *пайплайнами*. *Задачи* (tasks) в рамках этой книги условно эквивалентны *заданиям* (jobs) и *действиям* (actions). Задачи можно повторно использовать и ссылаться на них во всех рабочих процессах, в то время как задания определяются внутри рабочих процессов и не могут быть использованы повторно.

Finally-логика

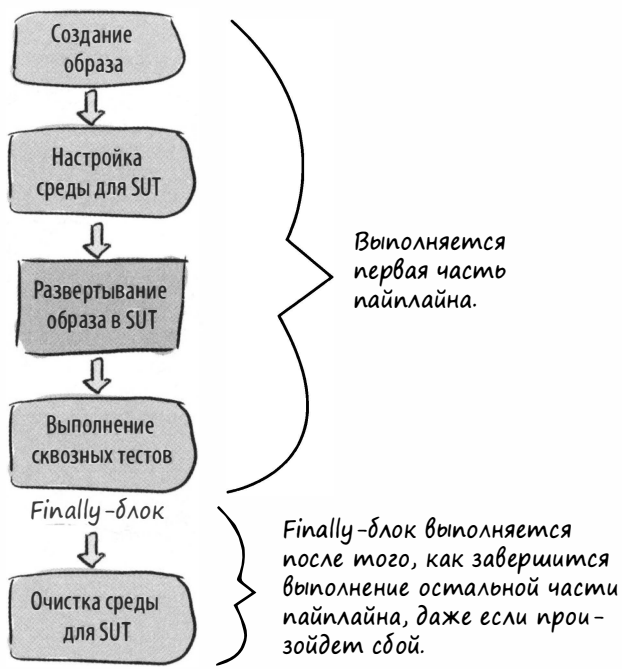
Функциональность, которая необходима команде сервиса подбора соответствий внутри пайплайна, предполагает, что задача очистки тестовой среды должна выполняться независимо от того, что происходит с остальной частью пайплайна. Во многих пайплайнах определенные задачи должны выполняться, даже если другие части пайплайна дают сбой. Это похоже на концепцию *finally-логики* в языках программирования, например инструкцию `finally` в Python:

> Точнее говоря, процесс очистки среды должен происходить, если выполняется задача настройки среды для SUT, но пока для простоты будем считать, что задача очистки среды выполняется независимо от этого. Для изменения процесса очистки среды так, чтобы он выполнялся только в том случае, когда выполняется задача настройки среды, нужно использовать условные функции, которые мы рассмотрим немного позже.

```
try:
    print("hello world!")
    raise Exception("oh no")
finally:
    print("goodbye world")
```

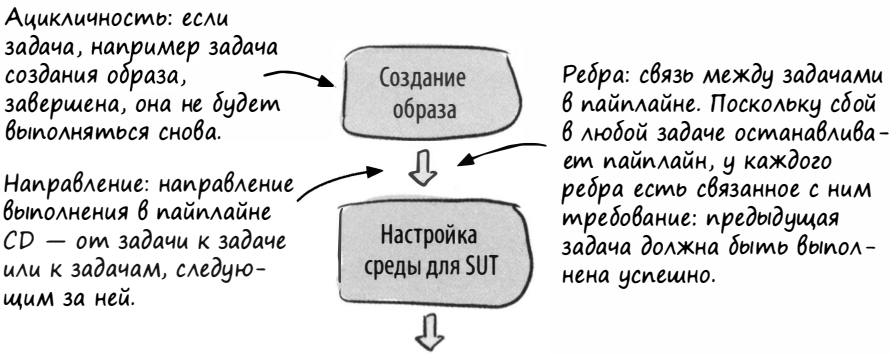
Код в инструкции finally будет выполнен, даже если возникло исключение.

Включение finally-логики в пайплайн означает, что он будет выполняться в два этапа. Сначала выполняется основная часть пайплайна, а после ее завершения (неважно, успешного или неудачного) выполняется finally-часть пайплайна:

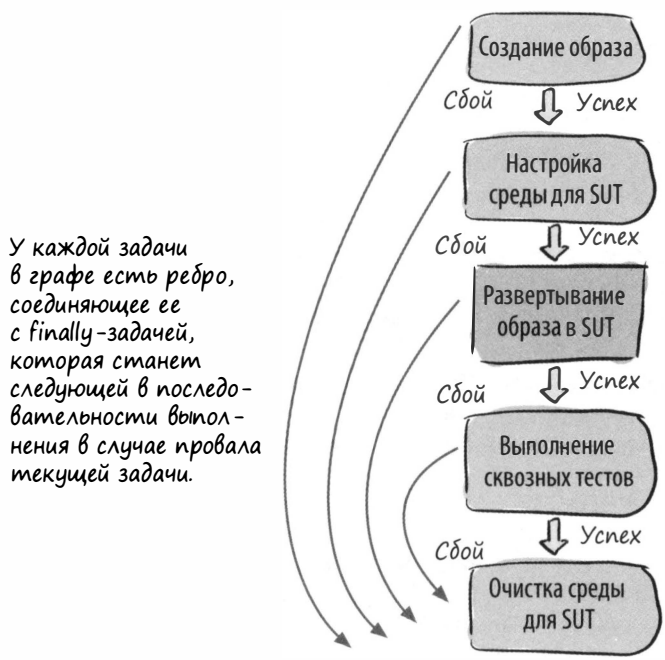


Концепция Finally в виде графа

Еще один полезный способ анализировать пайплайны, особенно когда вы начинаете выполнять задачи параллельно, — использовать *ориентированный ациклический граф* (DAG, directed acyclic graph). DAG — это граф, в котором ребра имеют направление (в нашем случае — от одной задачи к другой) и нет циклов (в нем невозможно, начиная с одной задачи и следуя по ребрам, вернуться к той же задаче).



Если рассматривать пайплайн как DAG, то включение finally-логики подобно созданию ребер от каждой задачи пайплайна к задачам блока finally, которые выполняются при сбое. Если какая-то задача завершается неудачей, сразу начинается выполнение finally-задач:



Принцип Finally в пайплайне сервиса подбора соответствий

Команде разработчиков сервиса подбора соответствий необходимо представить finally-функциональность с помощью синтаксиса выражений для GitHub Actions. В этом синтаксисе, чтобы отметить задание как требующее выполнения независимо от статуса остальных заданий в рабочем процессе, используется `if: ${{ always() }}`. Вот как выглядит обновленный пайплайн сервиса подбора соответствий:

```
name: Run System tests
on:
  schedule:
  - cron: '0 23 * * *'
jobs:
  build-image: ...
  setup-sut:
    needs: build-image
    outputs:
      env-ip: ${{ steps.provision.outputs.env-ip }}
    ...
  deploy-image-sut:
    needs: setup-sut
    ...
  end-to-end-tests:
    needs: [setup-sut, deploy-image-sut]
    env:
      SUT_IP: ${{ needs.setup-sut.outputs.env-ip}}
  ...
  clean-up-sut:
    if: ${{ always() }}
    needs: [setup-sut, end-to-end-tests]
    ...
```

Инструкция `if` указывает GitHub Actions всегда выполнять это задание. Если остальные задания завершатся успешно, GitHub Actions примет во внимание инструкции needs в следующей строке и инструкция `if` выполнится после end-to-end-tests, но если это или любое другое задание завершится неудачей, следующим сразу выполнится clean-up-sut.

Ошибка в пайплайне CD сервиса подбора соответствий исправлена: теперь очистка среды тестирования будет происходить в любом случае.

ВАЖНО

Синтаксис пайплайна, предоставляемый системой CD, должен позволять определять finally-логику в пайплайнах, то есть создавать задачи в пайплайне, которые должны выполняться всегда, даже если другие части пайплайна не смогут успешно выполниться.

ВАЖНО

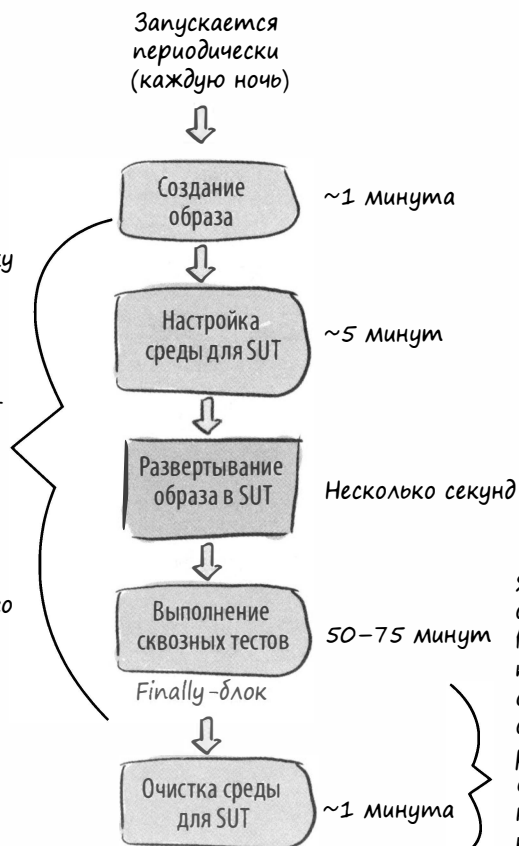
Наличие поддержки условного выполнения (которое в GitHub Actions реализовано с помощью инструкции `if`) позволяет использовать более гибкие пайплайны. Например, можно запускать пайплайн для каждого PR, но включать в него в том числе задачи, которые выполняются только после слияния.

Пайплайн сквозного тестирования и скорость

Команда сервиса подбора соответствий исправила ошибку в пайплайне CD, но проблемы со скоростью и сигналами остались:

- ~~*Ошибки* — сквозные тесты оставляют после себя тестовую среду в ненадлежащем состоянии.~~ Исправлено!
- *Скорость* — пайплайн сквозного тестирования слишком медленный, чтобы запускать его, когда это необходимо.
- *Сигнал* — пайплайны сквозного тестирования и выпуска релизов генерируют сигналы слишком поздно.

Команда переходит к проблеме скорости, решение которой также частично решит проблему сигналов, потому что если пайплайн сможет работать быстрее, его можно будет запускать чаще.

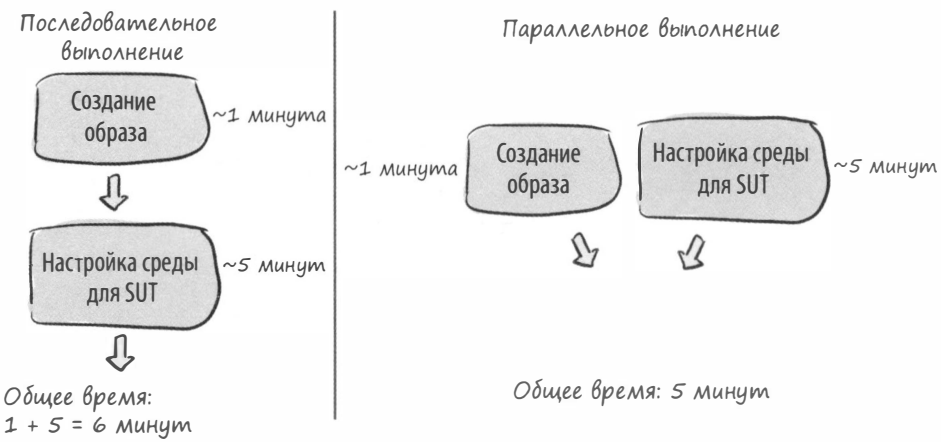


Параллельное выполнение задач

Команда сервиса подбора соответствий замечает, что хотя все задачи в пайплайне выполняются одна за другой, некоторые из них фактически не зависят от других. Анализируя задачи, которые не связаны с finally-логикой, они выявили следующие связи между задачами:

- *Сквозные тесты* — требуют развертывания образа в SUT.
- *Развертывание образа в SUT* — требует создания образа и настройки среды SUT.
- *Настройка среды SUT* — ничего не требует.
- *Создание образа* — ничего не требует.

Поскольку создание образа и настройка среды SUT не требуют выполнения других задач, их не нужно запускать одну за другой: они могут выполняться *параллельно* (то есть *одновременно*). Параллельное выполнение задач — это способ ускорить выполнение пайплайнов. Если две задачи выполняются одна следом за другой (то есть *последовательно*), общее время их выполнения равно сумме времени, необходимого для выполнения каждой из них. Если же они выполняются параллельно, время выполнения обеих задач будет ограничено только продолжительностью выполнения самой медленной из них.

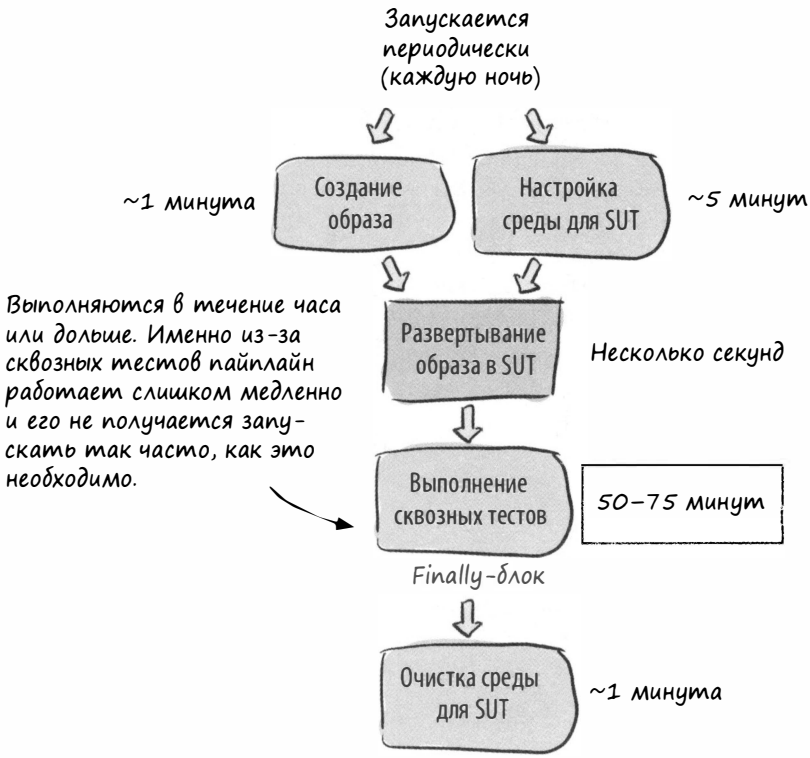


ВАЖНО

Параллельное выполнение задач — это способ сократить время выполнения пайплайнов. Чтобы определить, какие задачи могут выполняться параллельно, изучите зависимости между ними. При сравнении систем CD обращайте внимание на системы, которые позволяют выполнять задачи параллельно.

Пайплайн сквозного тестирования и скорость выполнения тестов

Несмотря на то что пайплайн CD сервиса подбора соответствий теперь использует возможности параллельного выполнения, общее время изменилось довольно незначительно. За счет одновременного выполнения задач создания образа и настройки среды для SUT удалось сэкономить всего 1 минуту.



Источник проблем со скоростью этого пайплайна — сами сквозные тесты. Никакие изменения в остальной части пайплайна не помогут: общее время выполнения всех остальных задач составляет ~6 минут, а одни только тесты занимают более часа.

Параллельное выполнение и шардинг тестов

Начинать решение любых проблем с медленными тестами стоит с оценки самих тестов, чтобы определить, какие фундаментальные первопричины лежат в основе этих проблем. Чтобы тесты оставались работоспособными в течение долгого времени и их было удобно сопровождать, необходимо устранить именно эти первопричины.

Команда сервиса подбора соответствий уверена, что уже сделала это (никаких фундаментальных проблем в наборе тестов не выявлено), и поэтому переходит к сле-

дующему этапу — шардингу тестов для параллельного запуска частей общего набора сквозных тестов.

Набор сквозных тестов состоит из 50 отдельных тестов, и на выполнение каждого уходит от 60 до 90 секунд. Полное время выполнения набора сквозных тестов таково:

> Подробнее об ускорении медленных тестов, в том числе с помощью шардинга, см. в главе 6.

50 тестов × 60–90 секунд на тест = 50–75 минут.

Чтобы ускорить работу пайплайна сквозных тестов и иметь возможность запускать его при каждом PR, выполнение тестов должно занимать не более 15 минут:

15 минут / (60–90 секунд на тест) = 10–15 тестов на шард

За 15-минутное окно, которое команда определила, можно выполнить минимум 10 тестов:

50 тестов / 10 тестов на шард = 5 шардов

Команда решает проводить тесты на 6 шардах, чтобы оставить немного места для маневра:

50 тестов / 6 шардов = 8,3 теста на шард ~= округляем до 9 тестов на шард

9 × 60–90 секунд = 9–13,5 минуты на шард

При выполнении тестов на 6 шардах общее время выполнения будет равно времени, которое потребуется самому долгому шарду. В худшем случае оно составит 13,5 минуты.

Чтобы обновить задачу сквозного тестирования и распределить тесты по 6 шардам, команда применяет тот же подход, что и Шридхар в главе 6, а именно использует библиотеки pytest-shard в Python для работы с шардингом совместно с функциональностью `matrix` из GitHub Actions:

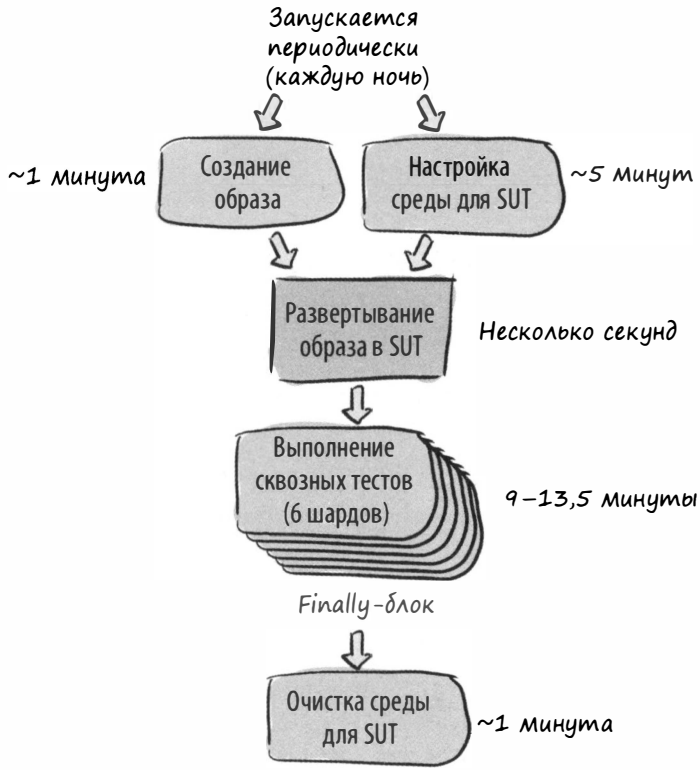
```
end-to-end-tests:
  needs: [setup-sut, deploy-image-sut]
  runs-on: ubuntu-latest
  env:
    SUT_IP: ${{ needs.setup-sut.outputs.env-ip}}
  strategy:
    fail-fast: false
    matrix:
      total_shards: [6]
      shard_indexes: [0, 1, 2, 3, 4, 5]
  steps:
    ...
    - name: Install pytest-shard
      run: |
        pip install pytest-shard==0.1.2
    - name: Run tests
      run: |
        pytest \
          --shard-id=${{ matrix.shard_indexes }} \
          --num-shards=${{ matrix.total_shards }}
```

Использование стратегии матрицы указывает GitHub Actions на то, что следует выполнять задание один раз для каждой комбинации элементов в total_shards и shard_indexes, что в данном случае составляет шесть комбинаций, по одной для каждого шарда.

Каждый экземпляр тестового задания, запускаемый GitHub Actions, получит разные комбинации значений для matrix.shard_indexes и matrix.total_shards: [6, 0], [6, 1], [6, 2], [6, 3], [6, 4], [6, 5].

Пайплайн сквозного тестирования с шардингом

После того как тесты были разделены на шарды, пайплайн сквозного тестирования стал выглядеть иначе:



Общее время: 5 минут + несколько секунд + 9–13,5 минуты + 1 минута = 15–19,5 минут

Общее время выполнения пайплайна сквозного тестирования составляет максимум 13,5 минуты. Тогда весь пайплайн будет выполнен менее чем за 20 минут, а возможно, даже и за 15 минут. Этого достаточно, чтобы его можно было запускать при каждом PR. Если разработчики захотят сделать его еще быстрее, они могут добавить больше шардов для сквозных тестов.

ВАЖНО

Благодаря использованию стратегии матрицы легко распараллелить выполнение задач в пайплайне (например, при шардинге тестов), что позволяет эффективно снизить общее время выполнения пайплайна и одновременно упростить повторное использование задач.

Пайплайн сквозного тестирования и сигналы

Решив проблемы со скоростью, команда переходит к проблеме сигналов:

- ~~*Ошибки* — сквозные тесты оставляют после себя тестовую среду в ненадлежащем состоянии.~~ Исправлено!
- ~~*Скорость* — пайплайн сквозного тестирования слишком медленный, чтобы запускать его, когда это необходимо.~~ Исправлено!
- *Сигнал* — пайплайны сквозного тестирования и выпуска релизов генерируют сигналы слишком поздно.

Сигналы от пайплайнов сквозного тестирования и выпуска релизов поступают слишком поздно. Пайплайн сквозного тестирования раньше запускался только ночью, поскольку был слишком медленным. Но теперь, когда весь пайплайн выполняется менее чем за 20 минут, разработчики могут запускать его при каждом PR и сразу же получать сигнал!

Они уже запускают пайплайн CI для каждого PR. Добавить пайплайн сквозного тестирования можно двумя способами:

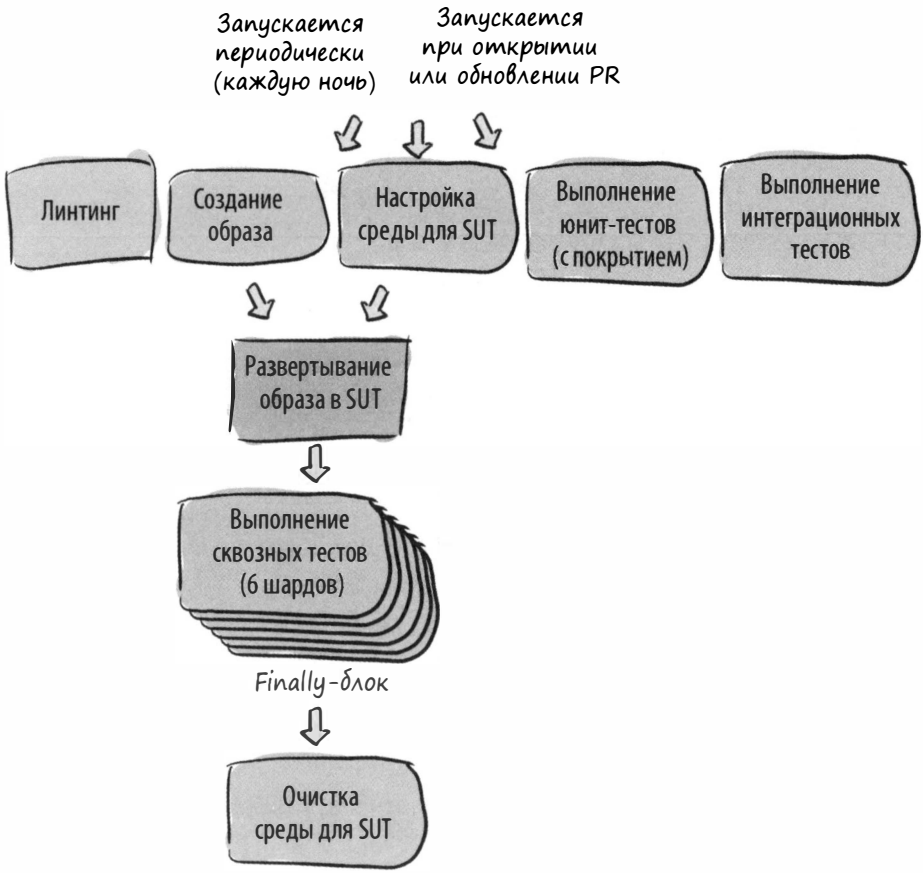
- запускать пайплайн сквозного тестирования после пайплайна CI;
- запускать пайплайн сквозного тестирования параллельно с пайплайном CI.

Чтобы определиться с выбором, необходимо рассмотреть несколько факторов:

- *Использование ресурсов* — пайплайн сквозного тестирования потребляет ресурсы для запуска SUT. Если пайплайн CI завершится с ошибкой, стоит ли задействовать эти ресурсы (для параллельного запуска второго пайплпйна) или лучше остаться консервативными и использовать их, только если интеграционные и юнит-тесты в пайплайне пройдут успешно (то есть будут запускать один пайплайн после другого)?
- *Получение полной информации о сбоях* — если сквозные тесты запускаются только после прохождения пайплайна CI, то автор PR может разозлиться, потратив время на исправление тестов CI, а потом обнаружив, что после их исправления сквозные тесты тоже не работают.

Единый пайплайн CI

Команда сервиса подбора соответствий решает запускать сквозные тесты параллельно с существующим пайплайном CI, чтобы оптимизировать процесс и получать как можно больше сигналов как можно быстрее, даже если это означает выделение дополнительных ресурсов на запуск сред для SUT. Вместо использования двух отдельных пайплайнов они объединяют два существующих в один более крупный пайплайн CI, который выполняет все три вида тестов (и линтинг тоже!):



Как только пайплайн запускается, он начинает параллельно выполнять линтинг, создание образа, настройку среды для SUT, интеграционные и юнит-тесты.

Пайплайн выпуска релизов и сигналы

Поскольку пайплайн сквозного тестирования теперь объединен с пайплайном CI и запускается при каждом PR, проблема с сигналами в этом пайплайне решена.

- ~~*Ошибки* — сквозные тесты оставляют после себя тестовую среду в ненадлежащем состоянии.~~ Исправлено!
- ~~*Скорость* — пайплайн сквозного тестирования слишком медленный, чтобы запускать его, когда это необходимо.~~ Исправлено!
- *Сигнал* — пайплайны ~~сквозного тестирования и~~ выпуска релизов генерируют сигналы слишком поздно.

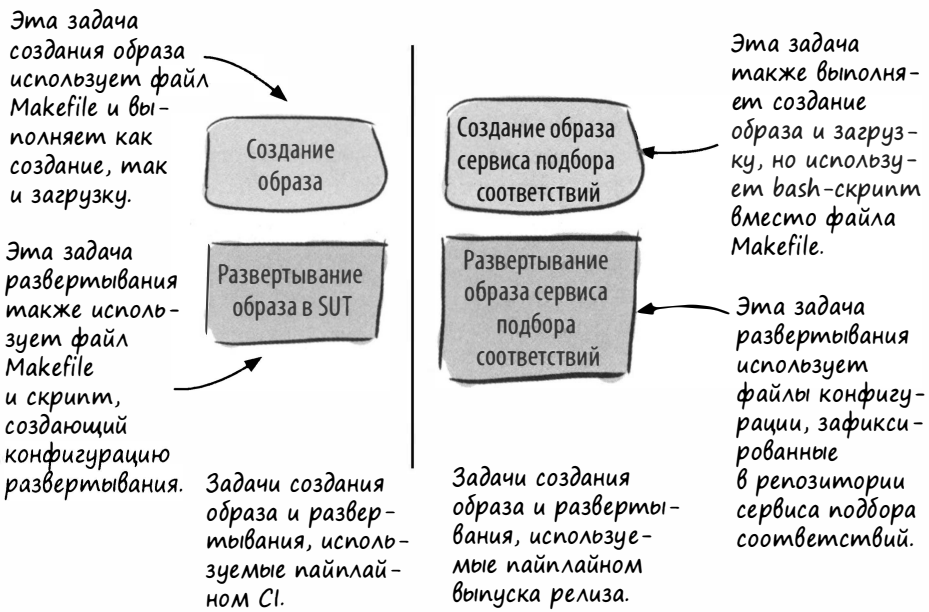
В пайплайне выпуска релизов осталась проблема с сигналами. Он запускается, только когда происходит развертывание (а это в лучшем случае один раз в несколько недель).

Этот пайплайн часто выявляет проблемы, пропущенные ранее. Например, вот какие проблемы он недавно обнаружил:

- Внесенное изменение требовало, чтобы переменная среды устанавливалась во время сборки, но это изменение было внесено в файл Makefile, используемый сквозными тестами, и не было внесено в задачу сборки, описанную ранее. При запуске пайплайна выпуска эта задача не сработала.
- Параметр командной строки в сервисе подбора соответствий был изменен, но конфигурация, используемая для развертывания сервиса, не была обновлена. Когда задача развертывания начала выполняться, она попыталась использовать конфигурацию со старым параметром, и развертывание завершилось сбоем.

Различия в пайплайнах CI и выпуска релизов

Почему пайплайн выпуска релизов находит проблемы, а пайплайн CI — нет? В конце концов, для выполнения части пайплайна CI, связанной со сквозным тестированием, тоже требуется сборка и развертывание. Причина, по которой CI не может отловить некоторые проблемы, заключается в том, что логика сборки и развертывания в CI отличается от логики этих процессов в пайплайне выпуска.



Как мы видели в главе 7 на примере проекта CoinExCompare, использование различной логики при сборке и развертывании в CI и на продакшен приводит к возникновению багов. Решение — обновить пайплайн CI так, чтобы он мог использовать пайплайн выпуска в части сборки и развертывания. Если использовать один пайплайн (выполняющий одни и те же задачи), вероятность обнаружения любых потенциальных проблем будет значительно выше.

А что, если использовать непрерывное развертывание, чтобы получать сигналы раньше?

Хорошая мысль! Если бы компания PetMatch использовала непрерывное развертывание, то развертывание сервиса подбора соответствий осуществлялось бы при каждом изменении и все проблемы выявлялись немедленно. Однако проблемы все равно будут обнаруживаться только *после* слияния, и тогда код будет оставаться недоступным для выпуска, пока ошибки не будут исправлены. Старайтесь находить ошибки до того, как они попадут в коммит основного кода.

Комбинация пайплайнов

Команда сервиса подбора соответствий намерена использовать существующий пайплайн выпуска релизов в двух сценариях:

- Когда необходимо выполнить развертывание (так они используют его сейчас).
- Для создания и развертывания образа в SUT в пайплайне CI.

Для второго варианта необходимо повторно использовать существующий пайплайн в рамках пайплайна CI. Этого можно добиться двумя способами:

- *Дублировать пайплайн* — использовать в пайплайне CI те же задачи, которые используются в пайплайне выпуска.
- *Вызывать пайплайн выпуска из пайплайна CI* — использовать пайплайн выпуска как есть из пайплайна CI, то есть вызывать один пайплайн из другого.

Оба варианта решат интересующую нас проблему, поскольку в обоих случаях пайплайн CI будет использовать те же задачи, что и пайплайн выпуска. Но у первого варианта есть недостатки:

- Если пайплайн выпуска будет меняться, то автор изменений должен будет не забыть обновить и пайплайн CI, иначе они рассинхронизируются.
- Скорее всего, со временем способы определения и использования задач в каждом из пайплайнов начнут разниться, в отличие от варианта, когда в обоих случаях используется один и тот же пайплайн.

Команда разработчиков сервиса подбора соответствий решила, что лучше вызывать пайплайн выпуска релизов из пайплайна CI целиком.

ВАЖНО

Системы CD с функцией вызова одного пайплайна из другого лучше подходят для повторного использования (и в них меньше ненужного дублирования и ошибкоопасной поддержки), чем системы, обеспечивающие повторное использование только на уровне задач (или вообще его не поддерживающие).

Пайплайн выпуска релизов

Чтобы использовать пайплайн выпуска релизов из пайплайна CI, команде сервиса подбора соответствий необходимо внести несколько изменений. Сейчас пайплайн выпуска определен как рабочий процесс GitHub:

```
name: Build and deploy to production from tag
on:
  push:
    tags:
      - '*'
jobs:
  build-matchmaking-service-image:
    runs-on: ubuntu-latest
    outputs:
      built-image: ${{ steps.build-and-push.outputs.built-image }}
    steps:
      - uses: actions/checkout@v2
      - id: build-and-push
        run: |
          IMAGE_REGISTRY="10.10.10.10"
          IMAGE_NAME="petmatch/matchmaking"
          VERSION=$(echo ${{ github.ref }} | cut -d / -f 3)
          IMAGE_URL="$IMAGE_REGISTRY/$IMAGE_NAME:$VERSION"
          BUILT_IMAGE=$(./build.sh $IMAGE_URL)
          echo "::set-output name=built-image::${BUILT_IMAGE}"
  deploy-matchmaking-service-image:
    runs-on: ubuntu-latest
    needs: build-matchmaking-service-image
    steps:
      - uses: actions/checkout@v2
      - id: deploy
        run: |
          ./update_config.sh ${{ needs.build-matchmaking-service-image.outputs.built-image }}
          ./deploy.sh
```

Это задание определяет выходные данные — полный URL и хеш-сумму созданного образа. Выходные данные генерируются на следующем шаге и могут быть использованы в качестве входных данных для следующего задания

Этот шаг с идентификатором build-and-push создает определенные ранее выходные данные.

На этом этапе задание по созданию образа устанавливает значение выходных данных built-image.

Это задание может использовать полный URL и хеш-сумму созданного образа (выходные данные предыдущего задания) в качестве входных данных.

Первое задание в пайплайне определяет выходные данные с именем `built-image`, которые следующее задание использует в качестве входных. Это обеспечивает гибкость развертывания, достаточную, чтобы работать с любым URL созданного образа, и позволяет разделить логику на два отдельных хорошо отлаженных задания.

> ***Скрипты или встроенный bash***
>
> В главе 12 для удобства повторного использования рекомендовалось писать bash, встроенный в задачи, а не хранить его отдельно в скриптах (а также переходить на языки общего назначения, если код на bash становится слишком длинным). В этих примерах используется отдельный скрипт, чтобы не вдаваться в подробности реализации каждой задачи и сосредоточиться на особенностях пайплайна и функциях задач, о которых идет речь в этой главе.

Хардкодинг в пайплайне выпуска релизов

Несмотря на то что существующий пайплайн хорошо использует входные и выходные данные на уровне задач, на уровне самого пайплайна дела обстоят не лучшим образом. На самом деле в пайплайне выпуска жестко закодированы несколько параметров, которые не позволяют использовать его для системных тестов:

```
name: Build and deploy to production from tag
on:
  push:
    tags:
      - '*'
jobs:
  build-matchmaking-service-image:
    runs-on: ubuntu-latest
    outputs:
      built-image: ${{ steps.build-and-push.outputs.built-image }}
    steps:
      - uses: actions/checkout@v2
      - id: build-and-push
        run: |
          IMAGE_REGISTRY="10.10.10.10"
          IMAGE_NAME="petmatch/matchmaking"
          VERSION=$(echo ${{ github.ref }} | cut -d / -f 3)
          IMAGE_URL="$IMAGE_REGISTRY/$IMAGE_NAME:$VERSION"
          BUILT_IMAGE=$(./build.sh $IMAGE_URL)
          echo $BUILT_IMAGE
          echo "::set-output name=built-image::${BUILT_IMAGE}"
  deploy-matchmaking-service-image:
    runs-on: ubuntu-latest
    needs: build-matchmaking-service-image
    steps:
      - uses: actions/checkout@v2
      - id: deploy
        run: |
          ./update_config.sh ${{ needs.build-matchmaking-service-image.outputs.built-image }}
          ./deploy.sh
```

Реестр, имя образа и способ определения версии заданы жестко. Для сквозных тестов нужно отправлять образы в другой реестр, называть их иначе и использовать другую схему определения версий.

Скрипт deploy.sh жестко запрограммирован на развертывание сервиса подбора соответствий в рабочей среде, однако для сквозных тестов нужно производить развертывание в среде SUT.

Чтобы использовать этот пайплайн из пайплайна сквозного тестирования, необходимо изменить жестко заданные параметры следующим образом:

- Отправка должна производиться либо в реестр рабочих образов, либо в реестр, используемый для системных тестов.
- Должна быть возможность изменять способ определения версии и имени образа. В продакшене имя всегда одно и то же, а версия определяется из тега. Для сквозных тестов тег отсутствует, а имя образа другое.
- Развертывание должно производиться либо в рабочей среде, либо в среде SUT, созданной пайплайном CI.

Повторное использование пайплайна с помощью параметризации

Чтобы использовать пайплайн выпуска релизов из пайплайна CI, команда сервиса подбора соответствий вносит несколько изменений. Она создает повторно используемый рабочий процесс GitHub Actions для развертывания, который получает параметры извне, что позволяет использовать его как для сквозных тестов, так и для итогового развертывания:

```
name: Build and deploy
on:
  workflow_call:
    inputs:
      image-registry:
        required: true
        type: string
      image-name:
        required: true
        type: string
      version-from-tag:
        required: true
        type: boolean
      deploy-target:
        required: true
        type: string
jobs:
  build-matchmaking-service-image:
    runs-on: ubuntu-latest
    outputs:
      built-image: ${{ steps.build-and-push.outputs.built-image }}
    steps:
      - uses: actions/checkout@v2
      - id: build-and-push
        run: |
          if [ "${{ inputs.version-from-tag }}" = "true" ]
          then
            VERSION=$(echo ${{ github.ref }} | cut -d / -f 3)
          else
            VERSION=${{ github.sha }}
          fi
            IMAGE_URL="${{ inputs.image-registry }}/${{ inputs.image-name }}:$VERSION"
            BUILT_IMAGE=$(./build.sh $IMAGE_URL)
            echo "::set-output name=built-image::${BUILT_IMAGE}"
  deploy-matchmaking-service-image:
    runs-on: ubuntu-latest
    needs: build-matchmaking-service-image
    steps:
      - uses: actions/checkout@v2
      - id: deploy
        run: |
          ./update_config.sh ${{ ... }}
          ./deploy.sh ${{ inputs.deploy-target }}
```

Этот рабочий процесс настроен на запуск по workflow_call, это означает, что его можно вызывать из других рабочих процессов.

Эти параметры должны быть предоставлены вызывающим рабочим процессом. Они позволяют использовать данный рабочий процесс как для тестирования, так и для развертывания на продакшен.

Рабочий процесс может быть настроен на то, чтобы либо генерировать версию на основе тега (извлекаемого из github.ref), либо использовать коммит (предоставляемый через github.sha). В рабочих сборках будет использоваться тег, а в тестах — коммит.

Реестр образов и имя образа теперь можно настраивать.

Задаче развертывания теперь будет передаваться IP-адрес среды для развертывания, в отличие от того, что ранее было жестко задано всегда осуществлять развертывание на продакшен.

> ***А где секреты?***
>
> Для развертывания в среде SUT, скорее всего, потребуются другие учетные данные, чем для развертывания на продакшен. Эти детали в рассматриваемом примере опущены, но обеспечить возможность их параметризации тоже важно.

Применение повторно используемых пайплайнов

Пайплайн повторного использования (хранящийся в репозитории сервиса подбора соответствий в файле с названием .github/workflows/deployment.yaml) предназначен для вызова только из других рабочих процессов, поэтому команда сервиса подбора соответствий обновляет существующий рабочий процесс (настроенный на запуск при появлении тега новой версии), чтобы его использовать:

```
name: Build and deploy to production from tag
on:
  push:
    tags:
      - '*'
jobs:
  deploy:
    uses: ./.github/workflows/deployment.yaml
    with:
      image-registry: '10.10.10.10'
      image-name: 'petmatch/matchmaking'
      version-from-tag: true
      deploy-target: '10.11.11.11'
```

Когда появится тег новой версии, рабочий процесс развертывания будет вызываться с использованием параметров, которые гарантируют, что рабочий образ будет создан с номером версии из тега, помещен в реестр рабочих образов и развернут на рабочем экземпляре.

Рабочий процесс CI также обновлен для применения повторно используемого рабочего процесса развертывания. Предыдущие задачи сборки и развертывания (специфичные для сквозных тестов) были удалены и заменены вызовом многоразового рабочего процесса:

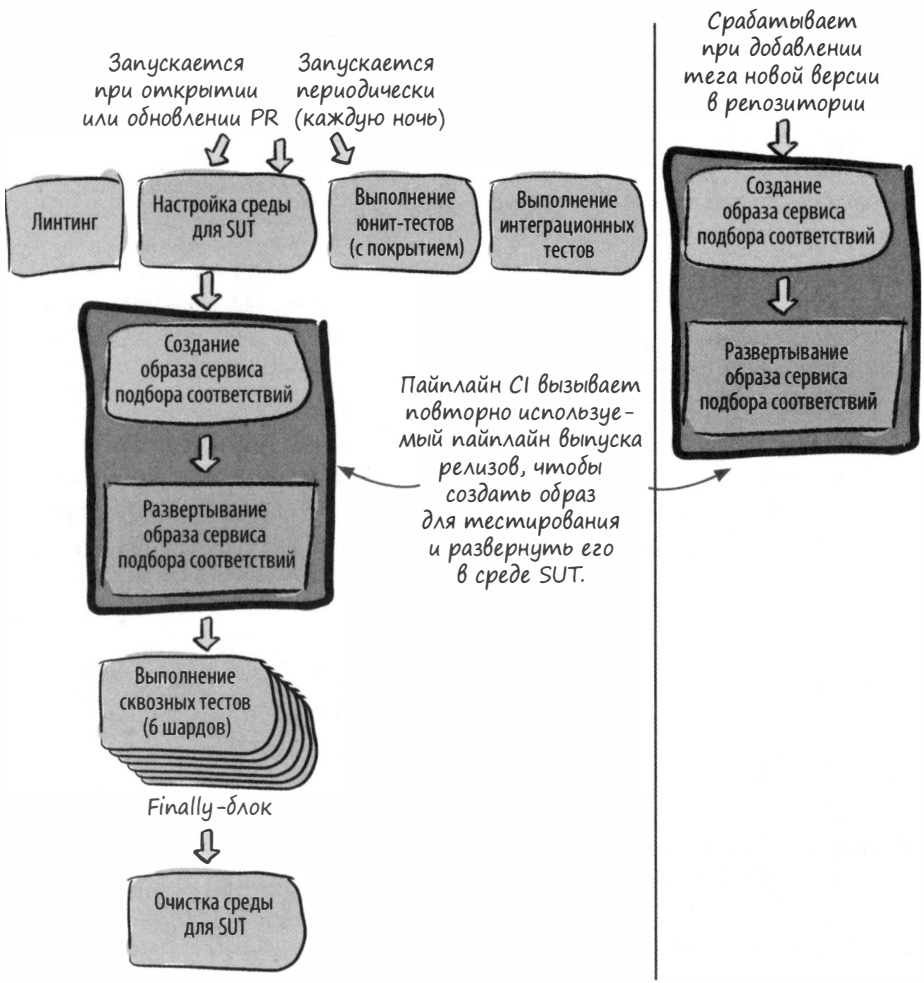
Когда запускается PR, рабочий процесс развертывания будет вызываться с использованием параметров, которые гарантируют, что тестовый образ будет создан с ожидаемым именем, версионирован в соответствии с коммитом, помещен в реестр тестового образа и развернут в среде SUT.

```
name: Run CI
..
jobs:
  setup-sut: ...
  deploy-to-sut:
    needs: setup-sut
    uses: ./.github/workflows/deployment.yaml
    with:
     image-registry: '10.12.12.12'
     image-name: 'petmatch/matchmaking-e2e-test'
     version-from-tag: false
     deploy-target: ${{ needs.setup-sut.outputs.env-ip}}
  end-to-end-tests: ...
  clean-up-sut: ...
```

setup-sut создает среду SUT для развертывания и предоставляет IP в качестве выходных данных. Этот IP можно передать в повторно используемый рабочий процесс в качестве входных данных.

Обновленные пайплайны

Пайплайн CI был обновлен с учетом нового повторно используемого пайплайна выпуска релизов (точно такой же пайплайн используется теперь и для развертывания на продакшен, но вызывается с другими параметрами) и теперь выглядит так:



ВАЖНО

Чтобы пайплайны можно было использовать в различных сценариях (например, при вызове другими пайплайнами), необходимо обеспечить возможность их параметризации.

Решение проблем непрерывной доставки в проекте PetMatch

Реализовав использование пайплайна выпуска релизов как часть сквозных тестов для каждого PR, разработчики сервиса подбора соответствий вернулись к списку проблем пайплайна CD, которые они должны были решить:

- ~~*Ошибки* — сквозные тесты оставляют после себя тестовую среду в ненадлежащем состоянии.~~ Исправлено! Использование finally-логики означает, что очистка среды после тестов будет происходить всегда.
- ~~*Скорость* — пайплайн сквозного тестирования слишком медленный, чтобы запускать его, когда это необходимо.~~ Исправлено! Благодаря использованию шардинга тестов и стратегии матрицы пайплайн сквозного тестирования теперь можно запускать для каждого PR.
- ~~*Сигнал* — пайплайны сквозного тестирования и выпуска релизов генерируют сигналы слишком поздно~~. Исправлено! Как сквозные тесты, так и пайплайн выпуска релизов сейчас запускаются для каждого PR. Проблемы с пайплайном выпуска теперь, скорее всего, будут обнаруживаться при запуске CI.

Поскольку все тесты CI теперь запускаются для каждого PR и разработчики PetMatch используют пайплайн выпуска как часть этого CI, они могут получить максимальное количество сигналов. Чтобы этого достичь, им пришлось принять ряд компромиссов:

- *Скорость получения сигналов* — пайплайн CI теперь генерирует больше сигналов, чем раньше, но на его выполнение уходит почти 20 минут. Без сквозных тестов этот пайплайн занимал максимум пару минут.
- *Частота получения сигналов* — команда сервиса подбора соответствий предпочла частоту получения сигналов CI скорости: теперь она получает большую часть сигналов быстрее, но вынуждена ждать дольше при каждом запуске пайплайна, что означает увеличение времени ожидания для PR, готовых к слиянию.
- *Влияние того, что код остается не готов к релизу*, — до сих пор код оставался в состоянии неизвестности; невозможно было обнаружить ошибки до проведения ночных сквозных тестов или до выпуска релиза. Как было показано в предыдущих главах, такое состояние противоречит основным принципам CD.
- *Ресурсы, необходимые для выполнения пайплайнов*, — каждый вызов пайплайна CI (который может происходить по несколько раз для каждого PR) теперь требует предоставления среды SUT и запуска тестов на 6 шардах. По сравнению с выполнением тестов один раз за ночь CI команды сервиса подбора соответствий будет потреблять гораздо больше ресурсов.

Важные особенности непрерывной доставки

Разработчики сервиса подбора соответствий смогли решить проблемы с CD, реорганизовав задачи и пайплайны. При этом они использовали систему CD, предоставляющую следующую функциональность:

- *Поддержка задач и пайплайнов* — определенная поддержка как небольших наборов функциональности (задач) с сильной связностью, так и организации этих повторно используемых наборов (пайплайнов) обеспечивает гибкость реализации пайплайнов CD.
- *Выходные данные* — благодаря тому, что задачи могут выпускать выходные данные, принимаемые другими задачами, можно создавать хорошо отлаженные, повторно используемые задачи с сильной связностью и слабой связанностью.
- *Входные данные* — точно так же разрешение задачам и пайплайнам принимать входные данные облегчает повторное использование задач и пайплайнов.
- *Условное выполнение* — повторное использование пайплайнов можно расширить, если управлять выполнением некоторых задач в них с помощью входных данных.
- *Finally-логика* — часто в пайплайны CD входят задачи, которые должны выполняться даже в случае сбоя в других частях пайплайна, например задачи привести ресурсы после выполнения в надлежащее состояние или отправить разработчикам уведомление об успехе или провале. Следовательно, должна существовать возможность указывать такие задачи.
- *Параллельное выполнение* — благодаря параллельному выполнению задач, которые не зависят друг от друга, можно быстро и просто увеличить скорость выполнения пайплайна.
- *Выполнение на основе стратегии матрицы* — во многих пайплайнах CD есть задачи, которые должны выполняться для каждой комбинации набора значений (например, при шардинге тестов), и применение матричного (или циклического) синтаксиса облегчает повторное использование и распараллеливание задач.
- *Пайплайны, инициирующие вызов пайплайнов*, — создание пайплайнов с возможностью повторного использования не только упрощает построение сложных пайплайнов в случае необходимости, но и позволяет обеспечить одинаковую логику, даже если сценарии немного отличаются (например, развертывание в тестовой и в рабочей среде).

Что делать, если в выбранной системе CD нет этих функций?

Если ваша CD-система не поддерживает эти функции, вам придется либо делать пайплайны очень простыми (что ограничивает возможности их использования), либо создавать функции самостоятельно. Зачастую при этом потребуется создавать собственные библиотеки для таких функций, как циклы, но это ограничит эффективность этих решений (например, цикл может работать только на одной машине). Из-за отсутствия перечисленных выше функций часто приходится создавать сложные задачи с большим количеством обязанностей, включая логику оркестровки. Старайтесь выбирать систему CD, которая выполняет эту работу за вас. Возможности некоторых популярных систем CD описаны в приложении А.

Заключение

Чтобы решить проблемы в пайплайнах CD, иногда достаточно перепроектировать пайплайн. Разработчикам сервиса подбора соответствий не нужно было менять функциональность задач и пайплайнов. Используя функции системы CD и реорганизовав существующие пайплайны, они значительно повысили их эффективность — ценой снижения скорости выполнения и увеличения потребления ресурсов, но эту цену они готовы были заплатить.

Итоги

Старайтесь выбирать систему CD, в которой предусмотрена следующая функциональность, и используйте эти функции, когда поймете, что пайплайны не генерируют необходимые сигналы:

- задачи (повторно используемые функциональные единицы) и пайплайны (которые организуют работу этих задач);
- входные и выходные данные (как параметры задач и пайплайнов);
- условное выполнение;
- finally-логика;
- параллельное выполнение и поддержка стратегии матрицы;
- повторно используемые пайплайны.

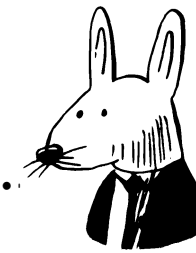
Кроме того, организуя пайплайны и принимая решение, что и когда в нем выполнять, учитывайте следующие факторы:

- скорость получения сигналов (насколько быстро работают пайплайны);
- частота сигналов (при их запуске);
- влияние того, что код остается не готов к релизу;
- ресурсы, необходимые (и доступные) для выполнения пайплайнов.

Далее...

Вы добрались до конца последней главы! Надеюсь, вам понравилось знакомство с непрерывной доставкой (CD) так же, как и мне. Приложения в конце книги содержат анализ рассмотренных особенностей некоторых популярных систем CD и контроля версий.

Приложения

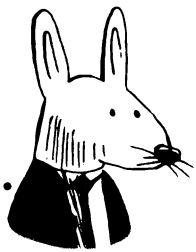


В двух приложениях содержится анализ некоторых функций, описанных в данной книге и предоставляемых популярными (на момент написания книги) системами непрерывной доставки и контроля версий.

В приложении A рассматриваются функции непрерывной доставки, предоставляемые инструментами Argo Workflows, CircleCI, GitHub Actions, Google Cloud Build, Jenkins Pipelines и Tekton.

В приложении B рассматриваются системы контроля версий с акцентом на Git и Git-сервисы Bitbucket, GitHub и GitLab.

A | Системы непрерывной доставки



В этом приложении

- ✓ Функции систем непрерывной доставки, рассматриваемые в данной книге
- ✓ Обзор распространенных систем непрерывной доставки и их возможностей

В этом приложении мы вспомним функциональные возможности систем непрерывной доставки, которые обсуждались на протяжении всей книги, и рассмотрим, какие из них доступны в нескольких часто используемых системах CD.

В рамках этой книги функционал систем CD рассматривался на примере GitHub Actions. Однако, выбирая наиболее подходящую систему CD для своих потребностей, важно изучить и сопоставить все доступные варианты (в том числе создание собственной системы CD — см. главу 11). Здесь мы проанализируем следующие системы CD:

- Argo Workflows;
- CircleCI;
- GitHub Actions;
- Google Cloud Build;
- Jenkins Pipeline;
- Tekton.

Этот список не исчерпывающий. При оценке систем CD, которые сюда не вошли, руководствуйтесь функциями, перечисленными в разделе «Список функций».

Функциональные возможности систем CD по главам

В 13 главах этой книги был рассмотрен широкий спектр функциональных возможностей систем CD, позволяющих упростить разработку сложных, повторно используемых задач и пайплайнов. О функциях систем CD шла речь в следующих главах:

- *Глава 2* — обзор основных элементов пайплайна CD и терминологии, используемой в книге для их обозначения (события, триггеры, веб-хуки, задачи, пайплайны и др.).
- *Глава 3* — концепция «конфигурация как код» и важность рассмотрения конфигурации CD как кода, с сохранением ее в системе контроля версий.
- *Глава 7* — перечень мест проникновения ошибок в код, польза от периодического запуска тестов.
- *Глава 9* — важность безопасного создания программных артефактов и необходимые для этого функции.
- *Глава 12* — скрипты в концепции «конфигурация как код»: вся конфигурация CD должна рассматриваться как код, а пайплайны и задачи CD должны быть повторно используемыми.
- *Глава 13* — набор функций уровня пайплайна, необходимый для создания эффективных пайплайнов, и зачем это нужно.

При оценке систем CD следует обращать внимание именно на эти функции. Если они отсутствуют в системе, это не значит, что она непригодна к использованию; вам просто придется проделать больше работы, чтобы получить эту функциональность.

Список функций

Для часто используемых систем CD мы рассмотрим следующие функции.

Функции запуска (см. главы 2 и 7)	
Запуск на основе событий — пайплайны должны запускаться и выполняться в ответ на различные события, например при открытии или слиянии запросов на вытягивание (пул-реквестов)	*Периодический запуск* — регулярное запланированное выполнение пайплайнов может выявить такие проблемы, как flaky-тесты, и использоваться для поддержки различных стратегий выпуска релизов, например ночных релизов

Безопасные и надежные функции процесса сборки (см. главы 3 и 9)		
Конфигурация как код (или сборка как код) — принципиально важно хранить пайплайны CD в системе контроля версий; ко всем данным, лежащим в основе разрабатываемого продукта, должны применяться те же лучшие практики, что и к бизнес-логике	*Запуск в виде сервиса* — без согласованного сервиса, выполняющего CD-пайплайны, невозможно обеспечить стабильность сборки продукта и практически невозможно обеспечить его аудит	*Временные среды* — сборку артефакта всегда необходимо начинать с создания чистой среды, чтобы получать одинаковые результаты. Для этого часто используются одноразовые виртуальные машины и контейнеры для выполнения

Элементы выполнения (см. главу 2)	
Пайплайны — основанные на графах оркестрации задач с функциями управления потоком, специфичными для случаев использования CD	*Задачи* — элементы логики, которые должны обладать сильной связностью, слабой связанностью, возможностью повторного использования и быть хорошо структурированы

Функции задач и пайплайнов (см. главы 12 и 13)		
Входные и выходные данные — чтобы задачи и пайплайны можно было использовать повторно, необходимо обеспечить их входными данными, которые позволят настраивать их работу. Выходные данные дают возможность объединять задачи и пайплайны, чтобы их поведение могло меняться в зависимости от результатов работы других задач и пайплайнов	*Условное выполнение* — возможность контролировать, какие части пайплайна выполняются, позволяет писать гибкие пайплайны, которые отлично подходят для повторного использования	*Finally-логика* — пайплайны CD часто содержат блоки, которые должны выполняться, даже если другие части пайплайна не работают. Например, очистка сред и отправка уведомлений, таких как мгновенные сообщения
Параллельное выполнение — параллельное выполнение в пайплайне не зависящих друг от друга задач может сократить общее время выполнения пайплайна	*Выполнение на основе стратегии матрицы* — это расширенная версия параллельного выполнения, автоматизирующая параллельное повторное выполнение задачи для комбинаций входных данных и поддерживающая сложные случаи использования тестирования, такие как шардинг	*Пайплайны, использующие пайплайны,* — в дополнение к повторно используемым задачам часто бывает полезно определить повторно используемые пайплайны, которые объединяют задачи, связывая их входные и выходные параметры, и использовать эти пайплайны в нескольких сценариях

Argo Workflows

Система Argo Workflows (https://argoproj.github.io/workflows/) — это один из нескольких проектов под брендом Argo. Argo Workflows — это обработчик рабочих процессов с открытым исходным кодом на основе Kubernetes, который был подарен фонду Cloud Native Computing Foundation (CNCF) компанией Intuit (а разработан компанией Applatix). Он поддерживает сценарии использования CD, но был создан для решения более широких задач автоматизации рабочих процессов.

Функции запуска
Запуск осуществляется на базе сопутствующего проекта *Argo Events*, который поддерживает множество источников событий (включая системы контроля версий, такие как GitHub, и интеграцию с облаком)

Безопасные и надежные функции процесса сборки	
Размещать и запускать Argo необходимо самостоятельно; он может использоваться для предоставления сервиса CD внутри организации	Контейнеры — основной элемент выполнения, обеспечивающий временную среду

Элементы выполнения	
Рабочие процессы (соответствующие пайплайнам в данной книге) выполняют шаблоны рабочих процессов	*Шаблоны рабочих процессов* (повторно используемые рабочие процессы) содержат *шаги* и/или *ориентированные ациклические графы*

Функции задач и пайплайнов		
Шаблоны рабочих процессов являются повторно используемыми, они используют входные параметры и выдают результаты посредством артефактов. Шаги определяют входные и выходные данные (в виде параметров или артефактов)		
Условное выполнение поддерживается с помощью синтаксиса when	Матричное выполнение поддерживается с помощью функции циклов	Функциональность типа «пайплайны использующие пайплайны» обеспечивается с помощью шаблонов рабочих процессов, вызывающих другие шаблоны

CircleCI

CircleCI (https://circleci.com/docs/2.0/concepts/) — это система CD, предоставляемая одноименной компанией и построенная на идее использования репозитория в качестве источника истины для конфигурации CD («конфигурация как код»).

Функции запуска
Интегрируется непосредственно с системами контроля версий и может инициировать выполнение на основе событий из этих систем, включая GitHub и Bitbucket, а рабочие процессы поддерживают запуск по расписанию

Безопасные и надежные функции процесса сборки		
CircleCI построен на идее «конфигурация как код» и предполагает размещение конфигурации CD для проекта в папке с названием .circleci в корне репозитория	Его можно использовать через общедоступный хостинг или запустить в качестве сервера локально	Среды сборки (называемые *исполнителями — executors*) временные и могут поддерживаться виртуальными машинами или контейнерами; при желании можно повторно использовать виртуальные машины для разных заданий (это означает, что среда выполнения не является временной); контейнеры никогда не используются повторно

Элементы выполнения	
Пайплайны (соответствуют пайплайнам из данной книги) содержат рабочие процессы, а также инициирующую информацию	*Рабочие процессы* (также почти соответствуют нашим пайплайнам) производят оркестровку заданий
Задания (соответствуют задачам в рамках этой книги) содержат последовательные шаги	*Команды* можно повторно использовать в рамках шагов заданий
Орбы (orbs) (также приблизительно соответствуют задачам из данной книги, но немного выходят за их рамки) определяют повторно используемые задания, команды и исполнителей	

Функции задач и пайплайнов		
Параметры могут определяться заданиями, командами и исполнителями. Использование выходных данных между заданиями осуществляется путем сохранения данных в рабочих пространствах или с помощью переменной среды `BASH_ENV`	Шаги внутри задания могут выполняться по условию при использовании ключевых слов `when` и `unless`	Finally-логика на уровне шагов поддерживается путем указания условия `always`
Задания в рамках рабочего процесса могут выполняться параллельно или последовательно. При последовательном выполнении заданий используется ключевое слово `requires` для определения зависимостей между заданиями. Шаги в задании могут выполняться параллельно с помощью ключевого слова `parallelism`		Ключевое слово `matrix` позволяет выполнять задания несколько раз, по одному разу для каждой уникальной комбинации элементов матрицы

Система GitHub Actions

GitHub Actions (https://docs.github.com/en/actions) — это система CD, встроенная в GitHub (подробнее о системах контроля версий см. в приложении B). Система GitHub Actions использовалась для примера в большинстве сценариев этой книги, потому что в ней легко создавать свои собственные репозитории GitHub и настраивать свои собственные задачи в GitHub. Она поддерживает функции, описанные в данной книге, как часть своей широкой функциональности.

Функции запуска
Инициализация множеством типов активности для каждого репозитория, от запланированных cron-событий до обновлений запросов на вытягивание и работы с проблемами GitHub

Безопасные и надежные функции процесса сборки		
Поддерживает подход «конфигурация как код» — более того, это единственный способ настройки рабочих процессов в GitHub. Характеристики рабочих процессов находятся в репозитории, который их запускает, и при этом можно ссылаться на рабочие процессы и действия, которые находятся в других репозиториях	При использовании общедоступного GitHub система CD размещается и запускается на самом GitHub. Если вы используете GitHub Enterprise локально, вы сами отвечаете за настройку и запуск платформы, выполняющей задачи	Каждое задание в GitHub Actions выполняется во временной среде, которая создается специально для этого и после удаляется. Задание может быть запущено как целая виртуальная машина или как контейнер внутри виртуальной машины

Элементы выполнения		
Рабочие процессы (соответствующие пайплайнам данной книги) производят оркестровку заданий	*Задания* (соответствующие задачам в рамках этой книги) содержат последовательно выполняемые шаги	*Действия* (также соответствующие задачам) представляют собой повторно используемые задания

Функции задач и пайплайнов		
Задания в рамках рабочих процессов могут определять и выпускать выходные данные, которые другие задания в рамках того же рабочего процесса затем используют как входные данные. Повторно используемые рабочие процессы могут также определять входные и выходные данные	Задания в рабочих процессах GitHub используют инструкцию `if` для определения условий, при которых они будут выполняться	Finally-логика также поддерживается с помощью этого синтаксиса путем указания `if always()` в качестве условия для задания, чтобы указать, что оно должно быть выполнено в любом случае
По умолчанию все задания в рабочем процессе выполняются параллельно, если только не используется синтаксис needs, указывающий, что одно задание должно выполняться после другого задания	Ключевое слово `matrix` позволяет выполнять задания несколько раз, по одному разу для каждой уникальной комбинации значений в матрице	Рабочие процессы могут быть определены как повторно используемые, которые могут вызываться другими рабочими процессами

Google Cloud Build

Google Cloud Build (GCB; https://cloud.google.com/build) — это платформа для создания CD, предоставляемая компанией Google в рамках облачных приложений. Изначально она проектировалась как способ создания образов контейнеров, но быстро превратилась в универсальный инструмент CD.

Функции запуска
Поддерживает срабатывание на основе событий из интегрированных систем контроля версий, включая GitHub, GitLab и Bitbucket, а также запуск веб-хуков посредством Pub/Sub в GCP и периодический запуск по расписанию

Безопасные и надежные функции процесса сборки		
Срабатывание GCB может быть настроено на чтение определения сборки из запускаемого репозитория, поддерживая принцип «конфигурация как код»	GCB предоставляется как сервис, размещаемый и запускаемый на Google в рамках Google Cloud Platform (GCP)	Шаги в сборках GCB выполняются в виде временных контейнеров, которые запускаются на виртуальных машинах

Элементы выполнения	
Сборки состоят из шагов. Они соответствуют задачам в данной книге, а также в некоторой степени пайплайнам (поскольку шаги могут выполняться как графы, с последовательным или параллельным выполнением шагов)	Каждый *шаг* выполняется как контейнер

Функции задач и пайплайнов	
Сборки могут использовать входные данные, предоставляемые во время выполнения, с помощью функции подстановок, определяемой пользователем	По умолчанию шаги в сборке выполняются последовательно; ключевое слово `waitFor` может быть использовано для явного объявления порядка выполнения шагов и для создания графов, в которых некоторые шаги выполняются параллельно

Jenkins Pipeline

Jenkins (https://www.jenkins.io/doc/book/pipeline/) — одна из самых известных и, вероятно, самых часто используемых систем CD. Jenkins имеет открытый исходный код, и благодаря огромной экосистеме плагинов его можно заставить делать практически все что угодно. Мы будем считать, что Jenkins используется с набором плагинов Jenkins Pipeline, которые ориентированы на поддержку пайплайнов CD в Jenkins.

Функции запуска
Систему Jenkins Pipeline можно настроить на запуск по различным событиям, включая триггер pollSCM, который можно использовать для опроса различных систем управления версиями (например, GitHub), и триггеры cron, которые позволяют планировать выполнение

Безопасные и надежные функции процесса сборки		
Пайплайны Jenkins определяются в *файле Jenkinsfiles*, который хранится в системе контроля версий и считывается непосредственно системой Jenkins, что позволяет использовать принцип «конфигурация как код» для характеристик пайплайна	Размещать и запускать Jenkins необходимо самостоятельно. Его можно использовать для предоставления сервиса CD внутри организации	Поддерживаются различные виды сред выполнения, включая контейнеры, которые могут использоваться как временные среды

Элементы выполнения	
Пайплайны (соответствующие пайплайнам этой книги) состоят из этапов	*Этапы* (соответствующие задачам этой книги) состоят из шагов

Функции задач и пайплайнов		
Пайплайны могут использовать ключевое слово `parameters` для определения необходимых входных данных	Директива `when` может использоваться для условного выполнения этапов	Finally-логика содержится в секции `post` пайплайна, где указаны шаги, которые должны быть выполнены после завершения остальных этапов (даже если эти этапы завершатся неудачно)
Можно настроить параллельное выполнение этапов пайплайна, определив их во вложенных блоках `parallel`	Ключевое слово `matrix` позволяет выполнять этапы повторно, по одному разу для каждой уникальной комбинации определенных наборов	Пайплайны могут использовать другие пайплайны, определенные в общих библиотеках Jenkins
Пайплайны могут быть как декларативными, так и полностью скриптовыми с использованием языка программирования *Groovy*. Скриптовые пайплайны обладают абсолютной гибкостью в поддержке условного выполнения и finally-логики (при помощи `try/catch`)		

Tekton

Tekton (https://tekton.dev/ и https://github.com/tektoncd) — это система CD с открытым исходным кодом на базе Kubernetes, которая была безвозмездно передана фонду Continuous Delivery Foundation (CDF) компанией Google. Ее цель — определить соответствующий стандарт, который могут поддерживать различные системы CD. Функции, описанные в данной книге,

Над системой Tekton в том числе работала и я!

в основном поддерживаются проектами Tekton Pipelines и Tekton Triggers. Так как это самая молодая из систем CD, перечисленных в этом приложении, некоторые из ее функций совершенно новые или находятся на этапе разработки.

Функции запуска
Срабатывание по любому произвольному событию осуществляется с использованием проекта Tekton Triggers, который поддерживает любые источники событий и имеет встроенную поддержку для функций запуска GitHub, GitLab и Bitbucket

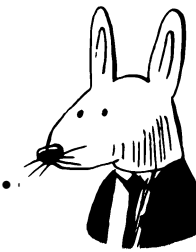
Безопасные и надежные функции процесса сборки	
Tekton можно запускать как сервис внутри кластера Kubernetes, использовать напрямую или в качестве платформы, на которой можно построить другой сервис CD	Базовым элементом выполнения в Tekton является контейнер, поэтому среды выполнения только временные

Элементы выполнения		
Пайплайны (соответствующие пайплайнам этой книги) выполняют оркестровку задач	*Задачи* (соответствующие задачам этой книги) последовательно выполняют шаги	Каждый *шаг* выполняется как контейнер

Функции задач и пайплайнов		
Как пайплайны, так и задачи могут определять входные (`parameters`) и выходные данные (`results`)	Условное выполнение задач внутри пайплайна поддерживается с помощью синтаксиса `when`	Finally-логика в пайплайнах поддерживается секцией `finally`, в которой определены задачи, требующие выполнения независимо от результата других
По умолчанию все задачи в пайплайне выполняются параллельно, пока между ними не появляется зависимость. Зависимости выражаются в том, что одна задача объявляет, что ей нужен `result` из другой задачи (и поэтому она должна выполняться после той задачи), или используется ключевое слово `runAfter` для указания на упорядоченность		Ключевое слово `matrix` в пайплайне может быть использовано для выполнения данной задачи для всех уникальных комбинаций значений массива, указанных в описании матрицы

На момент написания книги следующие функции находились в разработке	
Встроенная поддержка принципа «конфигурация как код» путем прямого обращения к пайплайнам и задачам в системе контроля версий (и в других местах, например в реестрах OCI)	Пайплайны, повторно использующие другие пайплайны: подобно тому как пайплайны производят оркестровку задач, они могут обращаться к другим пайплайнам

Системы контроля версий | B



В этом приложении

- ✓ Перечень функций систем контроля версий, рассматриваемых в данной книге
- ✓ Краткий обзор популярных систем контроля версий
- ✓ Обзор сервисов Git и их функций

В этом приложении мы кратко рассмотрим системы контроля версий в целом, а также напомним о возможностях удаленных систем контроля версий, которые обсуждались в этой книге.

Для многих примеров в книге использовалась система GitHub, но существуют и другие Git-сервисы, которые стоит рассмотреть в качестве возможной системы контроля версий для проекта. Мы кратко остановимся на некоторых из них и опишем, какие возможности они предоставляют:

- Bitbucket;
- GitHub;
- GitLab.

Этот список далеко не исчерпывающий. Принимая решение о том, нужен ли вам вообще сервис Git или лучше использовать другие инструменты, руководствуйтесь функциями удаленных систем контроля версий, перечисленными в разделе «Список функций».

Системы контроля версий

Подробнее об использовании VCS и о том, почему это необходимо при непрерывной доставке, читайте в главе 3.

Использование *системы контроля версий* (*VCS*, version control system) для текстовых данных, лежащих в основе создаваемого продукта (в том числе его конфигурации), — обязательное условие непрерывной доставки. VCS делятся на две категории:

- *Централизованная VCS* — работает на центральном сервере, который является источником истины, и все изменения направляются на этот сервер. Это модель по умолчанию для старых вариантов VCS.
- *Распределенная VCS* — каждый пользователь такой VCS получает собственную копию всей кодовой базы. Центрального сервера не существует. Однако на практике один сервер обычно рассматривается как источник истины для проекта. Современные и популярные VCS по умолчанию распределенные.

Примеры централизованных VCS:

- *Apache Subversion* (SVN; https://subversion.apache.org/) — система с открытым исходным кодом, созданная компанией CollabNet в 2000 году, версия 1.0 выпущена в 2004 году.
- *Perforce Helix Core*, ранее *Perforce* (https://www.perforce.com/products/helix-core), — выпущена компанией Perforce Software (компания основана в 1995 году).
- *Concurrent Versions System* (CVS; http://cvs.nongnu.org/) — создана в 1986 году, версия 1.0 выпущена в 1990 году.

Примеры распределенных VCS:

- *Git* (https://git-scm.com/) — система с открытым исходным кодом, выпущенная Линусом Торвальдсом (Linus Torvalds) в 2005 году.
- *Mercurial* (https://www.mercurial-scm.org/) — система с открытым исходным кодом, созданная в 2005 году Оливией Мэколл (Olivia Mackall).

Самым популярным вариантом VCS в настоящее время является Git. Если выбор неочевиден и для использования другой системы веских причин нет, выбирайте Git. Он может показаться сложным, особенно если вы привыкли использовать централизованные VCS, поэтому сперва уделите время изучению основ его работы.

Хостинг системы контроля версий

Если вы решили использовать Git, можете либо запускать его самостоятельно, либо использовать хостинг. Хостинговые Git-решения не просто предоставляют экземпляры Git для взаимодействия, они добавляют функции в сам Git.

Если вы все-таки решите не использовать хостинг, вам, скорее всего, придется добавить поверх Git ПО для управления проектами и проверки кода. Кроме того, хотя вы по-прежнему сможете перехватывать выполнение пайплайна CD, вам придется самостоятельно организовать управление инициализацией. В этой книге вы узнали, на какие функции следует обращать внимание при выборе решений VCS, размещаемых на хостинге:

- *Глава 2.* Основы системы CD, включая функциональные возможности, предоставляемые размещенными VCS, такие как веб-хуки, уведомления, события и триггеры.
- *Глава 3.* Важность принципа «конфигурация как код», когда во главу угла всей CD ставится контроль версий. Объяснение, почему для эффективной интеграции контроля версий в CD необходимо инициировать выполнение пайплайна на основе изменений данных в системе контроля версий.
- *Глава 7.* Жизненный цикл изменений и возможные места появления багов. Эффективное устранение этих багов требует инициирования выполнения на основе событий, поступающих от системы контроля версий, и использования очередей слияния, чтобы действительно убедиться, что баги не появляются по причине трудноуловимых конфликтов между изменениями.
- *Глава 8.* Влияние способа использования контроля версий на скорость доставки изменений пользователям. Важность магистральной разработки и проверки кода.
- *Глава 9.* Лучшие практики сборки, включая использование принципа «сборка как код» и возможность запуска при появлении тега (в системе управления версиями) для создания релизов.
- *Глава 11.* Как начать работу с пайплайнами CD. Краткий обзор частных репозиториев (функционал, который иногда поддерживает хостинг системы контроля версий).

Для описания размещенных на хостинге VCS иногда используется термин *SCM*; эту аббревиатуру обычно расшифровывают в широком смысле как «*управление исходным кодом*» (source code management). Однако термин SCM возник намного раньше, чем подобная нынешняя трактовка, и это может сбивать с толку. Например, в собственной документации Git и CVS называют себя *инструментами SCM* и при этом не имеют в виду размещенные на хостинге VCS с дополнительными функциями, такими как проверка кода, а именно в этом значении этот термин часто используется сегодня. Подробнее читайте в главе 3.

Список функций

Ниже приведен обзор основных функциональных возможностей, предоставляемых системами контроля версий, поддерживаемыми Git.

Управление и реализация самой системы		
Хостинг/локальное размещение — некоторые VCS размещаются на хостинге предоставляемой системы, а некоторые позволяют размещать и запускать собственные экземпляры. Требования безопасности и соответствия внутренним нормативным требованиям организации могут ограничить применение до только локальных решений	*Общедоступные/частные репозитории* — размещенные на хостинге VCS часто поддерживают общедоступные репозитории, однако многие организации (возможно, большинство) не хотят делать свой исходный код общедоступным и поэтому нуждаются в приватных репозиториях	*Открытый исходный код* — если сама размещенная на хостинге VCS имеет открытый исходный код, вы сможете не только его просматривать, но и теоретически отправлять обновления и исправления багов. Возможность просмотра исходного кода повышает доверие к системе. Для вас и вашей организации это может стать важным фактором при выборе системы (а может и не стать)

Планирование и координация проекта		
Отслеживание проблем — минимальная функциональность, необходимая для отслеживания и планирования проектов, — это возможность создавать задания (проблемы), которые описывают и отслеживают запросы функций и баги. Кроме того, полезно иметь возможность добавлять *метки* для этих заданий, чтобы классифицировать их и *назначать* конкретным исполнителям	*Управление проектами* — регистрировать проблемы (задания) полезно, но часто требуются дополнительные инструменты для планирования действий, например *доска проектов* для просмотра проблем и *контрольных точек*, чтобы спланировать, какие будут решаться задачи и в каких релизах	*Релизы* — обычно релизы в Git-репозиториях помечаются *тегами* (это функция самого Git); также полезно, если размещенное Git-решение позволяет добавлять дополнительные метаданные к тегу, такие как заметки о релизе и о самих выпущенных артефактах

Триггеры и выполнение CD	
Веб-хуки — позволяют инициировать выполнение действия на основе событий VCS объектами, находящимися за пределами VCS	*Очереди слияния* — их также иногда называют «*поездами слияния*» (merge trains). Эта функция гарантирует, что потенциальные изменения будут проверены с самой последней версией главной ветки, чтобы в коде не возникали конфликты

Триггеры и выполнение CD	
Перехватчики для систем CD — помимо поддержки запуска внешних систем с помощью веб-хуков, важно, чтобы эти внешние системы сообщали о своем состоянии, например, что пайплайн CD не прошел и поэтому слияние данного запроса на вытягивание должно быть заблокировано	*Встроенная система CD* — некоторые VCS в стандартном варианте включают полную систему CD в дополнение или вместо поддержки перехватчиков для внешних систем CD

Слияние и проверка кода		
Запросы на вытягивание, или пул-реквесты (PR), — стандартная последовательность действий при слиянии кода с главной веткой заключается в создании запроса на добавление изменений, называемого *запросом на вытягивание* (то есть запросом на «вытягивание» изменений в главную ветку) или *запросом на слияние*. На этом механизме основывается проверка изменений перед их внесением, как автоматически с помощью CD, так и вручную посредством проверки кода	*Форки, или ответвления* (forks), — в распределенных системах контроля версий, таких как Git, обычно существуют ограничения на типы создаваемых веток в репозитории источника истины. В то же время в процессе работы над изменениями полезно иметь возможность создавать (короткоживущие) ветки. Стандартным решением для репозиториев Git является поддержка создания собственных удаленных копий репозитория, называемых *форками*, над которыми создатель имеет полный контроль	*Проверка кода, или кодревью*, — чтобы поддерживать работоспособность кода и обмен информацией между разработчиками, важно иметь правильную систему проверки кода. Для этого необходимы инструменты, позволяющие *различать* изменения (то есть видеть, какие именно изменения относятся к текущему состоянию кода), *комментировать* изменения (например, запрашивать их уточнение и/или модификацию) и *одобрять* изменения для слияния

Bitbucket

Bitbucket (https://bitbucket.org/product/features) — это размещенная на хостинге VCS на основе одноименного стартапа, приобретенного компанией Atlassian в 2010 году. Первоначально она поддерживала Mercurial, но теперь переориентировалась на Git.

Управление и реализация самой системы		
Хостинг/локальное размещение — имеет как локальный, так и облачный вариант	*Общедоступные/частные репозитории* — размещенный на хостинге Bitbucket поддерживает как общедоступные, так и частные репозитории	Хотя Bitbucket не является приложением с открытым исходным кодом, компания предоставляет доступ к исходному коду пользователям с определенными лицензиями

Планирование и координация проекта
Отслеживание проблем и управление проектами — эта функциональность поддерживается благодаря интеграции Bitbucket с системами Jira и Trello

Триггеры и выполнение CD	
Веб-хуки — система Bitbucket поддерживает веб-хуки	*Очереди слияния* — эта функциональность не встроена в Bitbucket, но доступна через проект с открытым исходным кодом Atlassian Landkid
Перехватчики для систем CD — для принудительного прохождения проверок перед началом слияния требуется приложение Bitbucket Premium	*Встроенная система CD* — встроенная поддержка пайплайна CD осуществляется посредством функции Bitbucket Pipes

Слияние и проверка кода		
Пул-реквесты — поддерживаются	*Форки* — поддерживаются	*Код-ревью* — поддерживается

GitHub

GitHub (https://github.com/features) — это размещенное на хостинге приложение Git, которое используется во многих проектах с открытым исходным кодом. Оно было создано одноименной компанией в 2007 году и приобретено Microsoft в 2018 году.

Управление и реализация самой системы	
Хостинг/локальное размещение — многим знакома онлайн-версия GitHub. Однако существует и корпоративный вариант (Enterprise), который предлагает как локальные, так и облачные приложения	*Общедоступные/частные репозитории* — GitHub предлагает общедоступные репозитории (используемые во многих проектах с открытым исходным кодом), а также частные репозитории, как в рамках того же размещенного на хостинге приложения, так и в рамках Enterprise-версии

Планирование и координация проекта		
Отслеживание проблем — репозитории GitHub включают связанные с ними задания (проблемы). Проблемы могут иметь *метки* и *назначенных исполнителей*, а пользователи могут создавать *шаблоны* для конкретных видов проблем (например, багов или запросов функций)	*Управление проектами* — работу можно организовать и координировать с помощью *досок проектов*, *таблиц проектов* и *контрольных точек*	*Релизы* — в репозиториях GitHub могут размещаться релизы, связанные с Git-тегами. Релизы могут включать *примечания к выпуску* и *артефакты*

Триггеры и выполнение CD	
Веб-хуки — GitHub поддерживает веб-хуки и предоставляет подробные данные о событиях через информационное наполнение веб-хуков	*Очереди слияния* — на момент написания этой книги GitHub поддерживал очереди слияния в виде ограниченной общедоступной бета-версии
Перехватчики для систем CD — внешние CD-системы и боты могут посылать запросы в GitHub на предоставление подробной (даже построчной) информации об изменениях в запросе на внесение с помощью функции проверок	*Встроенная система CD* — GitHub предоставляет встроенную CD-систему GitHub Actions, которая используется во многих примерах этой книги (см. приложение A)

Слияние и проверка кода		
Пул-реквесты — поддерживаются	*Форки* — поддерживаются	*Код-ревью* — поддерживается

GitLab

GitLab (https://about.gitlab.com/features/) — это размещенное на хостинге Git-приложение с открытым исходным кодом, созданное одноименной компанией в 2014 году.

Управление и реализация самой системы		
Хостинг/локальное размещение — GitLab можно размещать локально или использовать через хостинг компании	*Общедоступные/частные репозитории* — проекты на хостинге GitLab могут быть как частными, так и общедоступными	*Открытый исходный код* — GitLab является приложением с открытым исходным кодом

Планирование и координация проекта		
Отслеживание проблем — в репозиториях GitLab можно определять задания (проблемы). Проблемы можно создавать на основе *шаблонов* для различных типов (например, баги или запросы функций) и *снабжать метками*. Поддержка нескольких *назначенных исполнителей* заданий и группировки заданий в *эпики* требует установки GitLab Premium	*Управление проектами* — работа во всех приложениях организуется и координируется с помощью *досок заданий (проблем)* и *контрольных точек*; *дорожные карты* доступны только для пользователей GitLab Premium	*Релизы* — в репозиториях GitLab могут размещаться релизы, связанные с Git-тегами. Релизы могут включать *примечания к выпуску* и *артефакты*

Триггеры и выполнение CD	
Веб-хуки — GitLab поддерживает веб-хуки	*Очереди слияния* — GitLab Premium поддерживает очереди слияния под названием «*поезда слияния*»
Перехватчики для систем CD — GitLab Ultimate поддерживает эту функциональность через функцию External Status Checks	*Встроенная система CD* — GitLab предоставляет встроенную систему CD для всех приложений

Слияние и проверка кода		
Пул-реквесты — поддерживаются. В GitLab они называются *запросами на слияние* (merge requests)	*Форки* — поддерживаются	*Код-ревью* — поддерживается

грокаем

continuous delivery

Кристи Уилсон

Вступительное слово: Джез Хамбл и Эрик Брюэр

Код должен быть готов к релизу всегда! Пайплайн Continuous Delivery автоматизирует процессы контроля версий, тестирования и развертывания при минимальном вмешательстве разработчика. Освойте инструменты и методы непрерывной доставки, и вы сможете быстро и последовательно добавлять функции и выпускать обновления.

«Грокаем Continuous Delivery» — это руководство по настройке и работе с пайплайном непрерывной доставки. В каждой главе рассматривается отдельный сценарий, с которым вы столкнетесь при создании системы CD, и приводятся реальные примеры, например автоматическое масштабирование и тестирование унаследованных приложений. Кристи Уилсон сопровождает каждый шаг иллюстрациями, кристально четкими объяснениями и практическими упражнениями для закрепления полученных знаний.

В книге:

- Разработка эффективных пайплайнов CD для новых и унаследованных проектов.
- Настройка пайплайнов.
- Контроль версий как источник истины.
- Безопасная автоматизация развертываний.

Для инженеров-программистов, которые хотят добавить CD в свои навыки разработки.

Кристи Уилсон — инженер-разработчик ПО в Google, она является одним из создателей Tekton, истинно облачной платформы CI/CD, построенной на базе Kubernetes.

«Для меня эта книга стала настоящим глотком свежего воздуха».

— Из «Вступительного слова» Эрика Брюэра, Google

«Эта книга поможет лучше понять, как правильно реализовать современный процесс доставки».

— Из «Вступительного слова» Джеза Хамбла, соавтора книги «Continuous Delivery»

«Самый полный контент по непрерывной доставке, который я когда-либо видел. Безусловно, это лучшее практическое введение в тему».

— Уильям Джамир Сильва, Adjust

«Эта книга незаменима для любого разработчика, создающего качественное программное обеспечение. Она проникает прямо в подсознание».

— Даниэль Васкес, Wizeline

«Для всех, кто ищет в реальной жизни способы бесперебойной непрерывной интеграции и доставки программного обеспечения».

— Прабхути Пракаш, Calsoft

ПИТЕР®

WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com

PiterBooks
PiterForPeople
ThePiterBooks
Company/piter

MANNING

ISBN:978-5-4461-2372-8