**7 способов помыть посуду и вариант реакивной системы на передаче сообщений**

**Понимание асинхронного, неблокирующего, конкурентного, параллельного и так далее**

Я работал в течение нескольких лет в попытках найти наиболее выразительное описание основной концепции построения эффективных реактивных приложений – асинхронных и неблокирующих, с минимизацией конкурентности и поддержкой линейной масштабируемости за счет повышения параллелизма. Это настоящий суп из эзотерических терминов, которые сложно понять даже самым опытным разработчикам. Тем не менее, их понимание имеет решающее значение в построении действительно реактивных приложений.

За последние несколько месяцев, я думаю, мне удалось найти способ выразить те концепции более четко. Это также освещает некоторые другие интересные концепции, такие как конвейерная обработка(пайплайнинг), пакетирование (батчинг), fork/join и закон Амдаля с помощью повседневной метафоры. Аналогии с реальным миром всегда помогали мне понять концепции, и когда я пробегался по ним с клиентами, они также находили это полезным.

**Посудомойщики**

Мне нравится мыть посуду. Серьезно, я делаю это. И пока вы думаете про себя «Это довольно странно», любой, кто встречал меня, скажет вам, что это полностью описывает меня. Не только это, я один из тех особенно странных людей, которые не только руками моют посуду, но и складывают ее, когда закончат, в посудомоечную машину для дальнейшей очистки. Я боялся иметь в колледже соседа по комнате, который ел бы макароны с сыром, оставлял бы посуду и кастрюли отмокать в раковине на несколько дней, а потом думал бы, что посудомоечная машина 80ых годов просто удалит всю присохшую еду, которую он оставил. Используя эту аналогию, я обнаружил там еще несколько человек, которые имели схожие посудо-неврозы, которые содействовали мне в преодолении моего обсессивно-компульсивного поведения относительно мытья посуды. Но эта история не о моей потребности в профессиональной терапевтической помощи.

Каждую ночь после ужина у меня собирается в раковине стопка тарелок, которые должны быть помыты. Этот процесс очень похож на то, что вы видите при обработке потоков данных. Я использую конвейерный процесс преобразований: смывание грязи с каждой тарелки, очистки в мыльной воде, полоскание, чтобы убрать пену, и помещение тарелки в посудомоечную машину. Этот процесс является чисто синхронным (все обрабатывается одним процессором/человеком) и последовательным (порядок не может быть изменен), так как я могу выполнять только одну задачу за раз. Я мог бы отправлять каждую тарелку по отдельности в конвейер или, без гарантии последовательности, попробовать пропустить через все операции по несколько, сперва прополоскав все блюда, далее очистив их, снова прополаскав и потом разместив все тарелки в посудомоечной машине группами. При этом, я, возможно, увеличил свою производительность незначительно за счет увеличения концентрации данных (каждой тарелки) в местах проведения операций (раковина, губка для мытья, посудомоечная машина и т.д.), но моя производительность ограничена тем фактом, что я всего лишь единственный, кто выполняет всю работу.

Представьте, что у меня есть хороший друг, который тоже любит заниматься мытьем посуды. Зная, как сильно он или она либит мыть посуду, я приглашаю этого человека ко мне домой, чтобы тот помог вечером. Теперь я создал пул потоков – несколько потоков выполнения, которые могут или не могут внести свой вклад в работу, которую я хочу сделать. Когда этот человек приезжает, я показываю ему стопку тарелок и прошу его, не смог бы он заняться мытьем посуды. Мой друг идет к раковине и начинает прогонять посуду через конвейер операций. Сейчас я породил асинхронную работу, так же, как если бы я использовал [Callable](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html) или [Runnable](http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) из Java на пуле потоков. Если я стою сзади него и жду, пока он заончит, и ничего не делаю, то я заблокированный поток. Я породил работу, которую предстоит завершить, и делегирова ее некоторому потоку выполнения, который выполнит работу в случайный момент времени, а я в свою очередь ничего не делаю, пока другая задача не завершится – как экземпляр Java Future. Вместо того, чтобы стоять, я мог бы пойти выпить лимонад – теперь я неблокирующий (как [Scala Future](http://www.scala-lang.org/api/current/#scala.concurrent.Future) или Java 8 [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html)), но не приносящий пользу. И мой друг, вероятно, становится весьма раздражен мною. К тому же, если они не отмывают посуду быстрее меня (более быстрый процессор, например), работа не будет закончена быстрее.

Что я действительно хочу, так это закончить работу эффективнее и быстрее, используя нас двоих. На данный момент я присоединяюсь к моему другу в выполнении работы. Мой друг несет ответственность за захват посуды из стекла, полоскания его и вытирания. Потом я перехватываю тарелку у него, снова полощу и засовываю в посудомоечную машину. Теперь я неблокирующий и продуктивно выполняю задачи, но постановка работы таким образом, когда у нас есть общие ресурсы, влияет на нашу способность делать работу в оптимально. Как поток исполнения, отвественный за обработку работы, делегированную моим другом, я должен ждать каждую тарелку, которая может оттираться неопределенное количество времени в зависимости о того, как сильно она грязная (суть процессора интенсивной работы). Хуже того, мы оба вынуждены использовать кран, чтобы ополоснуть посуду, поэтому мы имеем взаимно исключающую операцию над состоянием (кран), которая должна решаться засчет коммуникации, взаимоисключающих блокировок (мьютексов) ядром компьютера. В этом заключается суть параллелизма, как правило, над общим изменяемым состоянием. Если состояние (кран) является неудовлетворенным (не используется ни одним из нас в то время, когда это требуется), мы можем быстро работать над нашими задачами. Но если есть спор (один из нас использует кран, когда второй нуждается в нем), мы останавливаемся ждать, пока другой пользуется, чтобы выполнять работу.

**Способ улучшить конкурентность – это увеличение футпринта**. Если бы мой дом был достаточно большим, чтобы иметь вторую раковину, я мог бы взять стопку посуды и пойти мыть ее незавиимо от друга, и это было бы более эфективно. Подумайте, как похожи паралельные коллекции в Java8 и Scala при использовании ForkJoinPool. Я ответвил работу взяв стопку тарелок, мой друг и я выполняем работу как доступные ядра исполнения и мы будем объединяться, когда нам понадобится собрать посуду в посудомоечную машину. Однако, как и параллельные коллекции, фазы ответвления и присоединения все еще конкурентные – мы должны разделить данные (блюда) между нами, чтобы обработать их, а также соединить данные (опять же, блюда) в трансформированную коллекцию (посудомоечная машина)

Разделение может быть дешевым или дорогим, в зависимости от того, как распределена работа. Если бы я просто захватил верхнюю половину пластин – это дешевая операция. Однако, что если я хочу распределить их между мной и другом в какой-то последовательности? Это было бы более дорогостоещей операцией. И точка присоединения может иметь аналогичные затраты в зависимости от порядка. Это приводит к росту стоимости закона Амдаля – даже если мы распараллелили работу, она все еще может занять больше времени, чем если бы мы делали ее последовательно, если время на разделение и соединение слишком высоко.

Нам нужно, чтобы наша работа была как можно параллельной, чтобы максимально увеличить эффективность обработки. Для этого нам нужно увеличить футпринт (присутствие) еще больше. Было бы идеально, если бы мы могли как-то посредничать, пытаясь выснить, кто и что делает, или если бы задачи для нас брались из общей очереди. А если у меня есть две посудомоечные машины, у меня нет разногласий в точке соединения. Это увеличение в занимаемой площади означает дополнительные расходы, но снижение в конкуренции и увеличении параллелизма означает, что моя масштабированность стала линейной – если я добавляю ольше процессоров/раковин/посудомоечных машин, я увеличиваю количество посуды, которое я могу обрабатывать (данные я могу трансформировать), с одним и тем же коэфифициентом. Значение этой повышенной масштабируемости и эффективности может оправдать увеличение стоимости оборудования для моего бизнеса.

**Подведение итогов**

В конце концов, моя конечная цель состоит в том, чтобы создать асинхронное, неблокирующее и распараллеленное выполнение с минимальными точками конкурентности. Посредничая работу, я имею дело с каждой тарелкой (или стопкой тарелок) как с сообщением, где меня не волнует, какой конвейер выполняет трансформацию (мытье посуды). В этом суть прозрачности расположения, которая придает эластичность реактивным системам, позволяя разворачивать дополнительные узлы для обработки растущих рабочих нагрузок или сворачивать узлы по мере уменьшения работы. Прозрачность расположения также поддерживает повышение устойчивости, потому что вы не заботитесь о том, что конвейер завершился неудачно, а посуда, которая никогда не закончит успешно очистку, может быть обработанна снова. И пользовательский опыт может быть уверен сразу в эластичности в условиях нагрузки и в устойчивости к сбоям любого рода в системе. Таким образом, Message Driven архитектуры есть суть реактивных приложений.

**Микросервисы**

**определение нового понятия в архитектуре**

*Термин «Microservice Architecture» получил распространение в последние несколько лет как описание способа дизайна приложений в виде набора независимо развертываемых сервисов. В то время как нет точного описания этого архитектурного стиля, существует некий общий набор характеристик: организация сервисов вокруг бизнес-потребностей, автоматическое развертывание, перенос логики от шины сообщений к приемникам (endpoints) и децентрализованный контроль над языками и данными.*

«Микросервисы» — еще один новый термир на шумных улицах разработки ПО. И хотя мы обычно довольно настороженно относимся ко всем подобным новинкам, конкретно этот термин описывает стиль разработки ПО, который мы находим все более и более привлекательным. За последние несколько лет мы видели множество проектов, использующих этот стиль, и результаты со сих пор были весьма позитивными. Настолько, что для большинства наших коллег этот стиль становится основным стилем разработки ПО. К сожалению, существует не так много информации, которая описывает, чем же являются микросервисы и как применять их.

Если коротко, то архитектурный стиль микросервисов — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP. Эти сервисы построены вокруг бизнес-потребностей и развертываются независимо с использованием полностью автоматизированной среды. Существует абсолютный минимум централизованного управления этими сервисами. Сами по себе эти сервисы могут быть написаны на разных языках и использовать разные технологии хранения данных.

Для того, чтобы начать рассказ о стиле микросервисов, лучше всего сравнить его с монолитом (monolithic style): приложением, построенном как единое целое. Enterprise приложения часто включают три основные части: пользовательский интерфейс (состоящий как правило из HTML страниц и javascript-а), база данных (как правило реляционной, со множеством таблиц) и сервер. Серверная часть обрабатывает HTTP запросы, выполняет доменную логику, запрашивает и обновляет данные в БД, заполняет HTML страницы, которые затем отправляются браузеру клиента. Любое изменение в системе приводит к пересборке и развертыванию новой версии серверной части приложения.

Монолитный сервер — довольно очевидный способ построения подобных систем. Вся логика по обработке запросов выполняется в единственном процессе, при этом вы можете использовать возможности вашего языка программирования для разделения приложения на классы, функции и namespace-ы. Вы можете запускать и тестировать приложение на машине разработчика и использовать стандартный процесс развертывания для проверки изменений перед выкладыванием их в продакшн. Вы можете масштабировать монолитное приложения горизонтально путем запуска нескольких физических серверов за балансировщиком нагрузки.

Монолитные приложения могут быть успешными, но все больше людей разочаровываются в них, особенно в свете того, что все больше приложений развертываются в облаке. Любые изменения, даже самые небольшие, требуют пересборки и развертывания всего монолита. С течением времени, становится труднее сохранять хорошую модульную структуру, изменения логики одного модуля имеют тенденцию влиять на код других модулей. Масштабировать приходится все приложение целиком, даже если это требуется только для одного модуля этого приложения.

Эти неудобства привели к архитектурному стилю микросервисов: построению приложений в виде набора сервисов. В дополнение к возможности независимого развертывания и масштабирования каждый сервис также получает четкую физическую границу, которая позволяет разным сервисам быть написанными на разных языках программирования. Они также могут разрабатываться разными командами.

Мы не утверждаем, что стиль микросервисов это инновация. Его корни уходят далеко в прошлое, как минимум к принципам проектирования, использованным в Unix. Но мы тем не менее считаем, что недостаточно людей принимают во внимание этот стиль и что многие приложения получат преимущества если начнут применять этот стиль.

**Свойства архитектуры микросервисов**

Мы не можем сказать, что существует формальное определение стиля микросервисов, но мы можем попытаться описать то, что мы считаем общими характеристиками приложений, использующих этот стиль. Не всегда они встречаются в одном приложении все сразу, но, как правило, каждое подобное приложение включает в себя большинство этих характеристик. Мы попробуем описать то, что мы видим в наших собственных разработках и в разработках известных нам команд.

**Разбиение через сервисы**

В течение всего срока нашего пребывания в индустрии мы видим желание строить системы путем соединения вместе различных компонент, во многом так же, как это происходит в реальном мире. За последние пару десятков лет мы видели большой рост набора библиотек, используемых в большистве языков программирования.

Говоря о компонентах, мы сталкиваемся с трудностями определения того, что такое компонент. Наше определение такого: компонент — это единица программного обеспечения, которая может быть независимо заменена или обновлена.

Архитектура микросервисов использует библиотеки, но их основной способ разбиения приложения — путем деления его на сервисы. Мы определяем библиотеки как компоненты, которые подключаются к программе и вызываются ею в том же процессе, в то время как сервисы — это компоненты, выполняемые в отдельном процессе и коммуницирующие между собой через веб-запросы или удаленный вызов процедур (RPS).

Главная причина использования сервисов вместо библиотек — это независимое развертывание. Если вы разрабатываете приложение, состоящее из нескольких библиотек, работающих в одном процессе, любое изменение в этих библиотеках приводит к переразвертыванию всего приложения. Но если ваше приложение разбито на несколько сервисов, то изменения, затрагивающие какой-либо из них, потребуют переразвертывания только изменившегося сервиса. Конечно, какие-то изменения будут затрагивать интерфейсы, что, в свою очередь, потребует некоторой координации между разными сервисами, но цель хорошей архитектуры микросервисов — минимизировать необходимость в такой координации путем установки правильных границ между микросервисами, а также механизма эволюции контрактов сервисов.

Другое следствие использования сервисов как компонент — более явный интерфейс между ними. Большинство языков программирования не имеют хорошего механизма для объявления [Published Interface](http://martinfowler.com/bliki/PublishedInterface.html). Часто только документация и дисциплина предотвращают нарушение инкапсуляции компонентов. Сервисы позволяют избежать этого через использование явного механизма удаленных вызовов.

Тем не менее, использование сервисов подобным образом имеет свои недостатки. Удаленные вызовы работают медленнее, чем вызовы в рамках процесса, и поэтому API должен быть менее детализированным (coarser-grained), что часто приводит к неудобству в использовании. Если вам нужно изменить набор ответственностей между компонентами, сделать это сложнее из-за того, что вам нужно пересекать границы процессов.

В первом приближении мы можем наблюдать, что сервисы соотносятся с процессами как один к одному. На самом деле сервис может содержать множество процессов, которые всегда будут разрабатываться и развертываться совместно. Например, процесс приложения и процесс базы данных, которую использует только это приложение.

**Организация вокруг потребностей бизнеса**

Когда большое приложение разбивается на части, часто менеджмент фокусируется на технологиях, что приводит к образованию UI команды, серверной команды и БД команды. Когда команды разбиты подобным образом, даже небольшые изменения отнимают много времени из-за необходимости кросс-командного взаимодействия. Это приводит к тому, что команды размещают любую логику на тех слоях, к которым имеют доступ. Закон Конвея (Conway's Law) в действии.

«Любая организация, которая проектирует какую-то систему (в широком смысле) получит дизайн, чья структура копирует структуру команд в этой организация»

— Melvyn Conway, 1967

Микросервисный подход к разбиению подразумевает раазбиение на сервисы в соответствии с **потребностями бизнеса**. Такие сервисы включают в себя полный набор технологий, необходимых для этой бизнес-потребности, в том числе пользовательский интерфейс, хранилице данных и любые внешние взаимодействия. Это приводит к формированию кросс-функциональных команд, имеющих полный набор необходимых навыков: user-experience, базы данных и project management.

**Насколько большими должны быть микросервисы?**

Хотя термин «Микросервис» стал популярным названием для этого архитектурного стиля, само имя приводит к чрезмерному фокусу на размере сервисов и спорам о том, что означает приставка «микро». В наших разговорах с теми, кто занимался разбиением ПО на микросервисы, мы видели разные размеры. Наибольший размер был у компаний, следовавших правилу «Команда двух пицц» (команда, которую можно накормить двумя пиццами), т.е. не более 12 человек (*прим. перев.: следуя этому правилу, я в команде должен быть один*). В других компаниях мы видели команды, в которых шестеро человек поддерживали шесть сервисов.

Это приводит к вопросу о том, есть ли существенная разница в том, сколько человек должно работать на одном сервисе. На данный момент мы считаем, что оба этих подхода к построению команд (1 сервис на 12 человек и 1 сервис на 1 человека) подходят под описание микросервисной архитектуры, но возможно мы изменим свое мнение в будущем.

Одна из компаний, организованных в этом стиле — [www.comparethemarket.com](http://www.comparethemarket.com/). Кросс-фунциональные команды отвечают за построение и функционирование каждого продукта и каждый продукт разбит на несколько отдельных сервисов, общающихся между собой через шину сообщений.

Крупные монолитные приложения тоже могут быть разбиты на модули вокруг бизнес потребностей, хотя обычно этого не происходит. Безусловно, мы рекомендуем большим командам строить монолитные приложения именно таким образом. Основная проблема здесь в том, что такие приложения имеют тенденцию к организации вокруг слишком большого количества контекстов. Если монолит охватывает множество контекстов, отдельным членам команд становится слишком сложно работать с ними из-за их большого размера. Кроме того, соблюдение модульных границ в монолитном приложении требует существенной дисциплины. Явно очерченные границы компонент микросервисов упрощает поддержку этих границ.

**Продукты, а не проекты**

Большиство компаний по разработке ПО, которые мы видим, используют проектную модель, в которой целью является разработка некой части функциональности, которая после этого считается завершенной. После завершения эта часть передается команде поддержки и проектная команда распускается.

Сторонники микросервисов сторонятся этой модели, утверждая, что команда должна владеть продуктом на протяжении всего срока его жизни. Корни этого подхода уходят к Амазону, у компании есть правило "[вы разработали, вам и поддерживать](https://queue.acm.org/detail.cfm?id=1142065)", при котором команда разработки берет полную ответственность за ПО в продакшне. Это приводит к тому, что разработчики регулярно наблюдают за тем, как их продукт ведет себя в продакшне, и больше контактируют с пользователями, т.к. им приходится брать на себя как минимум часть обязанностей по поддержке.

Мышление в терминах продукта устанавливает связь с потребностями бизнеса. Продукт — это не просто набор фич, которые необходимо реализовать. Это постоянные отношения, цель которых — помочь пользователям увеличить их бизнес-возможности.

Конечно, этого можно также достичь и в случае с монолитным приложением, но высокая гранулярность сервисов упрощает установку персональных отношений между разработчиками сервиса и его пользователями.

**Умные приемники и глупые каналы передачи данных**

При выстраивании коммуникаций между процессами мы много раз были свидетелями того, как в механизмы передачи данных помещалась существенная часть логики. Хорошим примером здесь является Enterprise Service Bus (ESB). ESB-продукты часто включают в себя изощренные возможности по передаче, оркестровке и трансформации сообщений, а также применению бизнес-правил.

Комьюнити микросервисов предпочитает альтернативный подход: умные приемники сообщений и глупые каналы передачи. Приложения, построенные с использованием микросервисной архитектуры, стремятся быть настолько незавимыми (decoupled) и сфокусировнными (cohesive), насколько возможно: они содержат собственную доменную логику и выступают больше в качестве фильтров в классическом Unix-овом смысле — получают запросы, применяют логику и отправляют ответ. Вместо сложных протоколов, таких как WS-\* или BPEL, они используют простые REST-овые протоколы.

Два наиболее часто используемых протокола — это HTTP запросы через API ресурса и легковесный месседжинг. Лучшее выражение первому дал [Ian Robinson](http://www.amazon.com/gp/product/0596805829): «Be of the web, not behind the web».

Команды, практикующие микросервисную архитектуру, используют те же принципы и протоколы, на которых построена всемирная паутина (и, по сути, Unix). Часто используемые ресурсы могут быть закешированы с очень небольшими усилиями со стороны разработчиков или IT-администраторов.

Второй часто используемый инструмент коммуникации — легковесная шина сообщений. Такая инфраструктура как правило не содержит доменной логики — простые реализации типа RabbitMQ или ZeroMQ не делают ничего кроме предоставления асинхронной фабрики. Логика при этом существует на концах этой шины — в сервисах, которые отправляют и принимают сообщения.

В монолитном приложении компоненты работают в одном процессе и коммуницируют между собой через вызов методов. Наибольшая проблема в смене монолита на микросервисы лежит в изменении шаблона коммуникации. Наивное портирование один к одному приводит к «болтливым» коммуникациям, которые работают не слишком хорошо. Вместо этого вы должны уменьшить количество коммуникаций между модулями.

**Децентрализованное управление**

Одним из следствий централизованного управления является тенденция к стандартизации используемых платформ. Опыт показывает, что такой подход слишком сильно ограничивает выбор — не всякая проблема является гвоздем и не всякое решение является молотком. Мы предпочитаем использовать правильный инструмент для каждой конкретной работы. И хотя монолитные приложения тоже в некоторых случаях могут быть написаны с использованием разных языков, это не является стандартной практикой.

**Microservices and SOA**

Когда мы разговариваем о микросервисах, обычно возникает вопрос о том, не является ли это обычным Service Oriented Architecture (SOA), который мы видели десять лет назад. В этом вопросе есть здравое зерно, т.к. стиль микросервисов очень похож на то, что продвигают некоторые сторонники SOA. Проблема, тем не менее, в том, что [термин SOA имеет сликом много разных значений](http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html) и, как правило, то, что люди называют «SOA» существенно отличается от стиля, описанного здесь, обычно из-за чрезмерного фокуса на ESB, используемом для интеграции монолитных приложений.

В частности, мы видели так много неудачных реализаций SOA (начиная с тенденции прятать сложность за ESB, заканчивая провалившимися инциативами длительностью несколько лет, которые стоили миллионы долларов и не принесли никакой пользы), что порой слишком сложно абстрагироваться от этих проблем.

Безусловно, многие практики, используемые в микросервисах, пришли из опыта интеграции сервисов в крупных организациях. Шаблон [Tolerant Reader](http://martinfowler.com/bliki/TolerantReader.html) — один из примеров. Другой пример — использование простых протоколов — возник как реакция на централизованные стандарты, сложность которых просто [захватывает дух](http://wiki.apache.org/ws/WebServiceSpecifications).

Эти проблемы SOA привели к тому, что некоторые сторонники микросервисов отказываются от термина «SOA», хотя другие при этом считают микросервисы одной из форм SOA, или, возможно, правильной реализацией SOA. В любом случае, тот факт, что SOA имеет разные значения, означает, что полезно иметь отдельный термин для обозначения этого архитектурного стиля.

Разбивая монолит на сервисы, мы имеем выбор, как построить каждый из них. Хотите использовать Node.js для простых страничек с отчетами? Пожалуйста. C++ для real-time приложений? Отлично. Хотите заменить БД на ту, которая лучше подходит для операций чтения вашего компонента?

Конечно, только потому что вы можете делать что-то, не значит что вы должны это делать. Но разбиение системы подобным образом дает вам возможность выбора.

Команды, разрабатывающие микросервисы, также предпочитают иной подход к стандартизации. Вместо того, чтобы использовать набор предопределенных стандартов, написанных кем-то, они предпочитают идею построения полезных инструментов, которые остальные девелоперы могут использовать для решения похожих проблем. Эти инструменты как правило вычленены из кода одного из проектов и расшарены между разными командами, иногда используя при этом модель внутреннего опен-сорса. Теперь, когда git и github стали де-факто стандартной системой контроля версий, опен-сорсные практики становятся все более и более популярными во внутренних проектах компаний.

Netflix — хороший пример организации, которая следует этой философии. Расшаривание полезного и, более того, протестированного на боевых серверах кода в виде библиотек побуждает остальных разработчиков решать схожие проблемы схожим путем, оставляя тем не менее возможность выбора другого подхода при необходимости. Общие библиотеки имеют тенденцию быть сфокусированными на общих проблемах, связанных с хранением данных, межпроцессорным взаимодействием и автоматизацией инфраструктуры.

Комьюнити микросервисов ценит сервисные контракты, но не любит оверхеды и поэтому использует различные пути управления этими контрактами. Такие шаблоны как [Tolerant Reader](http://martinfowler.com/bliki/TolerantReader.html) и [Consumer-Driven Contracts](http://martinfowler.com/articles/consumerDrivenContracts.html) часто используются в микросервисах, что позволяет им эволюционировать независимо. Проверка Consumer-Driven контрактов как часть билда увеличивает уверенность в правильности функционирование сервисов. Мы знаем команду из Австралии, которая использует этот подход для проверки контрактов. Это стало частью их процесса сборки: сервис собирается только до того момента, который удовлетворяет требованиям контракта — элегантный способ обойти диллему YAGNI.

**Множество языков, множество возможностей**

Рост платформы JVM — один из последних примеров смешивания языков в рамках единой платформы. Переход к более высокоуровневым языкам для получения преимуществ, связанных с использованием высокоуровневых абстракций, был распространенной практикой в течение десятилетий. Точно так же, как и переход «к железу» для написания высокопроизводительного кода.

Тем не менее, множество монолитных приложений не требуют такого уровня оптимизации производительности и высокоуровневых возможностей DSL-подобных языков. Вместо этого, монолиты как правило используют единый язык и склонны к ограничению количества используемых технологий.

Пожалуй наивысшая точка в практике децентрализованного управления — это метод, популизированный Амазоном. Команды отвечают за все аспекты ПО, которое они разрабатывают, включая поддержку его в режиме 24/7. Подобная деволюция уровня ответственности совершенно точно не является нормой, но мы видим все больше и больше компаний, передающий ответственность командам разработчиков. Netflix — еще одна компания, практикующая это. Пробуждение в 3 часа ночи — очень сильный стимул к тому, чтобы уделять большое внимание качеству написанного кода.

**Децентрализованное управление данными**

Децентрализованное управление данными предстает в различном виде. В наиболее абстрактном смысле это означает, что концептуальная модель мира у разных систем будет отличаться. Это обычная проблема, возникающая при интеграции разных частей больших enterprise-приложений: точка зрения на понятие «Клиент» у продажников будет отличаться от таковой у команды техподдержки. Некоторые атрибуты «Клиента» могут присутствовать в контексте продажников и отсутствовать в контексте техподдержки. Более того, атрибуты с одинаковым названием могут иметь разное значение.

**Стандарты, проверенные в бою, против названных стандартов**

Команды, использующие микросервисную архитектуру, склонны избегать жестких стандартов, установленных группами системных архитекторов. Они также склонны использовать и даже продвигать открытые стандарты типа HTTP и ATOM.

Ключевое отличие в том, как эти стандарты разрабатываются и как они проводятся в жизнь. Стандарты, управляемые группами вроде IETF, становятся стандартами только тогда, когда находятся несколько реализаций в успешных open-source проектах.

Это отличает их от стандартов в корпоративном мире, которые часто разрабатываются группами людей с небольшим опытом реальной разработки или имеют слишком сильное влияние, оказываемое вендорами.

Эта проблема встречается не только у разных приложений, но также и в рамках единого приложения, особенно в тех случаях когда это приложение разделено на отдельные компоненты. Эту проблему хорошо решает понятие [Bounded Context](http://martinfowler.com/bliki/BoundedContext.html) из Domain-Driven Design (DDD). DDD предлагает делить сложную предметную область на несколько контекстов и мапить отношения между ними. Этот процесс полезен как для монолитной, так и для микросервисной архитектур, но между сервисами и контекстами существует естественная связь, которая помогает прояснять и поддерживать границы контекстов.

Кроме децентрализации принятия решений о моделировании предметной области, микросервисы также способствуют децентрализации способов хранения данных. В то время как монолитные приложения склонны к использованию единственной БД для хранения данных, компании часто предпочитают использовать единую БД для целого набора приложений. Такие решения, как правило, вызваны моделью лицензирования баз данных. Микросервисы предпочитают давать возможность каждому сервису управлять собственной базой данных: как создавать отдельные инстансы общей для компании СУБД, так и использовать нестандартные виды баз данных. Этот подход называется [Polyglot Persistence](http://martinfowler.com/bliki/PolyglotPersistence.html). Вы также можете применять Polyglot Persistence в монолитных приложениях, но в микросервисах такой подход встречается чаще.

Децентрализация ответственности за данные среди микросервисов оказывает влияние на то, как эти данные изменяются. Обычный подход к изменению данных заключается в использовании транзакций для гарантирования консистентности при изменении данных, находящихся на нескольких ресурсах. Такой подход часто используется в монолитных приложениях.

Подобное использование транзакций гарантирует консистентность, но приводит к существенной временн***о***й зависимости (temporal coupling), которая, в свою очередь, приводит к проблемамм при работе с множеством сервисов. Распределенные транзакции невероятно сложны в реализации и, как следствие, микросервисная архитектура придает особое значению координации между сервисами [без использования транзакций](http://www.eaipatterns.com/ramblings/18_starbucks.html) с явным обозначением того, что консистентность может быть только итоговой (eventual consistency) и возникающие проблемы решаются операциями компенсации.

Управление несогласованностями подобным образом — новый вызов для многих команд разработки, но это часто соответствует практикам бизнеса. Часто компании стремятся как можно быстрее реагировать на действия пользователя и имеют процессы, позвояющие отменить действия пользователей в случае ошибки. Компромисс стоит того до тех пор, пока стоимость исправления ошибки меньше стоимости потерь бизнеса при использовании сценариев, гарантирующих консистентность.

**Автоматизация инфраструктуры**

Техники автоматизации инфраструктуры сильно эволюционировали за последние несколько лет. Эволюция облака в целом и AWS в частности уменьшила операционную сложность построения, разворачивания и функционирования микросервисов.

Множество продуктов и систем, использующих микросервисную архитектуру, были построены командами с обширным опытом в [Continuous Delivery](http://martinfowler.com/bliki/ContinuousDelivery.html) и [Continuous Integration](http://martinfowler.com/articles/continuousIntegration.html). Команды, строящие приложения подобнымм образом, интенсивно используют техники автоматизации инфраструктуры. Это проиллюстрировано на картинке ниже.

Так как эта статья не про Continuous Delivery, мы уделим внимание лишь паре его ключевых моментов. Мы хотим получать как можно больше уверенности в том, что наше приложение работает, поэтому мы запускаем множество автоматических тестов. Для выполнения каждого шага автоматического тестирования приложение разворачивается в отдельной среде, для чего используется автоматического развертывание (automated deployment).

**Делай проще, чтобы делать правильно**

Одним из побочных эффектов автоматизации процесса развертывания является создание удобных инструментов для помощи разработчикам и администраторам (operations folk). Инструменты для управления кодом, развертывания простых сервисов, мониторинга и логирования сейчас довольно распространены. Возможно наилучший пример, который можно найти в сети, — это набор open source инструментов от [Netflix](http://netflix.github.io/), но существуют и другие, к примеру [Dropwizard](https://dropwizard.github.io/dropwizard/), который мы довольно интенсивно используем.

После того как вы инвестировали время и деньги в автоматизацию процесса развертывания монолита, развертывание большего количества приложений (сервисов) уже не видится таким пугающим. Вспомните, что одна из целей Continuous Delivery — это сделать развертывание скучным, так что одно это приложение или три не имеет большого значения.

Другая область, где команды используют интенсивную автоматизацию инфраструктуры, — это управление микросервисами в продакшне. В отличие от процесса развертывания, который, как описано выше, у монолитных приложений не сильно отличается от такового у микросервисов, их способ фунционирования может существенно различаться.

**Проектирование под отказ**

Следствием использования сервисов как компонентов является необходимость проектирования приложений так, чтобы они могли работать при отказе отдельных сервисов. Любое обращение к сервису может не сработать из-за его недоступности. Клиент должен реагировать на это настолько терпимо, насколько возможно. Это является недостатоком микросервисов по сравнению с монолитом, т.к. это вносит дополнительную сложность в приложение. Как следствие, команды микросервисов постоянно думают на тем, как недоступность сервисов должна влиять на user experience. [Simian Army](https://github.com/Netflix/SimianArmy) от Netflix искуственно вызывает (симулирует) отказы сервисов и даже датацентров в течение рабочего дня для тестирования отказоустойчивости приложения и служб мониторинга.

Подобный вид автоматического тестирования в продакшне позволяет сэмулировать стресс, который ложится на администраторов и часто приводит к работе по выходным. Мы не хотим сказать, что для монолитных приложений не могут быть разработаны изощренные системы мониторинга, только то, что такое встречается реже.

Так как сервисы могут отказать в любое время, очень важно иметь возможность быстро обнаружить неполадки и, если возможно, автоматически восстановить работоспособность сервиса. Микросервисная архитектура делает большой акцент на мониторинге приложения в режиме реального времени, проверке как технических элементов (например, как много запросов в секунду получает база данных), так и бизнес-метрик (например, как много заказов в минуту получает приложение). Семантический мониторинг может предоставить систему раннего предупреждения проблемных ситуаций, позволяя команде разработке подключиться к исследованию проблемы на самых ранних стадиях.

Это особенно важно с случае с микросервисной архитектурой, т.к. разбиение на отдельные процессы и [коммуникация через события](http://martinfowler.com/eaaDev/EventCollaboration.html) приводит к неожиданному поведению. Мониторинг крайне важен для выявления нежелательных случаев такого поведения и быстрого их устранения.

**Синхронные вызовы считаются опасными**

Каждый раз когда вы имеете набор синхронных вызовов между сервисами, вы сталкиваетесь с эффектом мультипликации времени простоя (downtime). Время простоя вашей системы становится произведением времени простоя индивидуальных компонент системы. Вы сталкиваетесь с выбором: либо сделать ваши вызовы асинхронными, либо мириться с простоями. К примеру, в [www.guardian.co.uk](http://www.guardian.co.uk/) разработчики ввели простое правило — один синхронный вызов на один запрос пользователя. В Netflix же вообще все API являются асинхронными.

Монолиты могут быть построены так же прозначно, как и микросервисы. На самом деле, так они и должны строиться. Разница в том, что знать, когда сервисы, работающие в разных процессах, перестали корректно взаимодействовать между собой, намного более критично. В случае с библиотеками, расположенными в одном процессе, такой вид прозрачности скорее всего будет не так полезен.

Команды микросервисов, как правило, создают изощренные системы мониторинга и логирования для каждого индивидуального сервиса. Примером может служить консоль, показывающая статус (онлайн/офлайн) сервиса и различные технические и бизнес-метрики: текущая пропускная способность, время обработки запроса и т.п.

**Эволюционный дизайн**

Те, кто практикует микросервисную архитектуру, обычно много работали с эволюционным дизайном и рассматривают декомпозицию сервисов как дальнейшую возможность дать разработчикам контроль над изменениями (рефакторингом) их приложения без замедления самого процесса разработки. Контроль над изменениями не обязательно означает уменьшение изменений: с правильным подходом и набором инструментов вы можно делать частые, быстрые, хорошо контролируемые изменения.

Каждый раз когда вы пытаетесь разбить приложение на компоненты, вы сталкиваетесь с необходимостью принять решение, как именно делить приложение. Есть ли какие-то принципы, указывающие, как наилучшим способом «нарезать» наше приложение? Ключевое свойство компонента — это независимость его замены или обновления, что подразумевает наличие ситуаций когда его можно переписать с нуля без затрагивания взаимодействующих с ним компонентов. Многие команды разработчиков идут еще дальше: они явным образом планируют, что множество сервисов в долгосрочной перспективе не будет эволюционировать, а будут просто выброшены на свалку.

Веб-сайт Guardian — хороший пример приложения, которое было спроектировано и построено как монолит, но затем эволюционировало в сторону микросервисов. Ядро сайта все еще остается монолитом, но новые фичи добавляются путем построения микросервисов, которые используют API монолита. Такой подход особенно полезен для функциональности, которая по сути своей является временной. Пример такой функциональности — специализированные страницы для освещения спортивных событий. Такие части сайта могут быть быстро собраны вместе с использованием быстрых языков программирования и удалены как только событие закончится. Мы видели похожий подход в финансовых системах, где новые сервисы добавлялись под открывшиеся рыночные возможности и удалялись через несколько месяцев или даже недель после создания.

Такой упор на заменяемости — частный случай более общего принципа модульного дизайна, который заключается в том, что модульность определяется скоростью изменения функционала. Вещи, которые изменяются вместе, должны храниться в одном модуле. Части системы, изменяемые редко, не должны находиться вместе с быстроэволюционирующими сервисами. Если вы регулярно меняете два сервиса вместе, задумайтесь над тем, что возможно их следует объединить.

Помещение компонент в сервисы добавляет возможность более точного (granular) планирования релиза. С монолитом любые изменения требуют пересборки и развертывания всего приложения. С микросервисами вам нужно развернуть (redeploy) только те сервисы, что изменились. Это позволяет упростить и ускорить процесс релиза. Недостаток такого подхода в том, что вам приходится волноваться насчет того, что изменения в одном сервисе сломают сервисы, обращающиеся к нему. Традиционный подход к интеграции заключается в том, чтобы решать такие проблемы путем версионности, но микросервисы предпочитают использовать версионность только в случае [крайней необходимости](http://martinfowler.com/articles/enterpriseREST.html#versioning). Мы можем избежать версионности путем проектирования сервисов так, чтобы они были настолько толерантны к изменениям соседних сервисов, насколько возможно.

**За микросервисами будущее?**

Наша основная цель при написании этой статьи заключалась в том, чтобы объяснить основные идеи и принципы микросервисной архиктуры. Мы считаем, что микросервисный стиль — важная идея, стоящая рассмотрения для enterprise приложений. Не так давно мы разработали несколько систем используя этот стиль и знаем несколько других команд, которые используют этот подход.

Известные нам пионеры этого архитектурного стиля — это такие компании как Amazon, Netflix, The Guardian, the UK Government Digital Service, realestate.com.au, Forward и comparethemarket.com. Конференции 2013 года были полны примеров команий, движущихся в направлении, которое можно классифицировать как микросервисы, например, Travis CI. К тому же, существует множество организаций, которые уже давно используют то, что мы называем микросервисами, но не используют это название. (Часто это называется SOA, хотя, как мы уже говорили, SOA может являться в самых разных и, зачастую, противоречивых формах.)

Несмотря на весь этот положительный опыт, мы не утверждаем, что микросервисы — это будущее проектирования ПО. И хотя наш опыт пока что весьма позитивен по сравнению с опытом использования монолитной архитектуры, мы подходим осознанно к тому факту, что прошло еще недостаточно времени для того, чтобы выносить такое суждение.

Часто настоящие последствия ваших архитектурных решений становятся видно только спустя несколько лет после того, как вы сделали их. Мы видели проекты, в которых хорошие команды с сильным стремлением к модульности разработали монолитные приложения, полностью «прогнившие» по прошествию нескольких лет. Многие считают, что такой результат менее вероятен в случае с микросервисами, т.к. границы между сервисами являются физическими и их сложно нарушить. Тем не менее, до тех пока мы не увидим достаточного количества проверенных временем систем, использующих этот подход, мы не можем с уверенностью утверждать, насколько микросервисная архитектура является зрелой.

Определенно существуют причины, по которым кто-то может считать микросервисную архитектуру недостаточно зрелой. Успех любых попыток построить компонентную систему зависит от того, насколько хорошо компоненты подходят приложению. Сложно понять где именно должны лежать границы компонентов. Эволюционный дизайн осознает сложности проведения правильных границ и важность легкого их изменения. Когда ваши компоненты являются сервисами, общающимися между собой удаленно, проводить рефакторинг намного сложнее, чем в случае с библиотеками, работающими в одном процессе. Перемещение кода между границами сервисов, изменение интерфейсов должны быть скоординированы между разными командами. Необходимо добавлять слои для поддержки обратной совместимости. Все это также усложняет процесс тестирования.

Еще одна проблема состоит в том, что если компоненты не подобраны достаточно чисто, происходит перенос сложности из компонент на связи между компонентами. Создается ложное ощущение простоты отдельных компонент, в то время как вся сложность находится в местах, которые труднее контролировать.

Также существует фактор уровня команды. Новые техники как правило принимаются более сильными командами, но техники, которые являются более эффективными для более сильных команд, необязательно являются таковыми для менее сильных групп разработчиков. Мы видели множество случаев, когда слабые команды разрабатывали запутанные, неудачные архитектуры монолитных приложений, но пройдет время прежде чем мы увидим чем это закончится в случае с микросервисной архитектурой. Слабые команды всегда создают слабые системы, сложно сказать улучшат ли микросервисы эту ситуацию или ухудшат.

Один из разумных аргументов, которые мы слышали, состоит в том, что вам не следует начинать разработку с микросервисной архитектуры. Начните с монолита, сохраняйте его модульным и разбейте на микросервисы когда монолит станет проблемой. (И все же этот совет не является идеальным, т.к. хорошие интерфейсы для сообщения внутри процесса не являются таковыми в случае с межсервисным сообщением.)

Итого, мы пишем это с разумным оптимизмом. К этому моменту мы видели достаточно примеров микросервисного стиля чтобы осознавать, что он является стоящим путем развития. Нельзя сказать с уверенностью к чему это приведет, но одна из особенностей разработки ПО заключается в том, что нам приходится принимать решения на основе той, зачастую неполной, информации, к который мы имеет доступ в данный момент.