Проектирование распределённых приложений, стратегические вопросы

Юрий Литвинов
yurii.litvinov@gmail.com

25.04.2022

1. Архитектурные стили распределённых систем

В этой лекции поговорим о «стратегических» аспектах проектирования распределённых (прежде всего, больших облачных) приложений. Начнём с типичных для них архитектурных стилей. Все они в целом строятся либо согласно слоистому стилю, либо согласно разным подвидам событийно-ориентированного стиля, однако в реальной жизни используются обычно более специфичные стили, о них-то и поговорим.

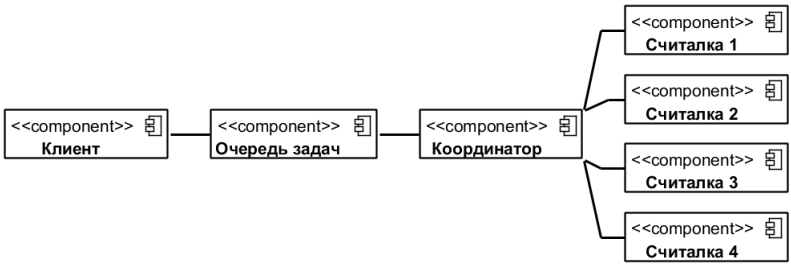
Далее архитектурные стили упорядочены по возрастанию «обычности» для современных облачных приложений (по крайней мере, по мнению автора). Изложение ведётся в основном по архитектурным гайдлайнам платформы Microsoft Azure[1] с аккуратно вырезанными вендор-специфичными штуками, тем не менее, данный обзор не претендует на полноту.

1.1. Big Compute

Архитектурный стиль «Большие вычисления» («Big Compute») предназначен для решения задач, требующих массовой параллельности. Лекционный курс «Параллельное программирование» на матмехе, как говорят, начинается со слов «представим, что у нас есть компьютер с бесконечным количеством процессоров, посмотрим, что на нём можно посчитать» — это и есть типовая ситуация применения Big Compute. Типичные задачи, решаемые в таком стиле — это прежде всего физические или химические расчёты, симуляция, различного рода моделирование, решение огромных дифуров и т.п., то есть задачи, которыми на матмехе активно занимается кафедра параллельных алгоритмов, и которые обычно требуют суперкомпьютер.

Архитектурно подобные системы устроены довольно просто:

[1] Azure Architecture Center, URL: https://docs.microsoft.com/en-us/azure/architecture/ (дата обращения: 11.12.2021).



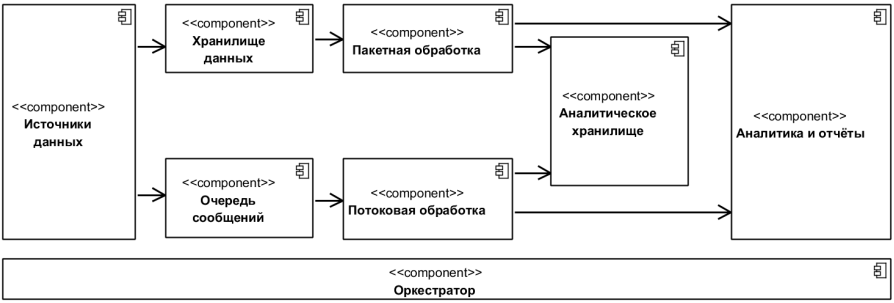
Есть клиент, снабжающий систему данными и запросами, есть очередь задач, куда запросы попадают от клиент, и есть координатор (на самом деле, самая сложная часть системы), который берёт задачи из очереди, делит на подзадачи и раскидывает по вычислительным узлам. Каждый вычислительный узел обычно устроен довольно просто, но интересно то, что их много — от десятков до десятков тысяч. При этом вычисления бывают хорошо параллелизуемыми (в идеале — представимыми в виде Map-Reduce-вычисления), тогда части задачи можно раскидать по узлам и они могут считать, никак не координируясь друг с другом. Когда все узлы закончат обработку, координатор собирает результаты, агрегирует их (возможно, задействуя ещё узлы) и выдаёт ответ. В этом случае каждый узел должен обладать большими ресурсами (прежде всего, процессором, но может потребоваться и оперативная память), но требований к сети почти нет.

Бывают, однако, параллельные задачи, требующие активной координации между узлами. В этом случае требуется также высокопроизводительная сеть, и тогда вычислительные узлы по сути соединены в вычислительный кластер, хоть и могут физически предоставляться облачным провайдером просто из каких-то доступных ресурсов.

Этот архитектурный стиль довольно специфичен, поскольку вычислительно сложные задачи в современном мире относительно редки (зато если вы умеете такое делать, можете просить любую зарплату — для этого на матмехе и учат дифурам и матфизике). Кроме того, требуется «embarrassingly parallel»-задача (что обычно переводится на русский как «чрезвычайно параллельная», но английский термин более точен — настолько параллельная, что даже стыдно), иначе выгоды от параллелизма будет немного. Кроме того, требуются весьма продвинутые и дорогие ресурсы, которые у современных провайдеров облачной инфраструктуры типа AWS или Azure, конечно, есть, но стоить их аренда будет немало (а купить себе небольшой суперкомпьютер и содержать его будет ещё в разы дороже). Однако для таких задач зачастую просто не существует других способов решения, кроме как грубой силой.

1.2. Big Data

Всеми любимые большие данные — это обратная сторона Big Compute, когда вычисления не сложны, просто их очень много. Архитектурный стиль для обработки этих данных применяется, когда они не лезут в обычную СУБД, и структурно на самом деле похож на Big Compute, особенно если данные надо обрабатывать «на лету»:



В любом случае, данные поступают в систему извне, из какого-либо источника (обычно многих разных), где они могут быть представлены в виде, не очень пригодном для обработки, поэтому надо их сначала импортировать. Дальше возможны два варианта:

- данные надо обрабатывать на лету — например, банковские транзакции или показания датчиков; тогда они обычно все кидаются в очередь сообщений (например, Apache Kafka), откуда их параллельно вычитывает несколько потоковых обработчиков (очередь служит и буфером, и балансировщиком нагрузки), обработанные данные попадают в аналитическое хранилище для дальнейшей работы с ними или в отчёты, выдаваемые пользователю (или другим системам);

- данные можно обрабатывать с регулярными интервалами — например, формировать сводки за день; тогда вместо очереди сообщений может использоваться распределённое хранилище, куда данные импортируются из источников, и оттуда они время от времени вычитываются обработчиками (опять-таки, возможно, параллельно).

Поскольку данных заведомо много, требуются механизмы их распределённого хранения и обработки. Писать их с нуля крайне не рекомендуется, потому что простую систему написать, конечно, можно быстро, но как только появляются требования оптимальности загрузки ресурсов, отказоустойчивости и безопасности, внезапно начинает требоваться продвинутая математическая и алгоритмическая теория, которую лучше доверить профессионалам. Примеры готовых решений — Apache Hadoop (с его распределённой системой хранения данных HDFS и распределённой системой обработки MapReduce), Apache Spark (который поновее и побыстрее за счёт хранения данных в памяти). Hadoop хорош для пакетной обработки, Spark тяжелее в настройке и эксплуатации, но хорош для обработки в реальном времени.

Хранение данных тоже не так просто. В традиционных СУБД данные должны соответствовать некоторой схеме, чтобы их можно было даже хранить, однако архитектурный стиль «Big Data» говорит, что это плохо — необходимость конвертации к единому формату может переусложнить систему и потенциально стать узким местом в плане производительности. Поэтому рекомендуют пользоваться принципом Schema-on-read — хранить данные в том виде, в котором они пришли (сырые логи, текстовые документы, таблицы и т.п.), и преобразовывать их в нужный формат уже при чтении для обработки. Это позволит разным обработчикам использовать разные форматы, при этом пользуясь одним общим распределённым хранилищем данных. Для такого хранилища слабоструктурированных или неструктурированных данных есть модный термин «Data Lake», и ему часто

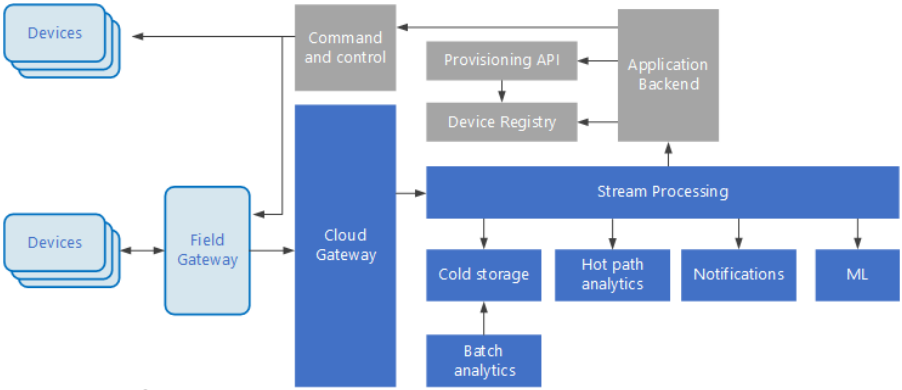
противопоставляют термин «Data Swamp» — когда данные в хранилище есть, но никто не знает, как ими пользоваться, чтобы извлечь что-то полезное.

Вполне возможно, что хранилище данных само может обеспечить предварительную обработку данных (фильтрацию, преобразование, некоторую агрегацию), тогда этим надо пользоваться. Традиционный подход к обработке данных — Extract, Transform, Load (ETL), вместо этого рекомендуется схема Transform, Extract, Load (то есть обработка на стороне хранилища — того же Apache Hadoop, извлечение обработанных данных и загрузка их в аланитическое хранилище).

Если обработка выполняется в пакетном режиме, полезный приём — это делить входные данные по интервалам обработки. Например, если метрики считаются по логам раз в день, настройте систему логирования так, чтобы она раз в день делала ротацию логов[2], и отправляйте на обработку как раз тот кусок данных, который должен быть обработан за один раз. Это несколько упрощает анализ и, если что-то пойдёт не так, упрощает локализацию проблемы.

Ещё один важный практический совет — если считается какая-то агрегатная статистика, личные данные должны удаляться как можно раньше (в идеале — никогда не покидать сам источник данных). Несмотря на все усилия по обеспечению безопасности, данные украдут, и если от этого никто не пострадает, все будут только в выигрыше.

Вот пример применения такого подхода, типичная архитектура систем «интернета вещей»:



© https://github.com/MicrosoftDocs/architecture-center/blob/main/docs/guide/architecture-styles/big-data.md

Устройства (например, охранные датчики, медицинские устройства) передают свои показания на Field Gateway (располагающийся близко к самим устройствам), он выполняет базовую обработку и агрегацию и отправляет данные на Cloud Gateway, находящийся уже в центре обработки данных (если каждое устройство будет слать свои показания прямо в облако, даже облако с этим может не справиться, ну или это будет стоить сказочных денег). Оттуда данные попадают в систему потоковой обработки, которая может делать разные вещи — сохранять данные в «холодное хранилище» для последующей пакетной обработки

[2] Общая практика для систем логирования, когда по достижению определённого размера или прошествию определённого времени лог архивируется и на его месте заводится новый. Например, log4j и log4net так, конечно, умеют — см. RollingFileAppender.

(считать разные статистики, например), анализ в реальном времени, посылать нотификации или скармливать данные разным алгоритмам машинного обучения (например, для детекции аномалий).

Также возможно, хотя и не обязательно, что есть и обратный поток, от облака к устройствам (показано серым на диаграмме). Результаты потоковой обработки попадают в бэкенд, который передаёт команды управления на устройства (возможно, опять-таки через Field Gateway). К тому же, бэкенд отвечает за регистрацию новых устройств и хранение информации об устройствах.

1.3. Событийно-ориентированная архитектура

Событийно-ориентированный стиль, конечно, также применяется для обработки данных в распределённых (в т.ч. облачных) системах. Однако в силу принципиальной ненадёжности сети издатель и подписчики редко связаны напрямую, между ними обычно находится событийная шина. В событийную шину может писать сколько угодно издателей и обработкой событий оттуда может заниматься сколько угодно подписчиков. Стиль архитектурно простой, и существующие решения позволяют сделать такие системы очень производительными, поэтому он часто применяется для обработки событий в реальном времени:

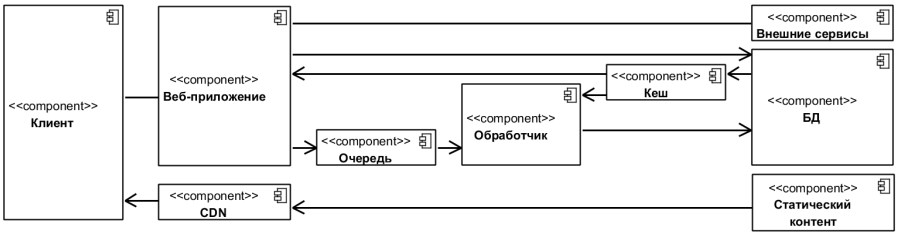
Принципиально моделей обработки событий две.

- Издатель/подписчик — когда сообщение попадает в шину, его забирает и обрабатывает один из подписчиков. В качестве шины может выступать очередь сообщений, имеющая такую семантику, например, RabbitMQ (хотя надо понимать, что их бывает много, например, ZeroMQ более легковесна и поэтому может с успехом применяться в IoT).

- Event Sourcing — когда сообщение попадает в шину и остаётся там навсегда, чтобы кому надо могли прочитать его и обработать — в том числе, вновь подключившийся клиент всегда может проиграть всю историю сообщений с начала, чтобы получить текущую картину мира. Тут в качестве шины могут выступать распределённые логи событий, например, Apache Kafka (хоть они могут быть внешне неотличимы от очередей сообщений). Для быстрой потоковой обработки это считается более прогрессивным подходом, потому что требует меньше координации между участниками взаимодействия и работает быстрее. В этом случае обычно текущее состояние системы в явном виде не хранится, или хранится в виде снапшотов, которые не являются источником истины, а всего лишь ускоряют доступ (то есть реализуется принцип eventual consistency — каждый обработчик может иметь свою картину мира, немного запаздывающую во времени, но рано или поздно все всё вычитают и узнают).

1.4. Web-queue-worker

Web-queue-worker — это некое уточнение событийно-ориентированного стиля, применяемое прежде всего для вычислительно сложных задач, если предметная область относительно несложна и не требует детального моделирования (пакетная обработка каких-то данных, например, типа конвертации pdf-документов). Поскольку это уже «приближенный к земле» стиль разработки веб-приложений, его структура более детальна:



Клиент работает в браузере у пользователя, через него пользователь заказывает выполнение работы. *Веб-приложение* на стороне сервера поддерживает действия клиента, и принимает от него работу. Задания вместе с необходимыми данными попадают в *очередь*, откуда вычитываются *обработчиками*. Результаты обработки сохраняются в *базу данных*, откуда, возможно, через *кеш* читаются веб-приложением (и обработчиками, если им нужны данные из базы). При этом веб-приложение может пользоваться также *внешними веб-сервисами*, а клиент — получать *статический контент* (картинки, скрипты, стили, файлы) через *Content Delivery Network*.

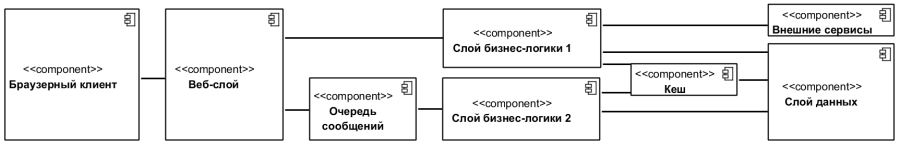
Удивительно, но очередь и обработчик в такой схеме не обязательны — если сложных вычислений нет, всё может делаться самим веб-приложением (и тогда это будет обычной трёхзвенной архитектурой). Однако если надо что-то сложное считать, то очередь очень полезна, поскольку работает буфером, смягчающим эффекты пиковой нагрузки, да ещё и балансировщиком. При этом разделение обработчика и веб-части позволяет масштабировать их независимо, так что если фронт справляется, а вычислительно сложная часть нет, можно отдельно поднять ещё несколько обработчиков, и остановить их, когда в них пропадёт нужда. Обратите внимание, что запросы принципиально выполняются асинхронно, то есть заставлять пользователя ждать, пока придёт ответ, тут пагубная практика.

Кроме того, такая архитектура позволяет эффективно использовать уже готовые технологии — например, веб-приложение на ASP.NET или Spring, очередь RabbitMQ, обработчик в виде веб-сервиса на том же ASP.NET, читающего из очереди, MariaDB в качестве БД и Redis в качестве кеша. При этом можно использовать готовые CDN для, например, React и стилей Bootstrap. Если знать, что делать, собрать такую штуку можно за выходные, почему Web-Queue-Worker и популярен в продакшене.

Однако за всё надо платить, при росте сложности предметной области начинает разрастаться либо веб-часть, либо обработчик, либо и то и другое. Очень легко пропустить тот момент, когда такое приложение превратится в большой бесформенный клубок кода, который невозможно больше поддерживать. В этом случае поможет более аккуратная декомпозиция ответственности, предполагаемая следующими архитектурными стилями.

1.5. N-звенная архитектура

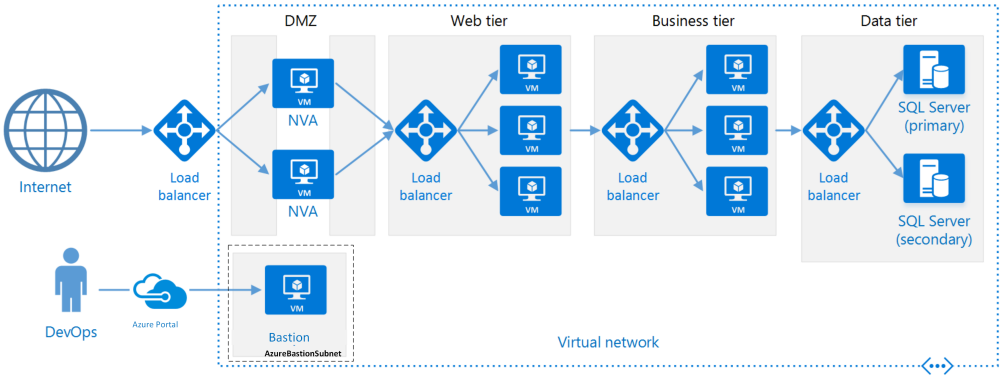
Знаменитая N-звенная архитектура, разумеется, в облачных приложениях тоже вполне используется:



Браузерный клиент, что не удивительно, работает в браузере у пользователя (это может быть одностраничное приложение на чём-нибудь вроде React или Vue, может быть и многостраничное на каком-нибудь из серверных веб-фреймворков). *Веб-слой* поддерживает клиентское приложение и, возможно, обрабатывает асинхронные запросы. Веб-слой общается со слоем *бизнес-логики*, который может быть сколь угодно сложно устроен (например, как на рисунке, их может быть два, и один из них подключен через очередь сообщений, но это не требование стиля, а только пример). Слой бизнес-логики общается с *внешними сервисами* и *слоем данных*, при этом слой данных может быть доступен на чтение через кеш (если это повысит производительность, конечно). Особенностью этого стиля является то, что компоненты приложения обычно физически размещаются на разных машинах, что помогает в масштабируемости и надёжности.

Такой архитектурный стиль хорош, когда у вас уже есть монолитное приложение, работающее на вашем любимом сервере, и вы хотите с минимумом трудозатрат перенести его куда-нибудь на AWS. Либо бизнес-логика несложная (но всё же требует некоторой структуризации, так что Web-Queue-Worker не подойдёт). Однако масштабировать при таком подходе можно только крупные куски системы, так что если ваша бизнес-логика содержит два метода, одним из которых никто никогда не пользуется, а один не справляется с нагрузкой, вы должны запустить второй экземпляр сервера бизнес-логики целиком. Ещё, как обычно, слой бизнес-логики имеет тенденцию разрастаться и превращаться в большой ком грязи.

Вот пример типичного N-звенного веб-приложения на Azure, использующего стек технологий от Microsoft:



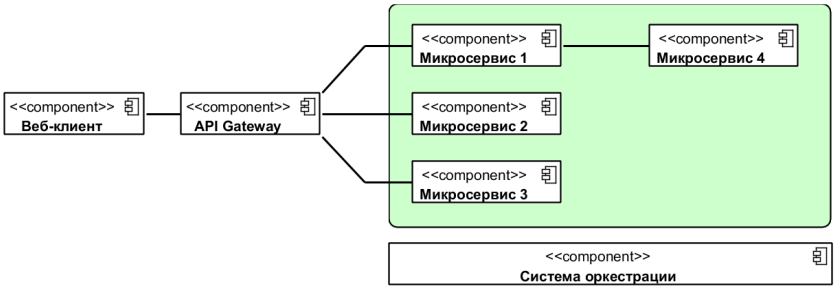
На входе стоит балансировщик нагрузки, перекидывающий запросы на сервера Network Virtual Appliances внутри Demilitarized Zone, обеспечивающие безопасность (инспекцию пакетов, файерволл и т.п.). Это всё готовые компоненты, которые надо просто подключить к инфраструктуре. Всё остальное происходит внутри виртуальной сети, доступа к машинам которой снаружи нет (есть ещё админский доступ через Bastion — по сути, ssh-сервер, с которого после прохождения всей необходимой авторизации можно ходить на другие машины в виртуальной сети). За DMZ находится пул виртуальных машин вебчасти, за которым — пул виртуалов бизнес-части, за которым — кластер с СУБД (который может заниматься просто репликацией, тогда балансировщик нагрузки единственное, что делает — переключает на горячую копию, если основная СУБД померла; а может заниматься продвинутым шардированием, о чём чуть позже).

Каждый из слоёв системы состоит из виртуальных машин за балансировщиком нагрузки — во-первых, для повышения производительности, во-вторых, для надёжности (если одна машина умрёт, балансировщик просто не будет направлять ей запросы, так что её смерть никто извне не заметит), в-третьих, для масштабируемости. Изначально в пуле может быть только одна виртуалка, но если она перестаёт справляться, может динамически запускаться вторая и брать на себя работу с помощью балансировщика. Запускать новую машину в пуле можно руками, а можно и автоматически, при достижении критического значения какой-то из метрик (например, загруженности процессора выше 80%), для чего облачные провайдеры имеют развитые средства мониторинга состояния виртуальных машин.

С инфраструктурой, тем более её автоматическим масштабированием, однако, надо быть осторожными, поскольку аренда виртуальных машин и прочих штук типа балансировщиков нагрузки, стоит денег, и иногда немалых. Сначала следует определиться с бизнес-целями — принесёт ли обеспечение дополнительной надёжности и производительности приложения достаточно денег, чтобы покрыть расходы на инфраструктуру.

1.6. Микросервисная архитектура

Микросервисная архитектура на данный момент самая популярная для распределённых приложений, хотя архитектурная мода как маятник раскачивается между микросервисными и монолитными приложениями. Микросервисный подход предлагает делить приложение на маленькие куски, каждый из которых представляет собой отдельный вебсервис и достаточно мал по размеру, чтобы его «мог написать один человек за две недели» (на самом деле, в реальной жизни микросервисы обычно побольше — подходящего размера, чтобы один микросервис могла разработать и сопровождать одна небольшая команда, но зависит от конкретного проекта, конечно). Микросервис имеет свою модель предметной области и является ограниченным контекстом с точки зрения предметно-ориентированного проектирования. Итоговое приложение собирается из микросервисов, вызывающих друг друга:



Как обычно, есть *веб-клиент*, который шлёт запросы на *API Gateway*. Также вполне может быть и выделенная веб-серверная часть, а может и не быть — отдаваемая клиенту страница может собираться из элементов, предоставляемых отдельно каждым микросервисом. Gateway не содержит никакой бизнес-логики, а просто перенаправляет запросы внутрь виртуальной сети, где работают микросервисы, и нужен прежде всего для того, чтобы у клиента был один фасад для всего приложения, и ему не надо было знать адреса отдельных частей. *Система оркестрации* (например, Docker Compose или Kubernetes) управляет временем жизни микросервисов, масштабированием, балансировкой нагрузки, мониторингом и т.п. — для микросервисных приложений это на самом деле очень важная вещь, поскольку сложность системы во многом определяется не реализацией самих микросервисов, а связями между ними.

Этот архитектурный стиль успешно применяется для предметных областей произвольной сложности (начиная с небольших приложений наподобие HwProj 2, где микросервисов меньше десятка, заканчивая гигантскими системами типа Facebook, с сотнями разных микросервисов).

Особенности микросервисного подхода таковы:

- Поскольку каждый микросервис — это отдельное приложение, он может быть написан на своём языке и стеке технологий, независимо от других. Хорошо это или плохо, открытый вопрос, потому что с одной стороны, это позволяет каждому писать на языке, на котором он наиболее продуктивен, с другой стороны, зоопарк из технологий усложняет поддержание инфраструктуры разработки. Но поскольку любой сервис можно переписать за две недели, это не то чтобы большая проблема.

- Каждый микросервис хранит свои данные сам, не имеет право разделять схему данных с другими. В этом смысле микросервис схож с объектом из классического ООП, то есть имеет своё внутреннее состояние и внешнее поведение. Обмениваться данными с другими микросервисами, конечно, можно, но только через публичный интерфейс. Это позволяет микросервисам выбирать наиболее эффективные способы хранения данных — не требуется поднимать один на всё приложение настоящий сервер с СУБД, как в N-звенной архитектуре, можно обойтись легковесными решениями типа SQLite или выбрать подходящую NoSQL-базу (но помните про масштабирование, все реплики одного микросервиса должны быть согласованы — проще всего это обеспечить, если сам микросервис состояния не имеет вовсе, но можно иметь СУБД, на которую смотрят все реплики).

- Каждый микросервис может быть масштабирован независимо, причём это делают даже не специальные балансировщики нагрузки, как в случае с N-звенной архитектурой, а сами оркестраторы, которые всё равно нужны и всё равно всем управляют. Так что если у вас есть функция, которая никому не нужна, запустите одну реплику с ней и пусть будет, и если есть функция, которая не справляется со своей работой, запустите хоть десять реплик только для неё. Микросервисы обычно работают в контейнерах, а не на виртуальных машинах, так что это ещё и дешевле, если делается на арендованных облачных мощностях.

- Каждый микросервис может быть задеплоен независимо от остальных, если его внешний интерфейс не менялся, так что циклы разработки и релизы отдельных микросервисов не должны быть согласованы. В отличие от монолита, где один маленький баг может задерживать релиз всей системы. ДАже если внешние интерфейсы менялись, согласовывать развёртывание надо только с соседними командами, а не со всем проектом. Поэтому микросервисные системы могут позволить себе по несколько релизов *в день* и даже подход «любой коммит сразу в production».

- Каждый микросервис падает отдельно, так что если вся система написана с расчётом на то, что любой компонент может отказать, любой сбой может быть легко локализован и не приведёт (по идее) к отказу всей системы.

- Микросервисы на то и микро, чтобы кодовая база каждого была очень маленькой и простой, что позитивно сказывается на стоимости сопровождения.

Однако, конечно, за всё надо платить. У микросервисного архитектурного стиля достаточно и недостатков. Прежде всего то, что сложность приложения перекладывается с кода на оркестрацию. А это не очень хорошо, потому что языки и приёмы программирования эволюционируют уже почти столетие, а технологии для конфигурирования взаимодействия микросервисов, в целом, только в начале пути. Так что зависимости между микросервисами могут быть неочевидны, их трудно отлаживать и мониторить, требуется набор специальных знаний и навыков, развитая культура DevOps во всей организации. Кроме того, само по себе микросервисное приложение сложнее эквивалентного по функциональности приложения в других стилях, в силу большего оверхеда на всякие инфраструктурные задачи. Да и разработка/тестирование сложнее, поскольку требуется либо куча mock-объектов, либо умение быстро поднимать и запускать все зависимости.

Кроме того, микросервисные приложения гораздо более требовательны к сетевой инфраструктуре, поскольку куча вызовов, которые в других стилях делаются локально, в микросервисном стиле сетевые (поэтому, кстати, не получится взять монолит и сделать каждый класс микросервисом — сеть ляжет). Обычно все микросервисы находятся внутри одной сети облачного провайдера, и там могут применяться очень скоростные соединения (например, InfiniBand, где скорость передачи измеряется в сотнях гигабит в секунду), но всё же.

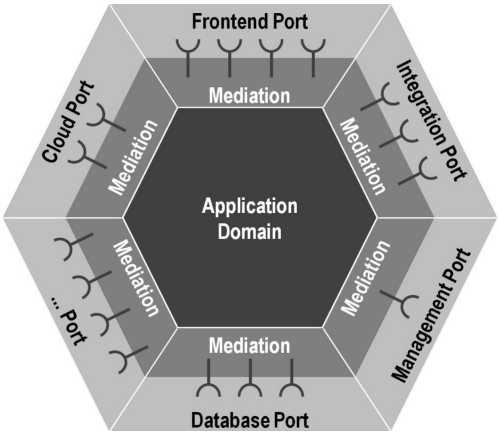
Наличие отдельных хранилищ данных для каждого микросервиса хорошо в плане сокрытия деталей реализации и обеспечения скорости работы, но плохо в плане поддержания целостности данных (тем более с учётом возможных отказов). Так что целостность данных гарантировать обычно даже не пытаются, строя архитектуру из семантики eventual consistency — все данные будут консистентны когда-то в будущем.

2. Архитектура конкретных сервисов

Вместе с архитектурными стилями «в большом», определяющими структуру распределённого приложения в целом, используются специфичные архитектурные стили «в малом», определяющие архитектуру каждого конкретного сервиса. В современном мире все они строятся на предметно-ориентированной архитектуре и служат прежде всего для изоляции модели предметной области от внешнего мира и «средств доставки» — веб-API, очередей сообщений и т.п. Вот в том, как они это делают, есть нюансы.

2.1. Гексагональная архитектура

Гексагональная архитектура, или «порты и адаптеры» — это дальнейшее развитие уровневой архитектуры. В классической уровневой архитектуре уровень предметной области, где находится содержательная бизнес-логика системы, находится где-то посередине — над ним слои взаимодействия с пользователем, под ним — инфраструктура, библиотеки, персистентность[3] и т.п. В гексагональной архитектуре предлагается инфраструктуру тоже рассматривать как часть внешнего мира, и тогда система получается устроенной очень просто. Самый нижний уровень — уровень предметной области. Над ним — слои, обеспечивающие взаимодействие с внешним миром, которые включают в себя предоставляемые и потребляемые интерфейсы, и адаптеры, преобразующие данные из внешнего мира в чистую предметно-ориентированную модель:



Основная идея подхода состоит в том, что классы, моделирующие предметную область, были максимально просты и максимально переиспользуемы. То есть если мы хотим сделать веб-приложение, десктопный и мобильны варианты — не вопрос, классы предметной области вообще не надо менять. Если хотим сменить СУБД с реляционной на объектно-ориентированную — не вопрос, классы предметной области менять не надо.

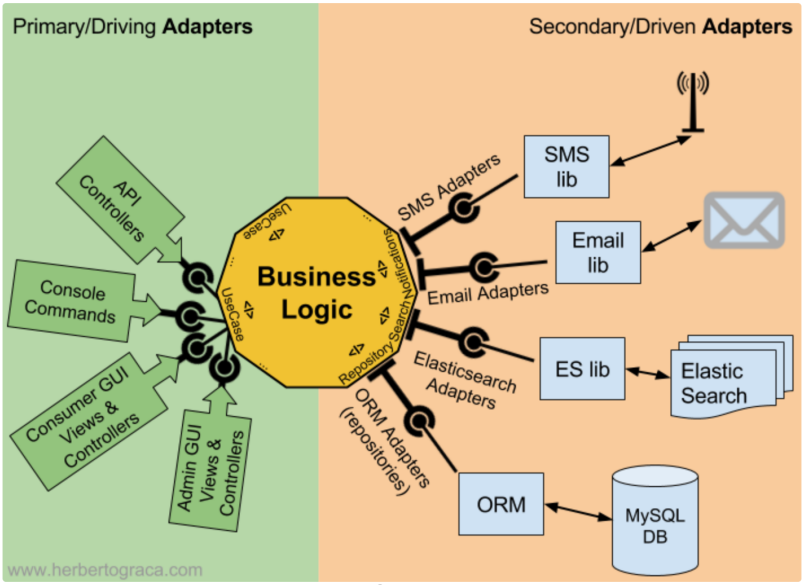
Чтобы этого добиться, слой предметной области описывает интерфейсы, которые он предоставляет, и которые ему нужны для работы. Слой адаптеров реализует эти интерфейсы, что само по себе может быть сложно, но предметной области не касается — например,

[3] Т.е. по сути сохранение в БД.

Data Access Layer может полностью инкапсулировать в себе работу с БД. Это на самом деле пример применения принципа Dependency Inversion — и для того, чтобы собрать всё воедино, используется подход Dependency Injection. То есть классам предметной области в конструкторы передаются конкретные реализации потребляемых ими интерфейсов, а внешнему миру передаются конкретные классы предметной области или адаптеры, реализующие интерфейсы, которые компонент обязуется реализовывать. Почему эта архитектура и известна как «порты и адаптеры» — всё взаимодействие с классами предметной области осуществляется через порты, вся валидация и преобразование данных — через адаптеры, и никак больше.

Вот почему архитектура называется гексагональной, вопрос менее понятный. На самом деле, один компонент может предоставлять много разных наборов интерфейсов, не обязательно шесть. Но так уж повелось, что компонент в работах по этому делу рисовали в виде шестиугольника. К тому же, шестиугольниками можно замостить плоскость, что иллюстрирует взгляд на систему в целом как на кучу таких вот компонентов-шестиугольников, интегрированных (через порты, естественно) друг с другом. Поэтому название прижилось.

Вот немного более страшная картинка, которая показывает пример типичного бэкенда веб-приложения:



© https://herbertograca.com/2017/09/14/ports-adapters-architecture/

Тут показана ещё довольно важная штука — порты у компонента делятся на предоставляемые (они же «primary» или «driving») и потребляемые (они же «secondary» или «driven»). Инициация действия возможна только через primary-порты, тогда как secondary-порты используются для реагирования на запросы и ими управляет ядро системы.

На рисунке видно, что в качестве primary-портов могут выступать интерфейсы, используемые «средствами доставки» запросов — сетевыми запросами разных видов, консольными командами. Однако сами порты ничего не знают про средства доставки, и им дела до них нет. Различные порты соответствуют различным случаям использования приложения

— например, как на рисунке, пользовательской функциональности, функциональности админа, функциональности API.

В качестве secondary-портов выступает вся служебная функциональность, нужная системе. Причём, как обычно, система ничего не знает про то, как эта функциональность реализуется — в ней есть лишь функциональность нотификаций, поиска и хранения, например. Конкретные библиотеки и технологии спрятаны за адаптерами, которые, в свою очередь, реализуют интерфейсы из ядра. Например, интерфейс нотификации реализуется адаптером электронной почты, который дёргает библиотеку, отправляющую письма. Ядро даже не знает, куда именно отправляет нотификации, потому что другая реализация этого же интерфейса может слать SMS-ки или сообщения в Slack.

У гексагональной архитектуры есть очевидные плюсы.

- Изоляция содержательного кода системы от «механизмов доставки» — от того, ка именно происходит взаимодействие с пользователем. Что позволяет легко заменить фронтенд или даже переиспользовать одну модель предметной области из разных приложений. Что, впрочем, типично не только для гексагональной архитектуры, но и для любой уровневой архитектуры вообще.

- А вот что классическая уровневая архитектура не умеет — это изоляция вспомогательных механизмов. Это делает модель предметной области независимой не только от конкретных библиотек и инструментов, но даже от прослоек типа Data Access Layer. Что хорошо не только в плане «поменять технологию на совсем другую», но, главное, позволяет держать модель предметной области максимально простой.

- Есть и чисто прагматическое преимущество такого разделения — лёгкость тестирования. Если ядро системы напрямую ни от кого не зависит, во все его внешние зависимости можно передать объекты-заглушки[4] и писать модульные и интеграционные тесты, не разворачивая реальную СУБД, не устанавливая специфическое оборудование и т.д. и т.п.

- Раз всё взаимодействие с внешним миром происходит через адаптеры, адаптеры могут заниматься верификацией, валидацией и конвертированием данных во внутреннее представление системы. Так что модель предметной области может спокойно хранить данные в том формате, в котором ей удобно, и считать, что данные, которые туда таки попали, заведомо корректны.

Есть, конечно, и минусы:

- Иногда гексагональная архитектура — это стрельба из пушки по воробьям. Если приложение планируется всего одно, предметная область проста, а менять технологии не планируется (хотя это всегда не планируется...), то можно потратить кучу времени на написание красивой модели, адаптеров, на внедрение зависимостей и т.д. и т.п. Часто оно того не стоит.

- Некоторая тонкость заключается в том, что считать платформой, на которой пишется ядро системы, а что внешней зависимостью, которую надо внедрять. Считать зависимостью стандартную библиотеку, наверное, довольно тупо. Но вот Guava в Java

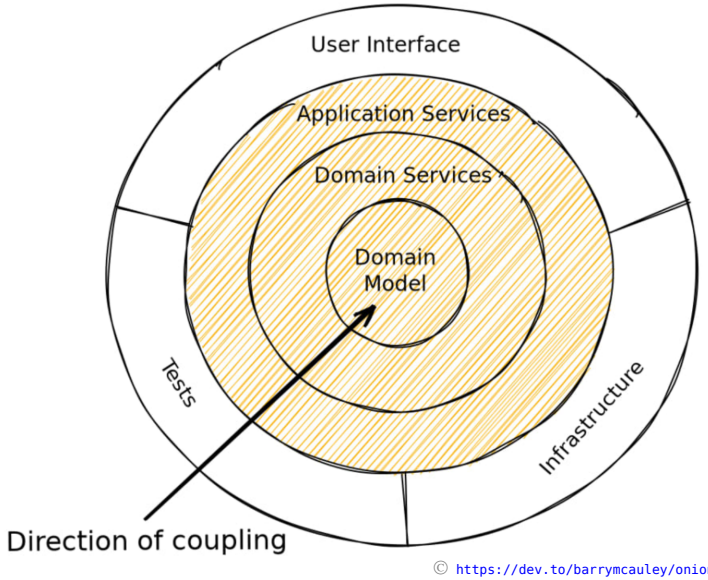
[4] Mock-объекты, если более точно.

или boost в C++ — это зависимость или часть платформы? Или вот библиотеки типа Qt — на них, в общем-то, пишется приложение от начала до конца, и там куча инфраструктурных вещей реализована — их надо внедрять? Если да, то как? Но если нет, то где грань между внешним миром и платформой? И много ли сейчас приложений пишется на голом языке программирования, без какого-то фреймворка и технологического стека вокруг него, со своими библиотеками, инструментами и даже иногда языками?

- Гексагональная архитектура не то чтобы очень специфична. Она говорит, что есть ядро, есть адаптеры, есть внешний мир, и работает это всё вместе так-то. Это хорошо, но при её применении остаётся слишком много свободы, что скорее плохо, чем хорошо.

2.1.1. Луковая архитектура

Луковая архитектура («Onion architecture») — архитектура с довольно странным названием, являющаяся, на самом деле, уточнением гексагональной архитектуры. Идея такая же — есть модель предметной области, включающая в себя всю бизнес-логику и модель данных приложения (не только приложения, а целой системы — ведь ядро может переиспользоваться в нескольких приложениях сразу). Есть внешний мир, который с ядром взаимодействует. Однако она предписывает наличие в ядре дополнительных уровней:



© https://dev.to/barrymcauley/onion-architecture-3fgl

Общее правило структурирования уровней тут понятное: внутренние слои ничего не знают о внешних, а лишь предоставляют интерфейсы и принимают в конструкторы зависимости по этим интерфейсам. В самом центре находится доменная модель (та самая модель предметной области) — классы, реализующие бизнес-логику и моделирующие сущности из реального мира. Над ней — доменные сервисы, то есть классы, использующие несколько сущностей в своей работе (функциональность, которую нельзя естественным образом

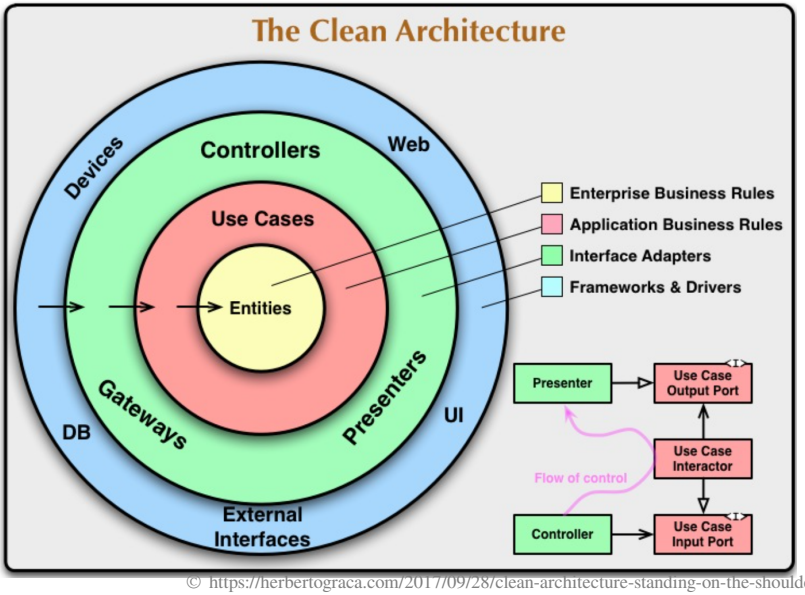
отнести к одному из классов предметной области). Часто сервисы — это статические классы. Над слоем доменных сервисов — сервисы приложений, они занимаются оркестрацией объектов и сервисов предметной области, реализуют случаи использования, всячески поддерживают код, который пользуется компонентом из внешнего мира. Над слоем сервисов приложений находится уже внешний по отношению к компоненту мир — пользовательский интерфейс и механизмы доставки, инфраструктура, хранение данных и т.д. Уровневость нестрогая, то есть уровень вправе обращаться к объектам любого из уровней ниже.

Кстати, структура уровней в «луковой архитектуре» примерно соответствует уровням, предлагаемым в Domain-Driven Design. Единственное, что DDD разделяет инфраструктурный уровень и уровень приложения, тогда как тут они смешаны в один, поскольку и то и другое — штуки для поддержки внешних зависимостей компонента. Зато в DDD доменные сервисы являются полноценными обитателями доменной модели, а тут они вынесены в отдельный слой.

Тут автор (очень рекомендую его цикл постов с подробным рассказом о том, откуда всё это пошло и как эволюционировало) обращает внимание, что в оригинале в луковой архитектуре репозитории были частью доменной модели, но на самом деле логично, чтобы они были вовне, на самом внешнем уровне (как в гексагональной архитектуре). Поэтому про луковую архитектуру теперь, в общем-то, обычно так и пишут.

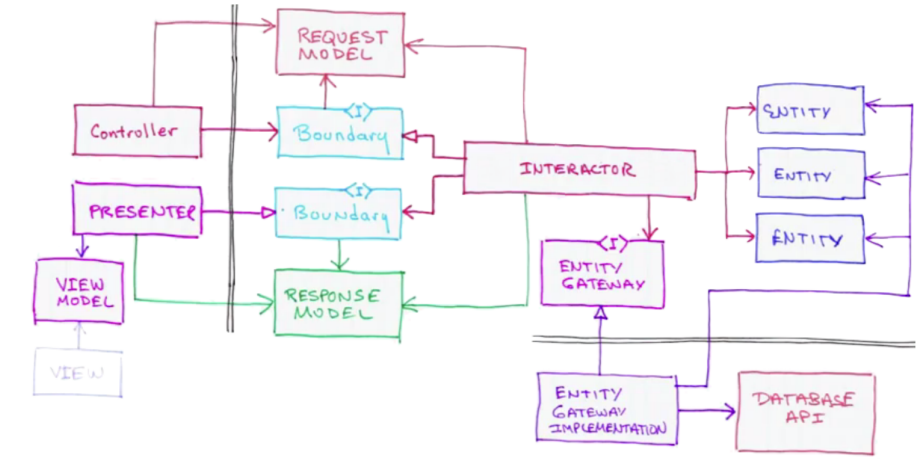
2.1.2. Чистая архитектура

«Чистая архитектура» («Clean architecture») — дальнейшее уточнение идей луковой и гексагональной архитектур. Тоже предполагается слоистый стиль, где самый нижний слой — доменная модель, тоже зависимости направлены строго от внешних слоёв к внутренним, тоже используется Dependency Injection, чтобы классы доменной модели могли работать с внешним миром. Но в дополнение ко всему этому она специфицирует поток управления и определяет конкретные способы взаимодействия с пользователем:



© https://herbertograca.com/2017/09/28/clean-architecture-standing-on-the-shoulders-of-giants/

На самом деле, самое интересное тут — как раз взаимодействие с внешним миром, детали статического устройства которого показаны на этом рисунке:



Двойные чёрные полоски — это границы компонента, отделяющие плохой внешний мир от хорошего внутреннего. Что происходит в рантайме:

1. Запрос попадает в контроллер (не важно откуда, приходит по сети или из GUI десктопного/мобильного приложения, всё равно есть какая-то точка входа, которая обслуживает команды от пользователя).

2. Контроллер парсит параметры запроса (архитектура больше всё-таки про веб-приложения), либо выполняет валидацию, и конвертирует запрос в Request Model, которая уже провалидирована и в виде, удобном для ядра системы.

3. Дальше контроллер дёргает интерактор, передавая ему Request Model в качестве параметра вызова одного из его методов. Контроллеру интерактор передаётся при инициализации системы по его интерфейсу Boundary (Dependency Injection тот самый).

4. Интерактор с помощью репозитория Entity Gateway (про который он опять-таки знает только его интерфейс, и реальный объект подставляется через DI) находит нужные для исполнения запроса сущности.

5. Репозиторий может подгрузить сущности из базы, создать при необходимости или просто вернуть уже готовые, это его дело, и это находится вне границ системы (хотя сами сущности — в самом её центре, такие дела).

6. Интерактор координирует работу сущностей по исполнению запроса, после чего формирует Response Model.

7. Response Model передаётся презентеру, про который интерактор опять-таки знает только его интерфейс.

8. Презентер, используя данные из Response Model, генерирует View Model и отправляет её клиенту (в случае, если речь идёт про настольное/мобильное приложение, там тоже бывают View Model-ы, на которые байндятся элементы на форме, либо презентер может сам управлять отображением информации на форме — зависит от используемой библиотеки и соответствующих ей паттернов).

9. У клиента View отображает обновившиеся данные.

В общем-то, ничего особо нового в этой архитектуре нет, но чёткое разделение ответственности сильно помогает и в разработке, и в тестировании. Поэтому такая архитектура стала весьма популярной. Например, есть паттерн VIPER — View, Interactor, Presenter, Entity, Routing — это буквально реализация идей «Чистой архитектуры» с некоторыми небольшими модификациями для программирования iOS-приложений. Там в основном только так и пишут.

3. REST

Большинство современных веб-сервисов использует архитектурный стиль REST для реализации публичного API, а микросервисы его особенно любят: REST легковесен и прост, и при аккуратном использовании может уменьшить нагрузку на сеть, слабое место микросервисов. Про REST вообще вам уже рассказывали, однако он достаточно важен, чтобы ещё раз остановиться на принципах дизайна REST-сервисов.

- API строится вокруг ресурсов, а не действий. Например, `http://api.example.com/customers/` — хорошо, а `http://api.example.com/get_customer/` — плохо. Действия — это методы HTTP, не элементы сервиса.

- Отношения между сущностями могут быть также выражены в URL как ресурсы. Например, `http://api.example.com/customers/5/orders` может быть коллекцией всех заказов покупателя номер 5. Соответственно, можно добавить заказ, изменить и удалить заказ конкретно для нужного покупателя. Однако увлекаться не надо, URL вида `http://api.example.com/customers/1/orders/99/products` уже считаются плохой практикой. Если клиенту требуется больше одного раза пройти по ссылке, пусть делает несколько запросов — сначала получит Id нужного заказа, потом сделает запрос к общей коллекции orders, предъявив Id, и уже оттуда получит информацию о продуктах.

- API сервиса — это модель предметной области, а не данных. Вообще, API сервиса, как в обычном ООП, не должно раскрывать деталей реализации, как бы ни было велико искушение напрямую отобразить запросы в таблицы в БД.

- Используйте стандартную семантику HTTP — например, заголовки. HTTP имеет стандартный заголовок Content-Type — его можно использовать, чтобы указать формат сериализации. И заголовок Accept — чтобы в запросе указать желаемый формат сериализации ответа, например:

```
GET http://api.example.com/orders/2 HTTP/1.1
Accept: application/json
```

Не забывайте также про стандартные коды ошибок — можно просто выбирать наиболее подходящие по смыслу и использовать их.

- Хороший сервис, если он предоставляет доступ к потенциально большим коллекциям, должен предоставлять механизмы фильтрации и «пагинации». Фильтрация позволяет указать, какие элементы нас интересуют, и какая информация про них нам нужна (например, в Google Drive API есть поле fields, которое мы в примере выше указали как «*», то есть «вернуть всё», но могли бы указать, например, только name), чтобы не возвращать кучу бесполезных данных и забивать сеть. Пагинация позволяет получать информацию небольшими порциями, например, по 30 элементов. Тогда в запросе можно будет передавать сколько элементов мы хотим, и начиная с какого. В Google Drive API это параметры pageSize и pageToken. Нелишне, если это может понадобиться, добавить и опцию сортировки — например, в Google Drive API есть параметр orderBy у запроса list.

- Если элементы сами могут быть большими по размеру, может быть полезна поддержка механизма Partial Content HTTP. Запрос HTTP HEAD в таком случае должен возвращать длину передаваемых данных, но не отдавать сами данные. Клиент, зная длину, может делать GET-запрос с заголовком Range, передавая диапазон байтов, которые хочет получить. Это позволит клиенту получать данные по частям, отменять скачивание и продолжать скачивание после отмены, что всегда позитивно.

- Совсем уж хороший веб-сервис должен предоставлять клиенту информацию о связанных с последним запросом возможных операциях. Это называется Hypertext as the Engine of Application State (HATEOAS) (то есть анонсирование возможных переходов в конечном автомате REST-сервиса через HTTP), и выглядит как-то так (например, в ответ на запрос о продукте в заказе):

```
{
    "orderID":3,
    "productID":2,
    "quantity":4,
    "orderValue":16.60,
    "links":[
        {
        "rel":"customer",
        "href":"http://api.example.com/customers/3",
        "action":"GET",
        "types":["text/xml","application/json"]
        },
        ...
}
```

По задумке это позволяет клиенту динамически получать информацию о сервисе, и в идеале если вы можете выполнить один запрос к сервису, вы можете обнаружить по ссылкам и все остальные (сродни WSDL-описанию в SOAP). Однако ни один известный автору сервис этого не поддерживает, а сам формат выдачи HATEOAS-ссылок не стандартизован.

- Хороший веб-сервис будет эволюционировать и менять свой API. Поэтому ему требуется механизм версионирования, который бы позволял старым клиентам продолжать работать. Есть сразу несколько подходов к версионированию, большинство из них сводится к тому, что все версии сервиса запускаются одновременно и есть какой-то механизм, позволяющий понять, к какой версии выполняется запрос. Например, в Google Drive API «v3» в URL означает, что это запрос к API третьей версии. Коллеги из Microsoft пишут, что это не только не единственный, но даже и не лучший вариант, но подробности, если интересно, можно посмотреть в первоисточнике[5].

4. Общие принципы проектирования распределённых приложений

Далее обсудим несколько важных принципов проектирования распределённых приложений. Список, опять-таки, не исчерпывающий.

4.1. Самовосстановление

Как вы, наверное, помните из предыдущей лекции, сеть принципиально ненадёжна, поэтому любое распределённое приложение должно проектироваться с учётом отказов. Отказы бывают временными, связанными с короткой потерей соединения или просто потерей пакетов. Бывают отказы, связанные с тем, что нужный нам сервис не работает совсем (например, пропало электричество в том месте, где он задеплоен).А бывают отказы, связанные с тем, что сервис просто не успевает обрабатывать запросы. Разные виды отказов требуют разных стратегий восстановления.

С временными отказами борются повторением запросов. При этом надо уметь отличать временный отказ от постоянного — например, если неверный пароль при аутентификации, повторением делу не поможешь. Если ошибка кажется временной, можно повторить запрос, потом подождать некоторое время, повторить его снова, подождать вдвое большее время, повторить снова и т.д., пока не исчерпается количество попыток (стратегия «экспоненциального отката»). Основная проблема с этой стратегией в том, что это надо делать при вообще каждом сетевом запросе, и, поскольку требуется некое знание предметной области в плане установления «временности» отказа, это обычно не делают библиотеки.

Если отказ вызван невозможностью сервиса исполнять запросы, например, из-за его перегруженности или временного выхода из строя, то огромное количество повторов (от каждого клиента) только усугубит ситуацию. Чтобы так не было, применяют паттерн «Circuit Breaker» — прокси-объект, который может находиться в трёх состояниях:

- Closed — все запросы направляются сервису как обычно. Если кто-то из них завершается с ошибкой, увеличивается счётчик ошибок, а если счётчик ошибок превысил допустимое значение, Circuit Breaker переходит в состояние Open

[5] RESTful web API design, https://github.com/microsoftdocs/architecture-center/blob/main/docs/best-practices/api-design.md (дата обращения: 12.12.2021).

- Open — запросы не отправляются сервису, Circuit Breaker сам выбрасывает исключение. При переходе в состояние Open запускается таймер, по истечении которого Circuit Breaker переходит в состояние Half-Open

- Half-Open — некоторое небольшое количество запросов отправляется на проксируемый сервис. Если они заканчиваются успешно, Circuit Breaker переходит в состояние Closed (и сбрасывает счётчик ошибок), иначе в Open (и перезапускает таймер).

Circuit Breaker вполне можно применять вместе с повторами, но, возможно, имеет смысл сделать механизм, позволяющий коду, выполняющему повторы, понять, что Circuit Breaker «разомкнул цепь» и повторы делать уже нет смысла.

Полезно для детекции ошибок также иметь методы API для самодиагностики сервиса. Начиная от просто чего-то в духе ping, когда если сервис ответил кодом 200 или 204 на HTTP-запрос, он жив. Заканчивая методом, возвращающим текущий статус сервиса с точки зрения бизнес-логики (типа длины очереди запросов, среднего времени обработки, количества записей в базе и т.п.). Такие методы могут использоваться как самим вашим приложением, так и оркестратором для управления масштабированием и перезапуска сервиса (Kubernetes, например, так умеет).

Также помогает ограничить последствия сбоев практика разделения ресурсов приложения на изолированные группы. Например, если два сервиса запускаются на одной машине, и один из сервисов из-за ошибки сжирает все ресурсы процессора, второй сервис тоже перестаёт работать. Желательно делать так, чтобы даже если один сервис захватил все доступные ему ресурсы, система в целом продолжала работать. Тут, опять-таки, могут помочь оркестраторы, позволяющие лимитировать ресурсы под каждый сервис.

В случае, если запросы поступают на обработку неравномерно (например, университетская столовка имеет некоторые проблемы с пропускной способностью в начале большого перерыва, тогда как в остальное время почти не загружена), необходима буферизация запросов. Используйте очереди сообщений для буферизации, чтобы смягчить пики нагрузки — время обработки увеличится, но, по крайней мере, все запросы рано или поздно будут обработаны. Лучше ограничивать длину очередей, чтобы если нагрузка радикально превышает возможности по её обработке, некоторые клиенты получали бы отказы в обслуживании, а не вся система падала.

Если для какого-то сервиса важна отказоустойчивость, имеет смысл запустить сразу два (как минимум) его экземпляра. И иметь механизм автоматического переключения на резервный экземпляр при недоступности основного. При этом переключение обратно обычно делается вручную, чтобы иметь возможность убедиться, что работоспособность основного экземпляра полностью восстановлена и он не начнёт портить пользовательские данные. В идеале резервный экземпляр должен находиться в другом регионе (это полезно в случае ядерной войны США и Китая — сервера в Австралии спокойно продолжат обслуживать ваших клиентов, даже если основной экземпляр приложения будет вместе с датацентром в Нью-Йорке превращён в расплавленное стекло; и в более прозаических сценариях, например, в случае, если строители экскаватором разорвут магистральное оптоволокно — в СПбГУ такое случалось). Правда, резервный экземпляр большую часть времени не работает, а деньги за него платить надо, особенно за деплой в разных регионах облачные провайдеры любят взимать особую плату.

Ещё один приём повышения надёжности — промежуточное сохранение длительных

операций (checkpoints). Если что-то пойдёт не так, при перезапуске сервис сможет продолжить работу с промежуточного состояния, а не начинать всё заново.

Если, однако, несмотря на все усилия система всё-таки отказала, она должна делать это плавно и постепенно (то, что в англоязычной литературе называется «Graceful degradation»). Например, микросервис изображений товаров в интернет-магазине может отказать, тогда магазин должен показывать изображения-заглушки и продолжать работать. Или вот без микросервиса рекомендаций можно обойтись — да, несколько клиентов получат несколько худший сервис, чем обычно, но смогут пользоваться системой, пока сервис перезапускается. Остальные ничего даже не заметят.

Что интересно, современная архитектурная мысль сошлась к тому, что надо самостоятельно вносить отказы, чтобы убедиться в работоспособности системы. Есть практики раз в месяц грохать продакшн и поднимать систему из бэкапов, причём каждый раз это должен делать случайно выбранный член команды — чтобы убедиться, что система корректно переходит на резервные сервисы, и что все в команде знают, как поднять основной экземпляр. Так что когда случится настоящая проблема, любой джун, случайно оказавшийся на рабочем месте, знает, где найти инструкцию по починке прода, и знает, как ей пользоваться.

Более того, есть *инструменты*, автоматически случайным образом убивающие сервисы или сетевые соединения, и работающие на продакшене непрерывно — например, Chaos Monkey[6]. Кто-то говорил, что хорошая система должна стабильно работать даже если в дата-центр, где она развёрнута, запускают стаю обезьян, которые хаотично выдёргивают провода и делают нехорошие вещи с серверами, отсюда и название. Непрерывное тестирование отказоустойчивости позволяет системе совершенно спокойно относиться к настоящим отказам.

4.2. Избыточность

Лучший способ обеспечения отказоустойчивости — это избыточность, потому что если иметь на каждый сервис несколько его копий, при отказе одной остальные продолжат функционировать. Однако избыточность стоит денег, поэтому есть хорошие практики, позволяющие добиваться избыточности эффективно.

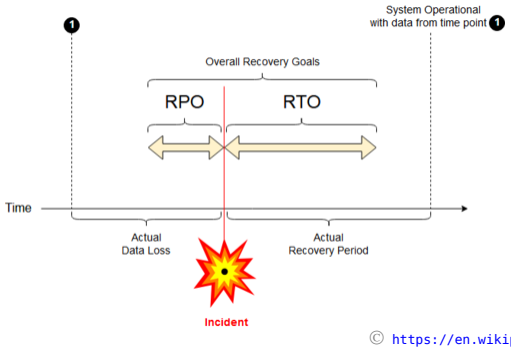
Во-первых, надо определиться с тем, что вы хотите от надёжности сервиса. Это чисто бизнес-решение, поскольку оно подразумевает балансировку стоимости эксплуатации и ожидаемую выгоду от надёжности системы. Так что стоит перед началом разработки определить ключевые показатели надёжности:

- Recovery Time Objective — сколько времени должно максимум пройти с момента обнаружения отказа до восстановления системы;
- Recovery Point Objective — данные за какой период до обнаружения отказа можно потерять (например, как часто надо жать на Ctrl-S при программировании, чтобы в случае падения IDE или всей системы страдать не сильно — если раз в день, то потеряете день труда, если раз в две секунды, то две секунды, но клавиатуру испортите);

[6] Инструмент от Netflix, домашняя страница: https://netflix.github.io/chaosmonkey/ (дата обращения: 12.12.2021).

- Maximum Tolerable Outage — сколько максимум времени бизнес-процесс организации может быть недоступен без значимого ущерба для организации (в отличие от RTO, MTO считает успешным восстановлением ручное исполнение бизнес-процесса).

Вот рисунок, иллюстрирующий происходящее:



© https://en.wikipedia.org/wiki/Disaster_recovery

После того, как с целевыми показателями определились, можно заложить в архитектуру механизмы их достижения.

- Балансировщики нагрузки — пожалуй, самый универсальный и надёжный инструмент. Если сервис не имеет своего состояния, запустите два (три, десять) и поставьте перед ними балансировщик нагрузки, который в случае отказа одного экземпляра просто перестаёт посылать на него запросы (впрочем, он может реализовывать более сложную логику типа Circuit Breaker). Балансировка нагрузки ещё и позитивно скажется на скорости работы системы в целом (но негативно на цене).

- Репликация БД — время от времени сохранять копии базы данных (желательно в другую СУБД, готовую мгновенно включиться в работу, но можно и в снапшоты, из которых потом можно восстановить данные). Насколько часто это делать — собственно, определяется RPO. Это абсолютно обязательная практика, так что облачными провайдерами часто делается сама собой.

- Разделение по регионам — ни балансировка нагрузки, ни репликация не поможет, если всё это физически делается в одном здании, и на него упал самолёт. Все крупные и важные системы географически разнесены, что не только повышает надёжность в случае масштабных отказов, но и, опять-таки, позитивно влияет на скорость работы, при правильной настройке DNS (если пользователи автоматически направляются на ближайший к ним сервер). Очень дорого стоит.

- Шардирование — техника сродни репликации, но репликация копирует всю базу, а шардирование равномерно размазывает данные по нескольким разным базам. Это сродни RAID-массиву: репликация работает как RAID 1, то есть просто копирует данные для надёжности, шардирование — как RAID 0, то есть просто распределяет данные с целью прежде всего уменьшить время доступа к ним (операции над разными шардами всегда могут выполняться параллельно). Шардирование, тем не менее,

позитивно влияет и на надёжность, поскольку в случае потери одного шарда вы теряете не все данные вообще, а только некоторые. 10% пользователей, заказы которых вы потеряли, лучше, чем 100% (хотя, конечно, тоже не очень, поэтому шардирование и репликация используются совместно — пока один шард восстанавливается из резервной копии, остальные продолжают работать как ни в чём не бывало). Политика шардирования может быть разной — можно делить записи по регионам пользователей, по первой букве из фамилии (в одной базе только фамилии на А, в другой на Б и т.п.), можно по хеш-функции от чего-нибудь, чтобы более случайно перемешать данные по шардам.

4.3. Минимизация координации

Компоненты распределённых приложений должны совместно работать над решением общей задачи, и если они уж слишком совместно это делают, это может нанести серьёзный ущерб производительности и надёжности системы. Поэтому с необходимостью координации действий сервисов нужно бороться, и, конечно, для этого есть несколько известных приёмов.

Первый — использовать событийный архитектурный стиль для организации взаимодействия между сервисами. Вместо того, чтобы вызывать методы других сервисов, чтобы сообщить им, что у нас что-то произошло (например, зарегистрирован новый пользователь), можно задекларировать событие, на которое все желающие могут подписаться и асинхронно обработать. Тогда источник событий сможет даже не знать, интересно это событие кому-то или нет, или есть ли в системе другие сервисы в принципе. Такие события должны быть связаны с предметной областью, поэтому известны как доменные (domain events). Их обычно рекомендуют использовать для «дополнительных» сценариев (например, добавление пользователя в базу при регистрации выполняется прямым вызовом, а посылка нотификации админу — доменным событием), но бывает так, что вся система строится только на событиях. Есть целые библиотеки, которые поддерживают такую модель, например, MediatR[7] для .NET.

Второй — явно разделять команды и запросы (Command and Query Separation, CQS) и его эволюция, Command and Query Responsibility Segregation, CQRS. Идея в том, что команда должна менять состояние, но ничего не возвращать, а запрос наоборот, ничего не менять, а только возвращать текущее состояние — и это хорошая практика в ООП вообще. CQRS делает наблюдение, что вообще говоря для команд и запросов можно использовать разные схемы БД и даже разные способы их хранения (реляционная для запросов и ООБД для команд, например), и это может в разы повысить эффективность выполнения операций. Особенно с учётом того, что при выполнении запросов не может быть конфликтов между транзакциями, раз запросы ничего не меняют. Кроме того, CQRS предполагает наличие явных доменных команд, которые содержат в себе бизнес-логику, и с которыми взаимодействуют клиенты сервиса. Разумеется, поддержание в актуальном состоянии двух разных баз с разными схемами может быть не очень простой задачей, так что реализовывать этот паттерн надо аккуратно (точнее, использовать семантику Eventual Consistency, о которой чуть дальше).

Сильно помогает, когда данные немутабельны, то есть записали один раз — и всё. То-

[7] Страница MediatR на GitHub: https://github.com/jbogard/MediatR (дата обращения: 12.12.2021).

гда, опять-таки, конфликты при записи невозможны, есть масса возможностей для распараллеливания и эффективного хранения таких данных (вспомните Apache Kafka, она делает именно так). Это использует подход Event Sourcing — идея его в том, что а давайте вообще не будем хранить состояние, а будем хранить только набор событий, которые в него привели. Тогда любой, кому надо, может построить текущее состояние, зная начальное и этот самый набор событий. Для скорости Event Sourcing может использовать read-only-снапшоты с текущим состоянием, которые время от времени (опять Eventual Consistency) обновляются из потока событий. Такой подход очень эффективен, когда событий много, и обработчиков много, и поддерживать между ними согласованное состояние слишком накладно (то есть на самом деле довольно часто, так что Event Sourcing нынче весьма популярен).

Наверное, не надо даже говорить, что все операции, требующие участия нескольких сервисов, должны быть асинхронными — пока запрос выполняется, сервис должен иметь возможность делать какую-то другую работу, а не ждать ответа. Иначе может возникнуть каскад задержек, когда один сервис ждёт второй, второй — третий и четвёртый, и т.д. Ещё на практике очень полезны так называемые *идемпотентные* операции — то есть операции, состояниекоторые могут быть повторены без изменения состояния сервера. Например, все запросы идемпотентны — если они не меняют внутреннее состояние, от случайного повторения запроса ничего не случится. Запросы в духе «включить» или «выключить» тоже идемпотенты, потому что если мы один раз что-то включили, можем вызывать «включить» дальше сколько угодно раз, ничего не изменится. А вот операция «переключить» не идемпотентна. Идемпотентные операции хороши тем, что могут быть безопасно повторены при восстановлении после временной ошибки, и если обработчик упал, другой обработчик может просто взять очередь первого и начать последовательно выполнять операции как ни в чём ни бывало.

Шардирование данных, как, наверное, понятно, также способствует уменьшению координации, поскольку запросы не толкаются вокруг одной БД, а исполняются параллельно и не мешают друг другу.

Кстати, насчёт БД, если их несколько (что бывает почти всегда, если используется микросервисная архитектура, и очень часто в остальных случаях), поддерживать их согласованными обычно очень трудно. Есть понятие «распределённая транзакция», когда хочется добиться атомарности и изолированности операции, включающей в себя несколько независимых сервисов (или независимых БД), они кажутся очень привлекательными для программистов, но их почти никто не умеет. Современная архитектурная мысль говорит, что их следует вообще избегать. Проще, быстрее и, как ни странно, надёжнее просто оптимистично выполнять операции в каждом сервисе отдельно, а если что-то пойдёт не так (например, один из участников вернёт ошибку), выполнить *компенсационную транзакцию*, которая вернёт всё как было. В этом случае опять-таки используется семантика Eventual Consistency — состояние в любой момент времени может быть несогласовано и не все сервисы могут иметь актуальные данные, но в какой-то момент в будущем гарантированно данные будут консистентны.

Это кажется не очень хорошо в плане программирования, потому что мы должны учитывать потенциальную неактуальность данных (прямо как в многопоточном программировании, где lock-free-операции должны учитывать, что данные в кеше каждого ядра могут быть свои и не соответствовать данным в памяти). Однако это суровая правда жизни, как говорит нам CAP-теорема:

В любой распределённой системе можно обеспечить не более двух из трёх свойств:

- *Согласованность данных (Consistency) — во всех вычислительных узлах данные консистентны;*
- *Доступность (Availability) — любой запрос завершается корректно, но без гарантии, что ответы всех узлов одинаковы;*
- *Устойчивость к разделению (Partitioning Tolerance) — потеря связи между узлами не портит корректность.*

Пи этом третий пункт теоремы в распределённых системах должен быть выполнен всегда, в силу принципиальной ненадёжности сети (кстати, в отличие от традиционных СУБД, которые работают на одной машине и абсолютно не устойчивы к разделению, зато гарантируют первые два пункта). Так что мы можем выбрать либо доступность, либо согласованность. Согласованность можно обеспечить, если остановить работу всех узлов, пока они не договорятся об общем состоянии (что само по себе нетривиально при возможности отказов, см., например, задачу византийских генералов[8]), что весьма негативно скажется на скорости работы системы (а следовательно, её доступности — клиенты будут получать отказ в обслуживании). Доступность, однако, сама по себе тоже не очень интересна, потому что если половина серверов банка думает, что у пользователя сто рублей, а другая половина — тысяча, это печально.

На самом деле, эта ситуация известна в теории баз данных, и есть две известные семантики работы с данными: ACID и BASE.

ACID:

- Atomicity — транзакция не применится частично;
- Consistency — завершённая транзакция не нарушает целостности данных;
- Isolation — параллельные транзакции не мешают друг другу;
- Durability — если транзакция завершилась, её данные не потеряются.

BASE:

- Basically Available — отказ узла может привести к некорректному ответу, но только для клиентов, обслуживавшихся узлом;
- Soft-state — состояние может меняться само собой, согласованность между узлами не гарантируется;
- Eventually consistent — гарантируется целостность только в некоторый момент в будущем.

Семантика ACID характерна для централизованных систем, и, поскольку очень удобна в работе, её на заре интернетов пытались перенести и на распределённые приложения, однако CAP-теорема не дала. Поэтому современные распределённые приложения используют более слабую, но всё ещё полезную семантику BASE — сервисы поддерживают целостность только своих данных и асинхронно координируются друг с другом. Собственно, Eventual Consistency, которая постоянно упоминается в этой лекции — это часть семантики BASE.

[8] https://ru.wikipedia.org/wiki/Задача_византийских_генералов (дата обращения: 12.12.2021)

4.4. Проектирование для обслуживания

Ещё один важный набор приёмов связан с тем, что распределённые приложения довольно сложно контролировать, и не всегда понятно даже, работают они или нет и что такое «работают» вообще (например, три секунды грузится веб-страница — это ок или нет?). Поэтому распределённые приложения должны сразу проектироваться с учётом необходимости управлять их работой и обслуживать их во время выполнения (так называемый принцип Design for operations, основа DevOps).

Самый важный принцип в этом плане — «делать всё наблюдаемым». В частности, обеспечение логирования всего, что происходит в каждом сервисе, и централизованное отображение и анализ логов (то есть подсистема логирования внезапно сама является распределённым приложением). Как пример такой технологии см. «Elastic Stack», включающую в себя такие известные штуки, как Logstash (для сбора логов), Elasticsearch (для индексации и классификации логов) и Kibana (для визуализации состояния системы по логам). В самих реализациях сервисов при этом используются библиотеки, пишущие логи (в правильно место у себя, откуда их потом заберут, или сразу в сетевое хранилище) — log4j/slf4j и его порты и аналоги на других языках (например, log4net).

Помимо логирования есть ещё понятие «трассировка» — это отслеживание того, что происходит с запросом пользователя. Трассировка, как правило, требует отслеживания обработки запроса несколькими сервисами, поэтому требуется поддержка «корреляции» запросов, то есть механизма, позволяющего понять, к обработке чего относится тот или иной вызов. Лог — это как бы горизонтальный срез состояния одного сервиса, как он обрабатывает много запросов, трасса — это как бы вертикальный срез состояния запроса, как его обрабатывают много сервисов. Трассировка позволяет локализовать отказы, идентифицировать узкие места, сетевые задержки и т.п.

На основе логов, трасс и «counter-ов» (то есть метрик производительности каждого конкретного сервиса) должна быть выстроена система *мониторинга*, основывающаяся на численных *метриках*, измеряемых в реальном времени. Примерами метрик могут быть количество обработанных запросов в секунду, количество ошибок в секунду, uptime сервисов и т.д. и т.п. Метрики специфичны для каждого конкретного приложения и определяются в основном бизнес-задачами, хотя есть и более-менее стандартные, которые измеряются оркестраторами (количество запросов, время на ответ, объём принимаемых и передаваемых данных, количество кодов HTTP 5xx в ответах, например). Организация мониторинга и выбор правильных метрик — это в каком-то смысле искусство, но какой-то мониторинг в любом случае необходим.

Чтобы содержательный мониторинг был в принципе возможен, надо договориться о стандартном формате логов и стандартных метриках для приложения, и следить, чтобы стандартов придерживались все сервисы, из которых оно состоит. Для микросервисных приложений это само по себе может быть сложно, потому что сервисов много и они могут быть написаны на разных технологиях. Но поэтому в таких проектах грамотные архитекторы особенно полезны.

Ещё один, кажется, довольно очевидный совет — автоматизировать всё, что может быть автоматизировано, включая развёртывание и мониторинг. Если для запуска приложения требуется четыре часа копировать куда-то файлы, запускать какие-то скрипты и руками лезть в базы, то ошибки неизбежны. В идеале всё, что делается с приложением, должно делаться автоматически одной командой.

Этого можно добиться, если относиться к конфигурации приложения, как к коду, который можно хранить, версионировать, согласованно редактировать и автоматически применять в рамках обычного CI/CD-процесса. Этому принципу следуют популярные оркестраторы, такие как Docker Compose и Kubernetes (про которые чуть позже) — в конфигурации оркестратора (обычно в виде YAML или JSON-файлов) описывается, какие сервисы в скольки экземплярах на каких портах должны быть запущены, кто раньше, кто позже, у кого какая конфигурация (задаваемая, например, через переменные окружения) и что делать, если кто-то упал. Конфигурация выкладывается в систему контроля версий и применяется одной командой (docker compose up или kubectl apply), лезть в конфигурацию работающей системы руками (например, чтобы остановить или запустить сервис) считается чем-то плохим.