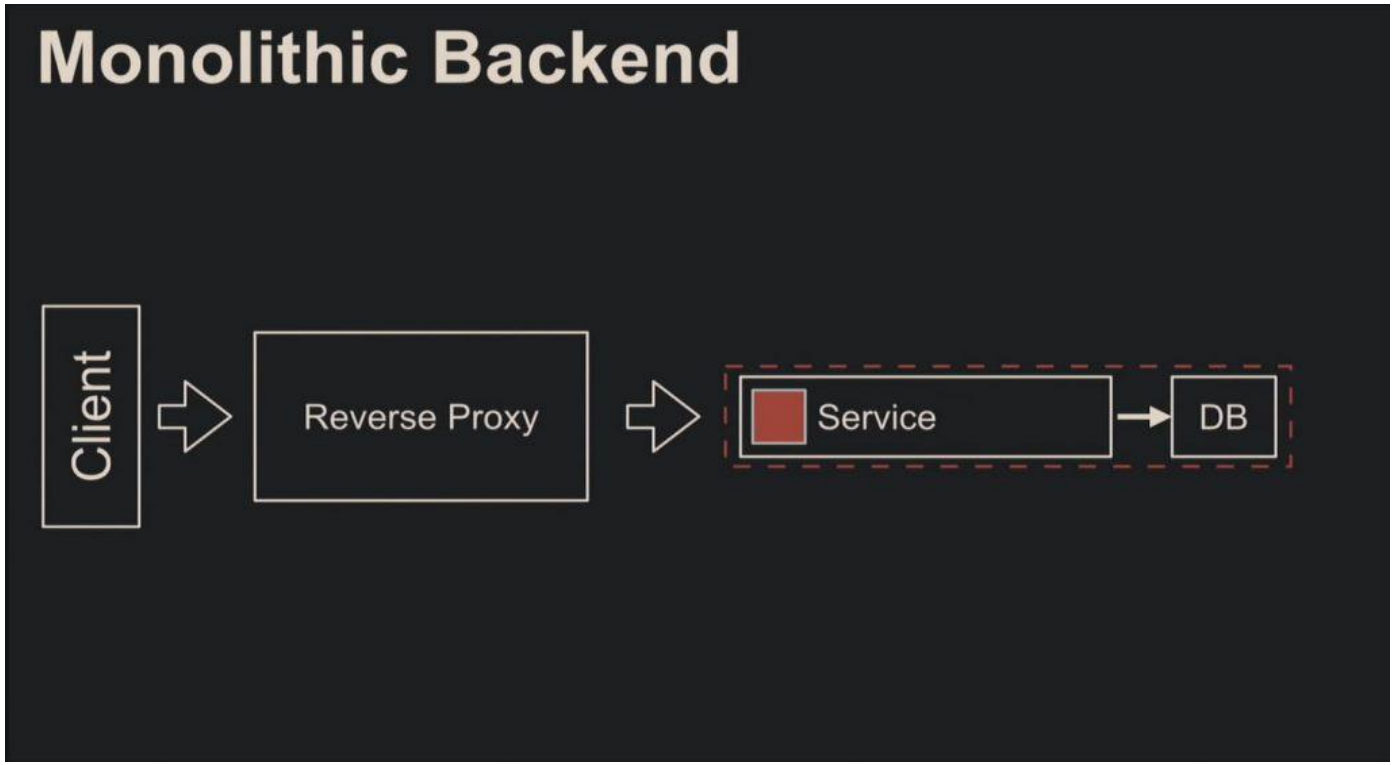
ЛЕКЦИЯ 4_1. КОНФИГУРАЦИИ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ, ШИНА ДАННЫХ, ПРОТОКОЛЫ СООБЩЕНИЙ МЕЖДУ СЕРВИСАМИ.

Виды архитектуры

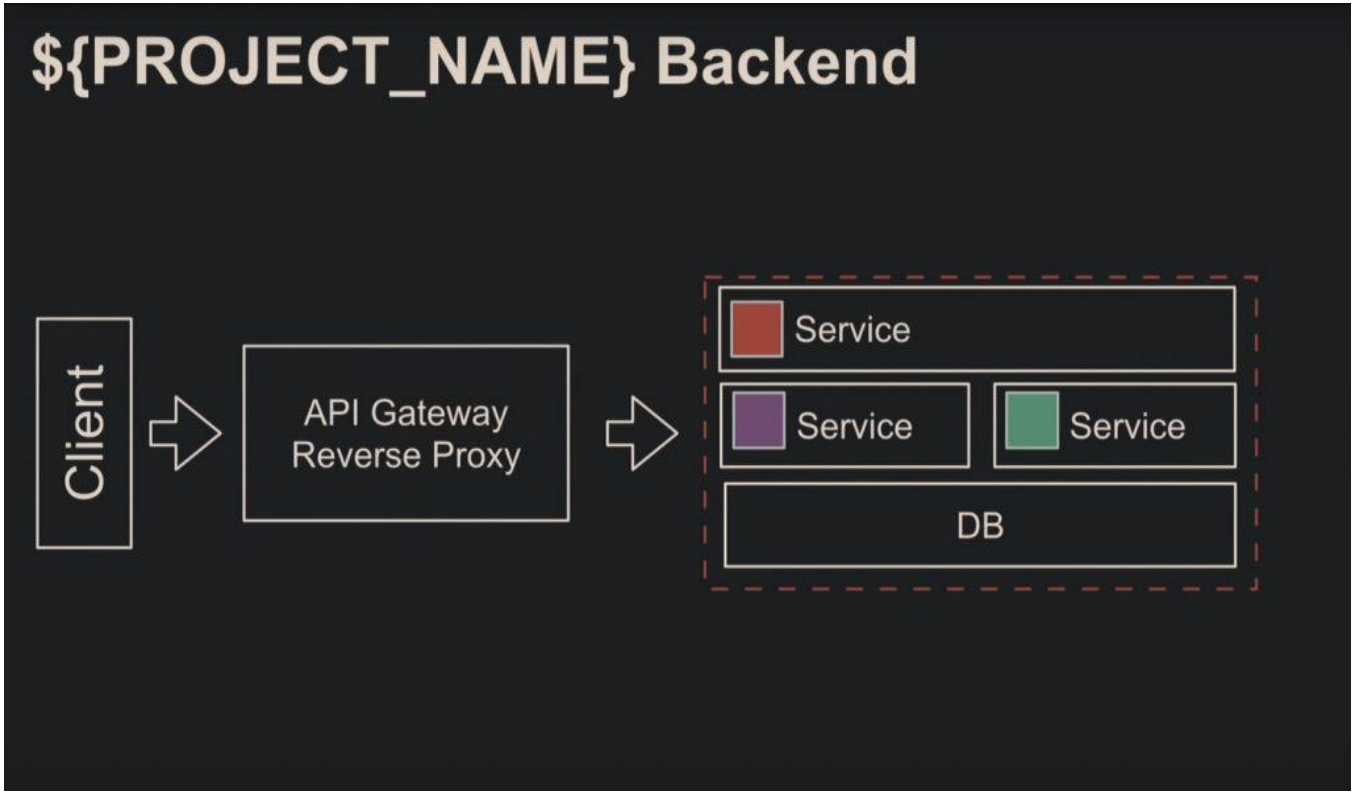
Простейший и популярный вариант архитектуры – **монолитная**. Каждый начинал с неё, и здесь нет никакой изоляции и распределённости: один монолит обрабатывает все запросы:



Проблемы:

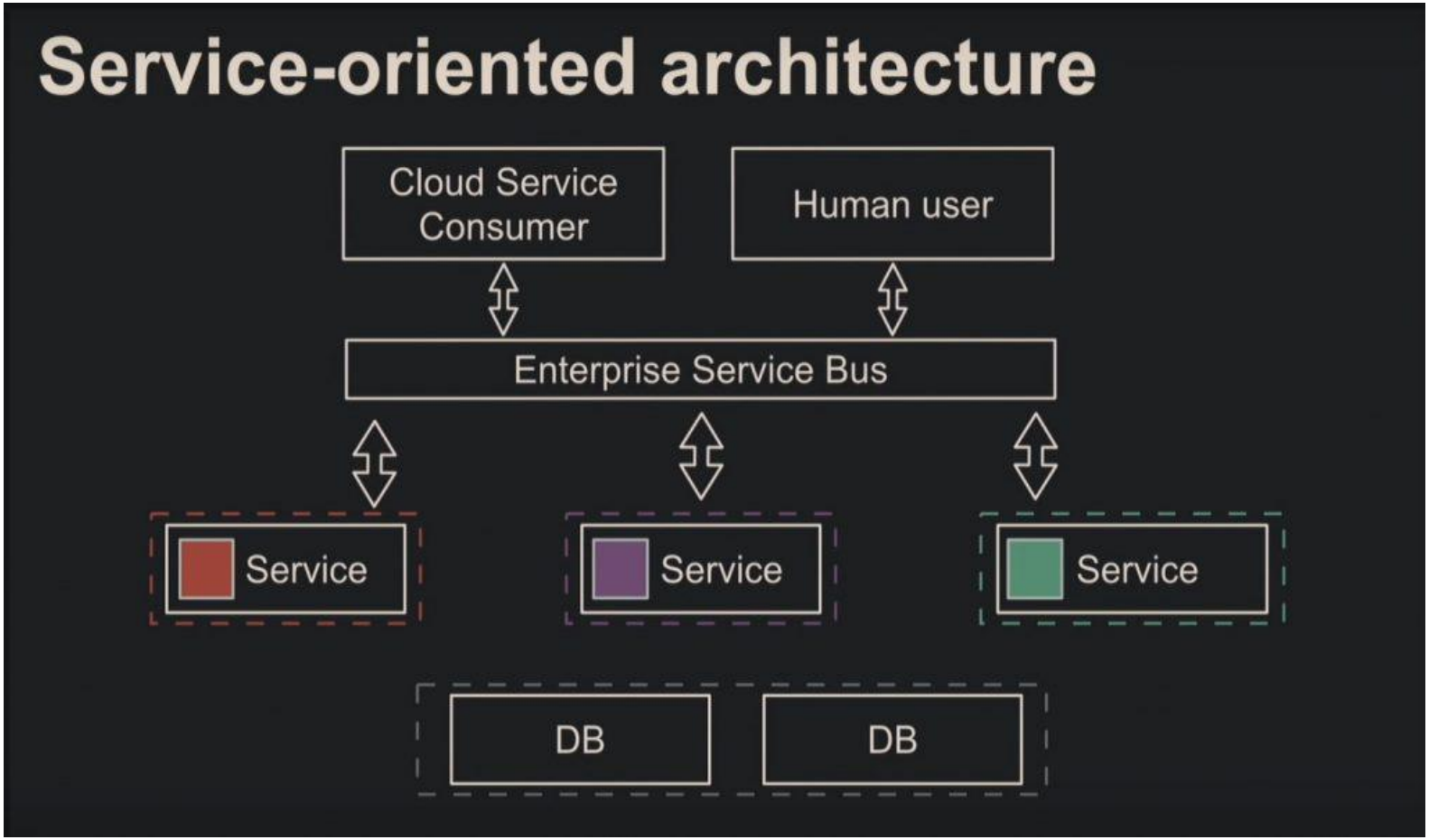
-отказоустойчивость;
-горизонтальное масштабирование;
-применение одной технологии или языка и невыгодность переписывать огромный монолит;
-сложность рефакторинга из-за хранения кода в одном месте;
-трудности работы в команде разработчиков;
-чтобы использовать повторно, придётся дробить.

Второй по популярности вид архитектуры – па-ра монолитов, микс из монолита и сервисов или даже микросервисов. То есть вы сохраняе-те монолит, а доработки выполняете с исполь-зованием современных технологий.



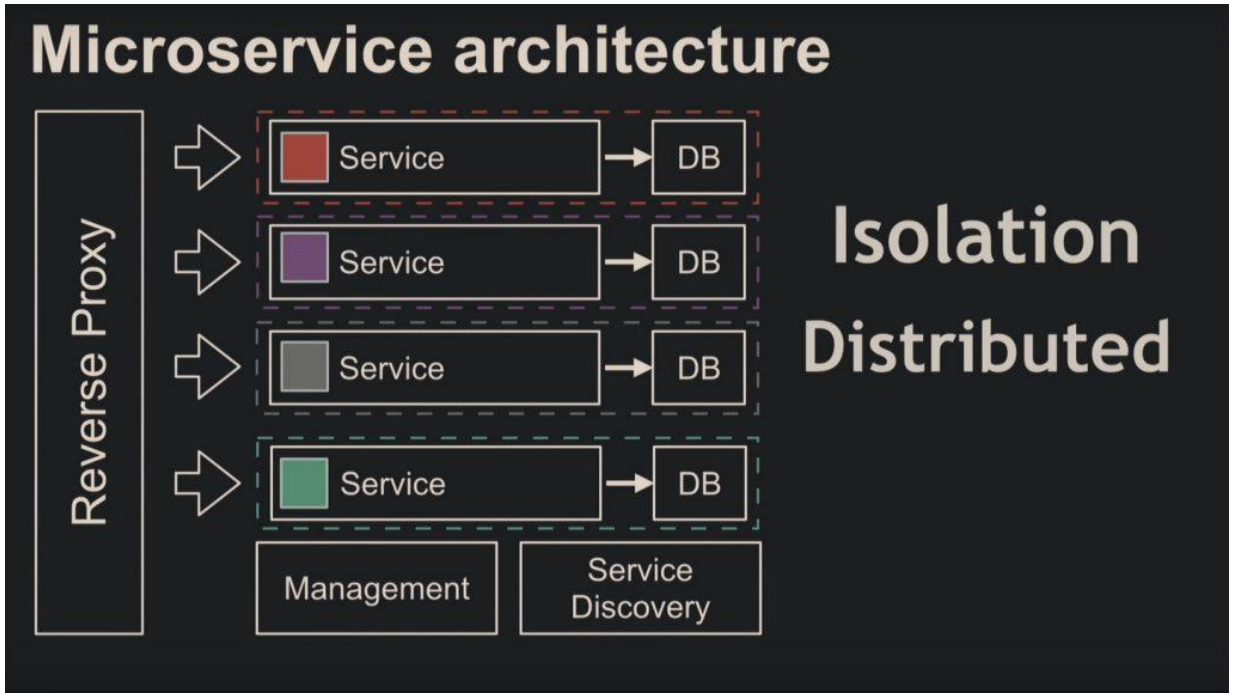
Это частично решает проблемы отказоустойчивости, масштабируемости и одного стека технологий.

Сервис-ориентированная архитектура предусматривает модульность разработки и слабую связанность компонентов, поэтому получаем изолированную и распределённую систему.
Главный минус – общая шина данных Enterprise Service Bus (ESB) с огромными спецификациями и сложностями работы с абстракциями и фасадами.



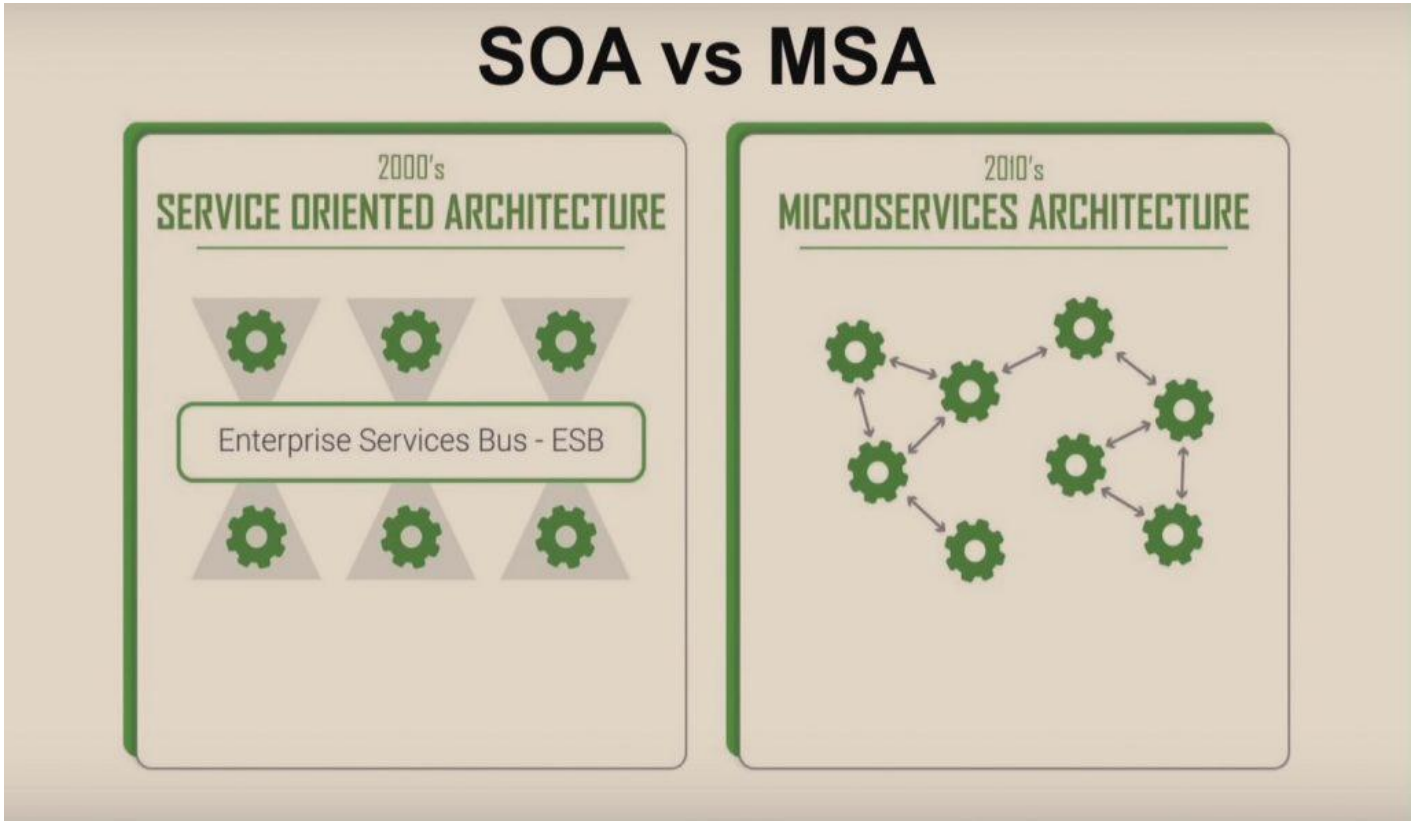
Сервисно-ориентированная архитектура

Микросервисная архитектура – не новая идея, а разновидность сервис-ориентированной архитектуры.

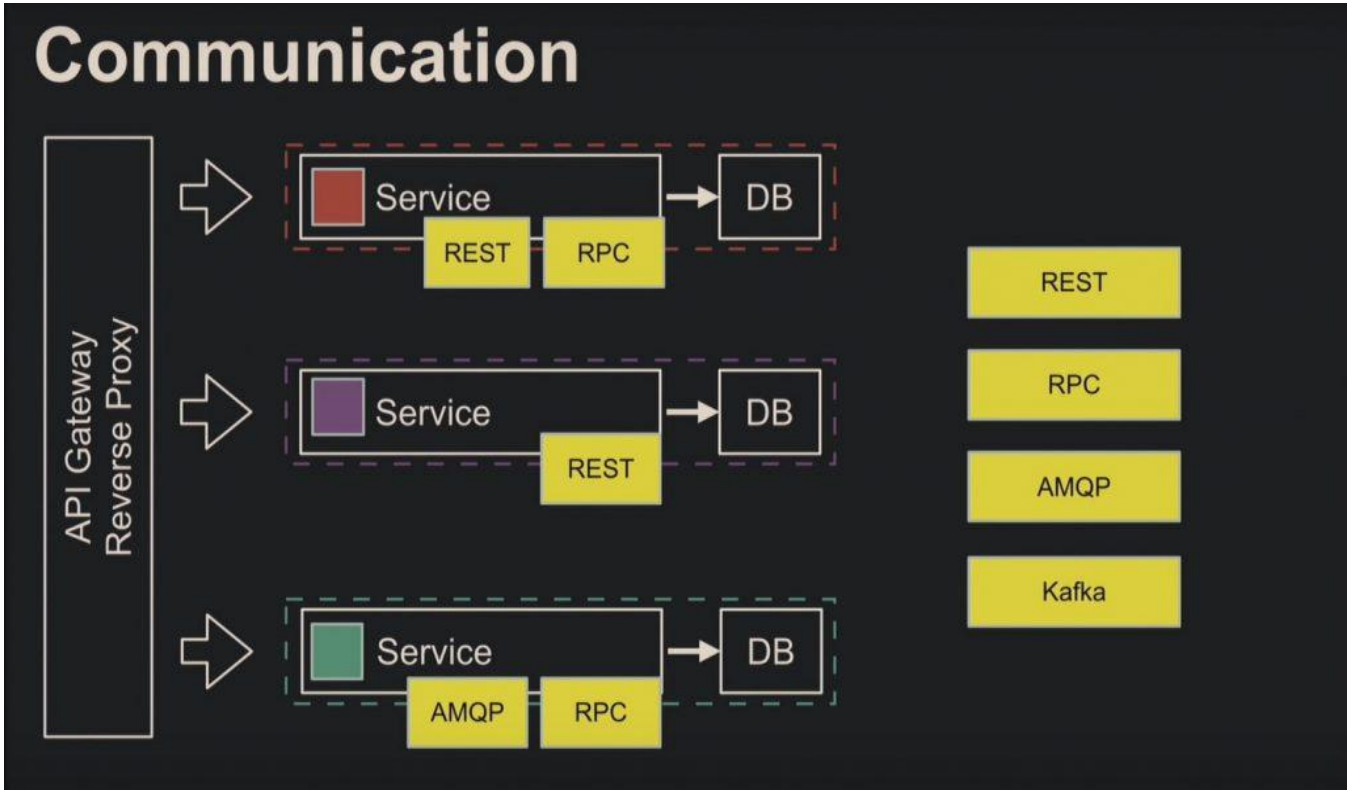


Микросервисная архитектура наследует от SOA изоляцию и распределённость. Здесь база данных не используется как шина данных. Компоненты изолируются и на уровне кода, и на уровне базы.

Следующее преимущество – протоколы обнаружения сервисов. Наглядная разница коммуникаций сервис-ориентированной и микросервисной архитектуры: у последней нет общей шины, и сервис обращается к любому другому напрямую:



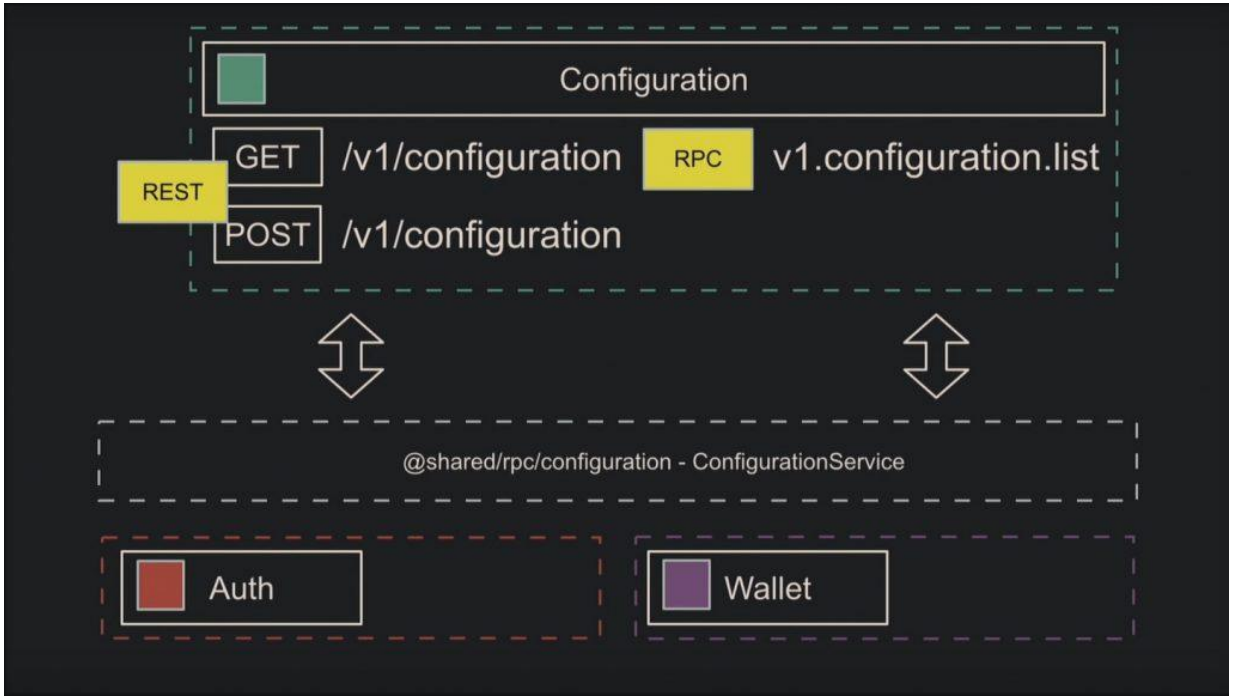
Выбор протоколов общения зависит от программиста. Например, вы используете REST для публичных запросов и RPC через AMQP для внутренних либо один общий протокол для всех.



Разделяют микросервисы с точки зрения либо бизнеса, либо программиста для переиспользования.

Но мешают этому две вещи:

-внутренние связи – при тесном взаимодействии микросервисы объединяют;
-транзакции – у разных микросервисов базы данных изолированы, а нужна одна общая.



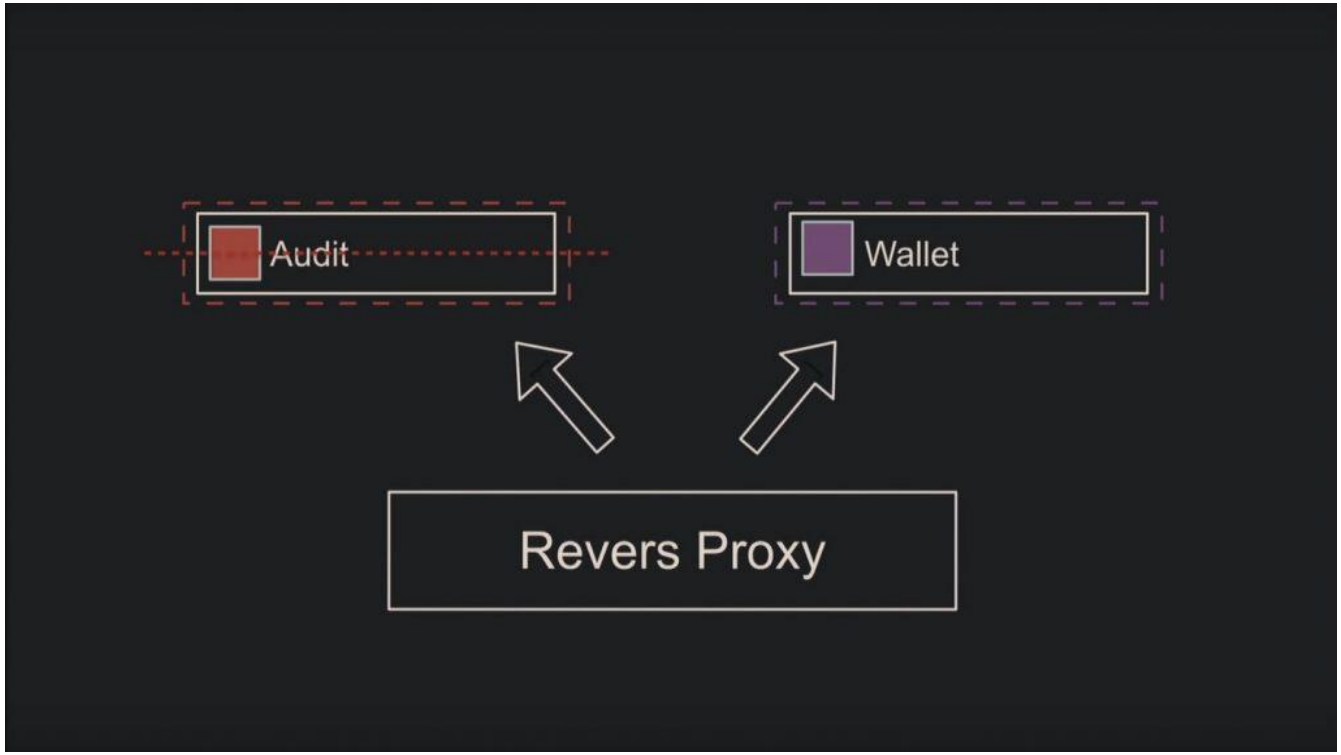
Пример разделения сервисов

Достоинства и недостатки микросервисной архитектуры:

Как в любой распределённой архитектуре, получим накладные расходы на коммуникацию.

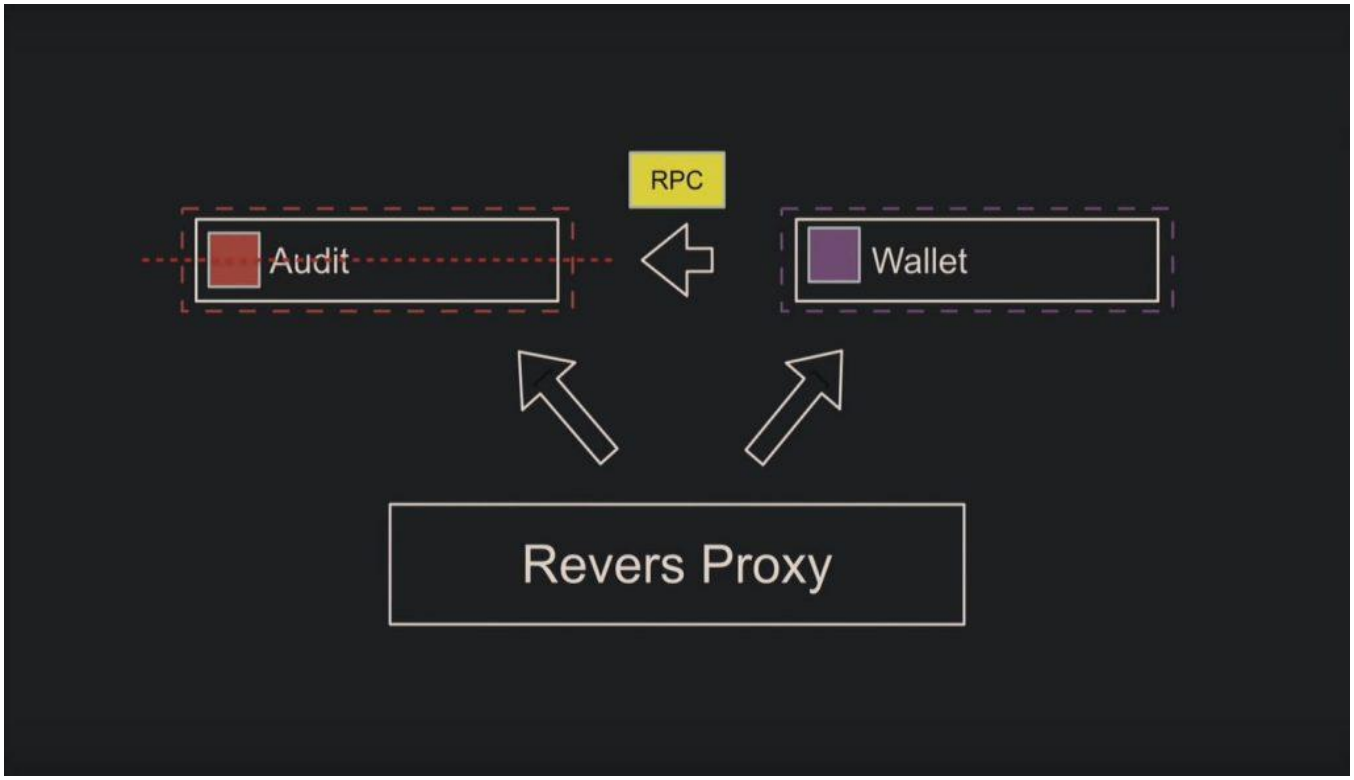
Концепция непрерывной интеграции и доставки (CI/CD) и построение архитектуры (контейнеризация, оркестрация, мониторинг и другое) требует большого количества времени.

Отказоустойчивость:

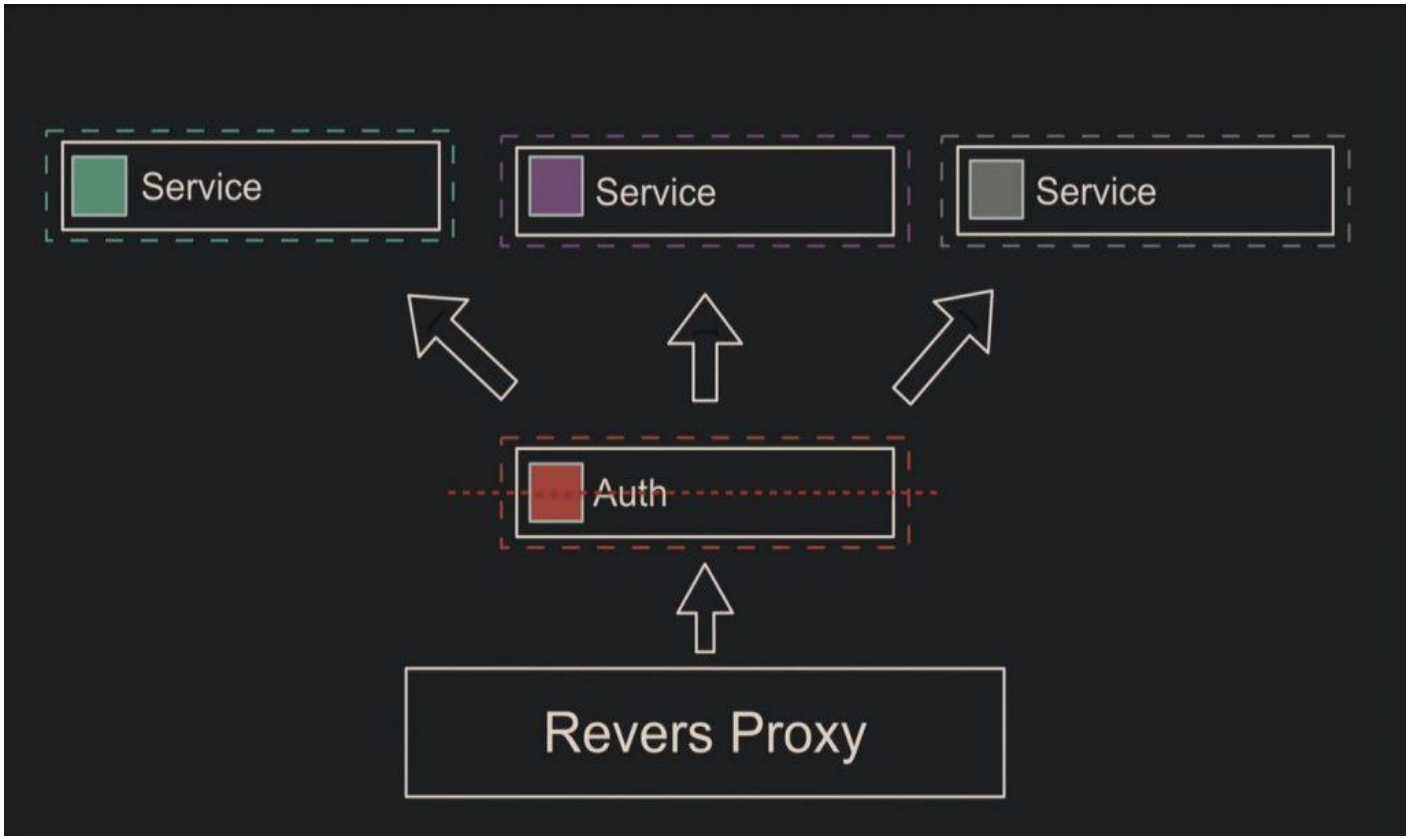


Часто её определяют как «падение одного сервиса не отражается других». Представьте, падает Audit, а Wallet теоретически продолжает работать – похоже на отказоустойчивость.

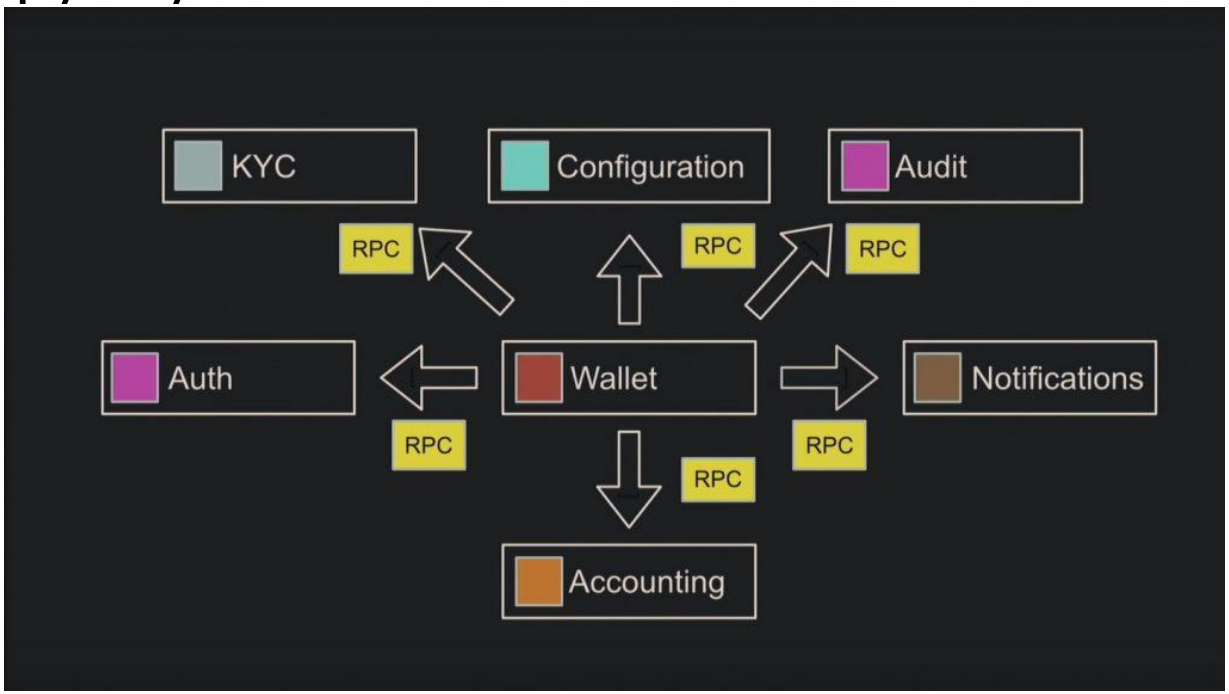
А как же запросы по RPC, которые Wallet продолжает слать? Необходимо программно предусмотреть ситуацию, когда Audit не отвечает, и грамотно настроить rollback, поскольку базы разные, и транзакционно это сделать не получится.



Или другая ситуация: падает микросервис авторизации, через который ходят другие. Чтобы продолжать обрабатывать запросы, добавляют код для неавторизованного пользователя. По существу, это мощный отказ.



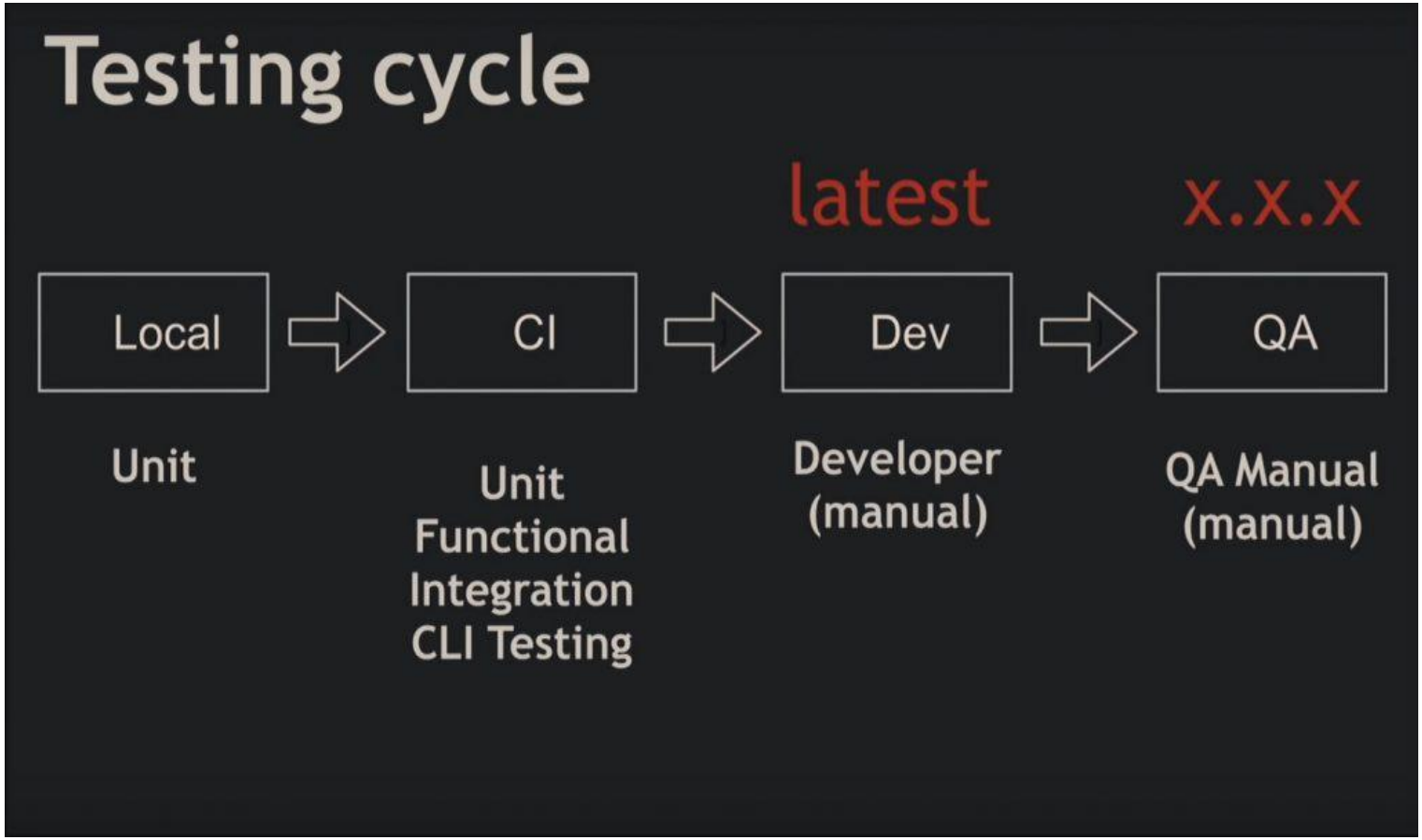
Стандартный процесс разработки – кодинг, тестирование и развёртывание – в микросервисной архитектуре выглядит иначе. Первые два этапа сливаются, поскольку микросервис взаимодействует с массой других. Чтобы локально сделать хоть один запрос, придётся запустить все эти микросервисы, поэтому тестирование вручную не подходит для подобной задачи.



Локально разработчик проводит юнит-тестирование, где вместо ответов микросервисов будут mock-объекты.

Ещё понадобятся функциональные тесты, например, для отлавливания проблем коммуникации, а также интеграционные тесты. Они прогоняются вместе с юнит-тестами на этапе слияния рабочий копий в главную ветку разработки.

И только потом программист проверяет функциональность руками. До развёртывания релизную версию тестирует QA.



Тестирование микросервисов

Микросервисная архитектура делает компоненты независимыми при разработке и развёртывании, чего не было в монолите.

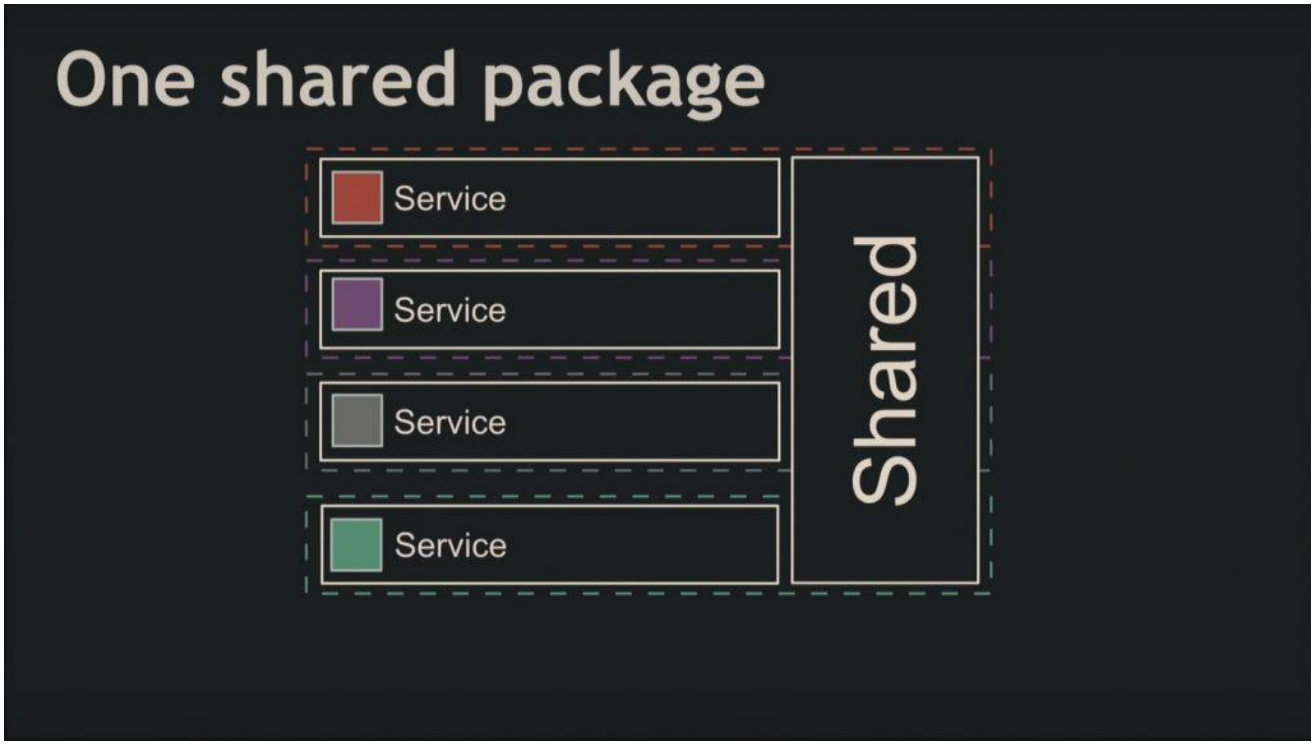
Микросервисы используются повторно и экономят средства в плане бизнеса.

Программист получает относительную свободу в выборе языка и технологий для разработки отдельных частей проекта.

Контроль зависимостей

Трудно сопроводить и поддерживать 50 проектов с 50 репозиториями, если вдруг обнаружится проблема безопасности, которую нужно срочно решить во всех них. Используйте *контроль зависимостей*, общие библиотеки для микросервисов и семантическое версионирование. Одну библиотеку разбивайте на ряд небольших, чтобы избежать страха выпуска мажорной версии.

Инвертируйте зависимости: прописывайте версию драйвера в пакет, куда добавляете нужные микросервисы, чтобы потом не пришлось сломя голову перепроверять версии во всех проектах. Для внешних зависимостей лучше зафиксировать версии.



Контроль зависимостей

Базы данных

Поскольку базы данных в микросервисной архитектуре изолированные, вы используете разные их виды одновременно и получаете повышенный уровень безопасности, благодаря общению сервисов только через RPC.

Но что делать, когда объединяемые данные в разных микросервисах?

Вот возможные решения проблемы:

-храните одно и то же значение в двух микросервисах, но появляются трудности с актуализацией данных;

-делайте RPC, правда, усложните работу с большими объёмами информации;

-выгрузите данные из всех баз для аналитики;

-сделайте миграцию данных, что тоже непросто и повлечёт написание RPC.

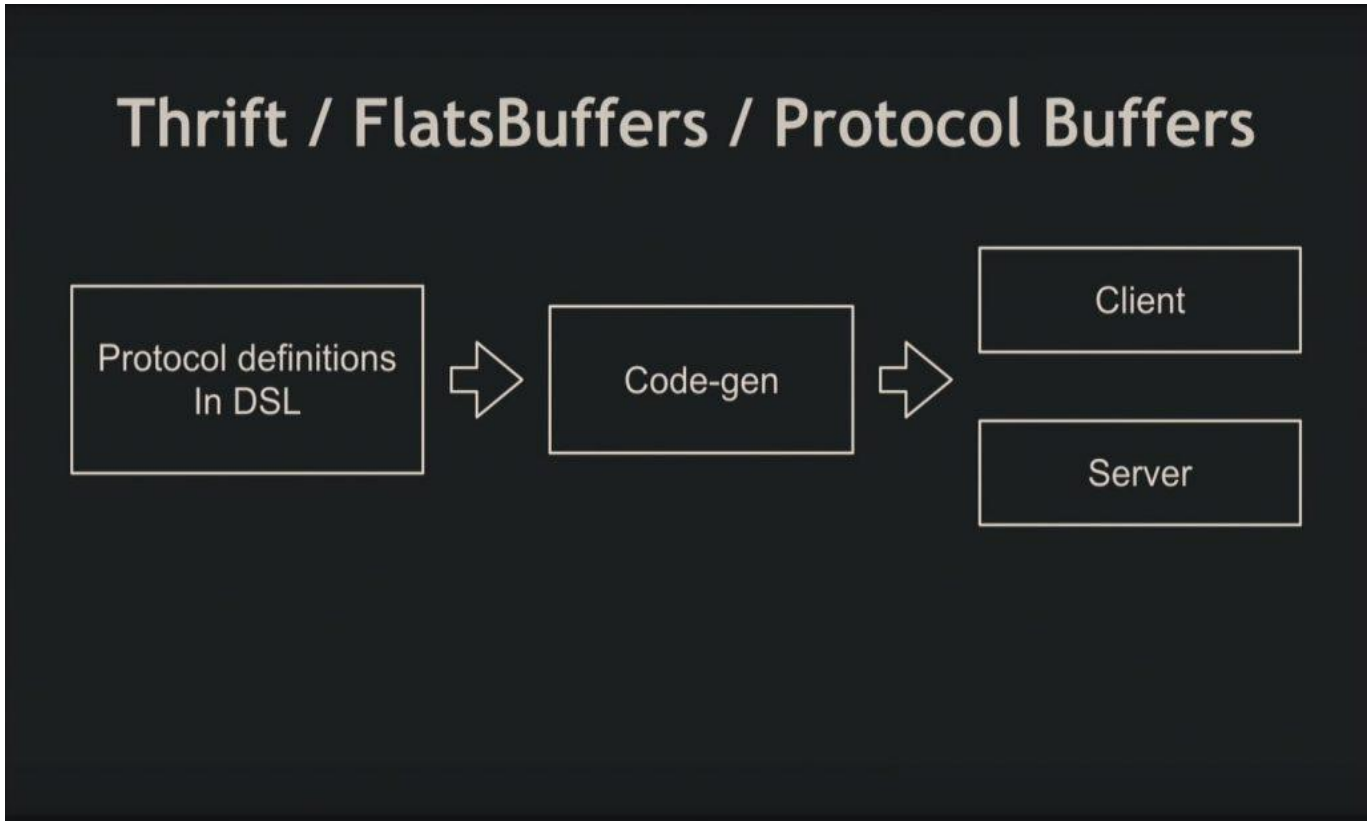
Внутренняя коммуникация в микросервисной архитектуре

Для общения микросервисам нужен контракт: протокол и валидация данных. С последним справляется JSON Schema, но протокол также необходим.

Требования при выборе способа коммуникации:

-строгость;
-общий протокол для сервера и клиента;
-генерация кода для любого языка;
-производительность.

В качестве протоколов используют Protocol Buffers, FlatBuffers, Apache Thrift. Сначала вы пишете предметно-ориентированный язык, отдаёте это программе-генератору кода и получаете сгенерированный клиент и сервер.



Организация работы в команде

Команды делят по технологиям и следят за их размерами (не более 7–8 человек). Важно, чтобы программисты взаимодействовали между собой не только в локальной группе по конкретной задаче, но и с другими. Тогда в области их знаний будет много общего:

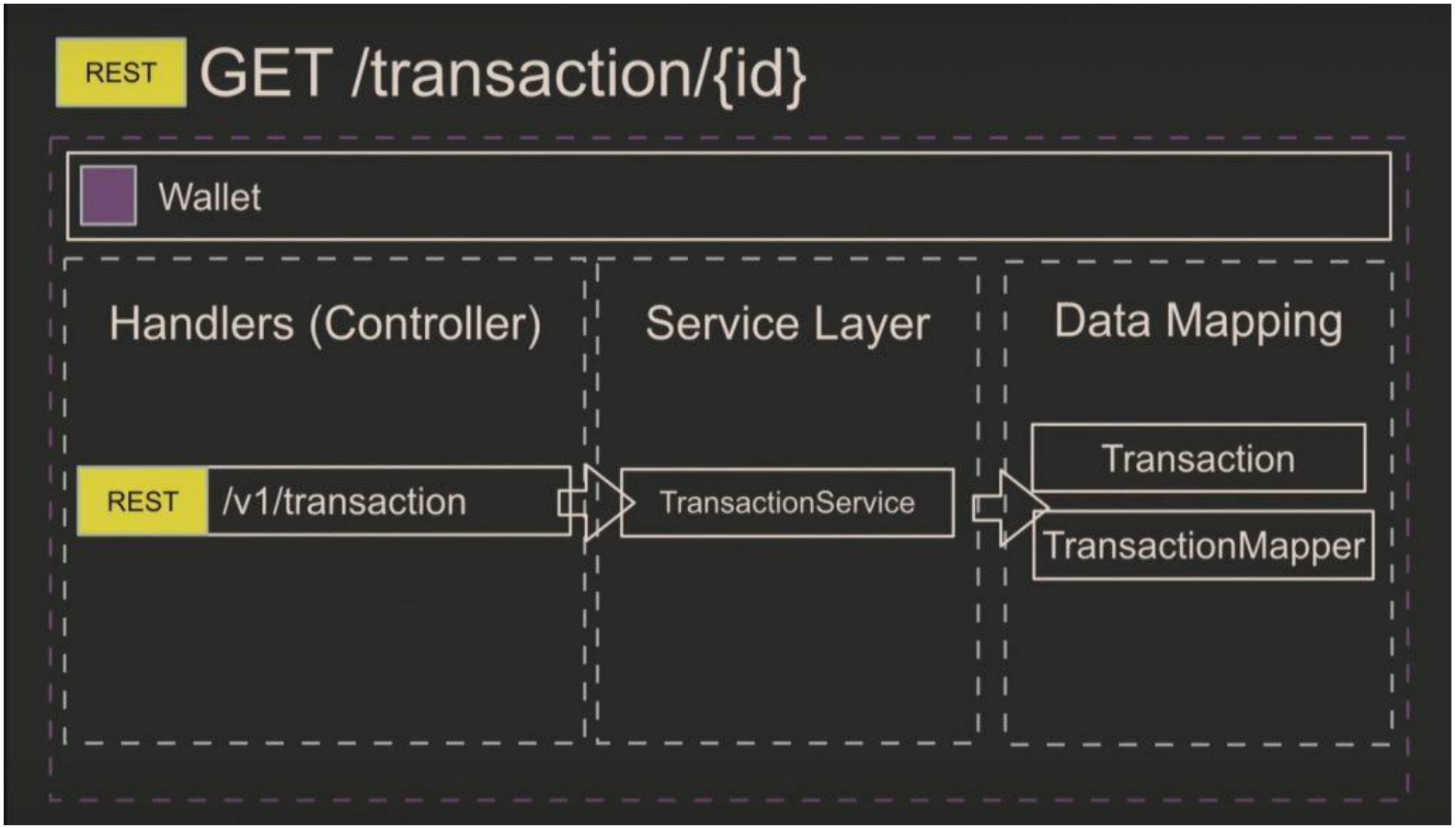
-язык;
-организация и шаблонизация кода для каждого микросервиса;
-библиотеки;
-концепция непрерывной интеграции и доставки;
-протокол коммуникации;
-документация.

Устройство микросервисов

Микросервисы состоят из трёх слоёв: небольших обработчиков, бизнес-логики и мапперов данных.

В сервисном слое сосредотачивается 99% всего кода.

Поскольку в микросервисе несколько обработчиков, используйте Data Transfer Object (DTO), к которому вы будете приводить GET-запрос. Это облегчает обработку и валидацию.



Последовательность выполнения запроса

Заключение

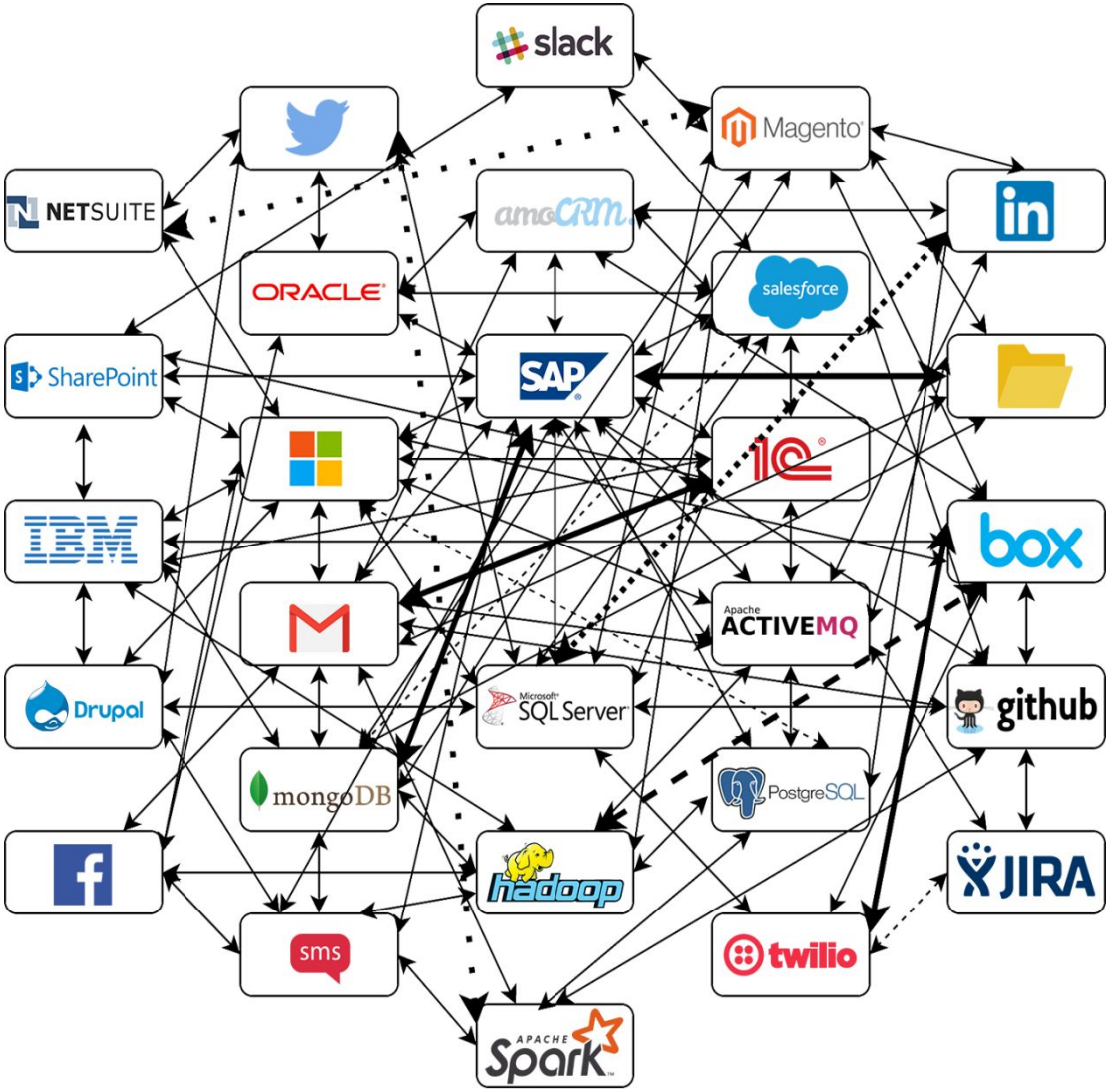
Принимайте решение об использовании микросервисной архитектуры, только чётко осознав и взвесив все достоинства и недостатки.

Вам будут нужны знания о создании распределённой архитектуры и возможность реализации необходимых структурных элементов со стороны бизнеса.

Ведь без логирования, мониторинга, трассировки, непрерывной интеграции и оркестрации вы получаете огромную массу, порой неразрешимых, проблем.

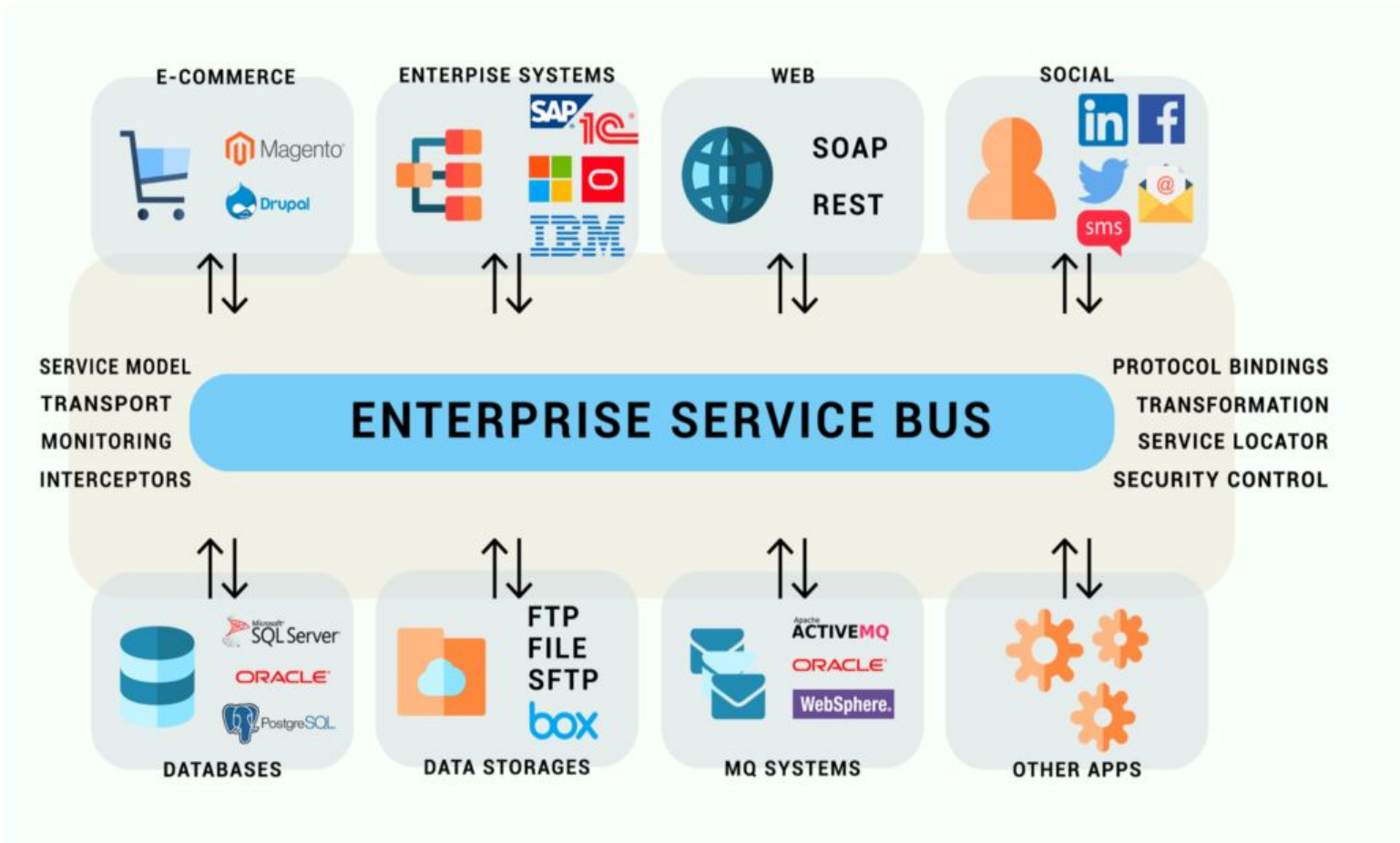
Необходимость использования Шины Данных (Enterprise Service Bus, ESB)

По мере развития любой компании появляются новые бизнес-процессы, требующие автоматизации, усложняются схемы взаимодействия IT-систем. Таким образом, по прошествии нескольких лет многие IT-директора сталкиваются с проблемой: в состав используемого ПО входит целый набор «проверенных временем» систем, но при этом взаимодействие между ними реализовано лишь частично, плохо структурировано, не подчинено единому стандарту, а необходимость создания новой интеграции IT-систем почти всегда требует использования собственных разработок или приобретения еще одного дорогостоящего программного продукта.



Многообразие IT систем на предприятии

В начале 2000 годов на рынке программного обеспечения стали появляться решения, сформировавшие кластер под названием Сервисная шина масштаба предприятия (Enterprise Service Bus, ESB), или сокращенно Шина Данных. Шина Данных – это, в первую очередь, концепция, элемент архитектуры IT-ландшафта, используемый для решения задачи интеграции разрозненных информационных систем в единый программный комплекс с централизованным управлением передачей информации и применением сервис-ориентированного подхода.



Enterprise Service Bus (ESB)

Архитектура ESB строится на 3 компонентах:

-набор коннекторов;
-очередь сообщений;
-платформа.

Коннекторы используются для подключения к различным системам и обеспечивают прием и отправку данных.

Очередь сообщений (Message Queue, MQ) служит для организации промежуточного хранения сообще-ний в ходе их доставки.

Платформа обеспечивает связь коннекторов с очередью, а также организацию асинхронной передачи информации между источниками и приемниками с гарантированной доставкой сообщений и возможностью трансформации.

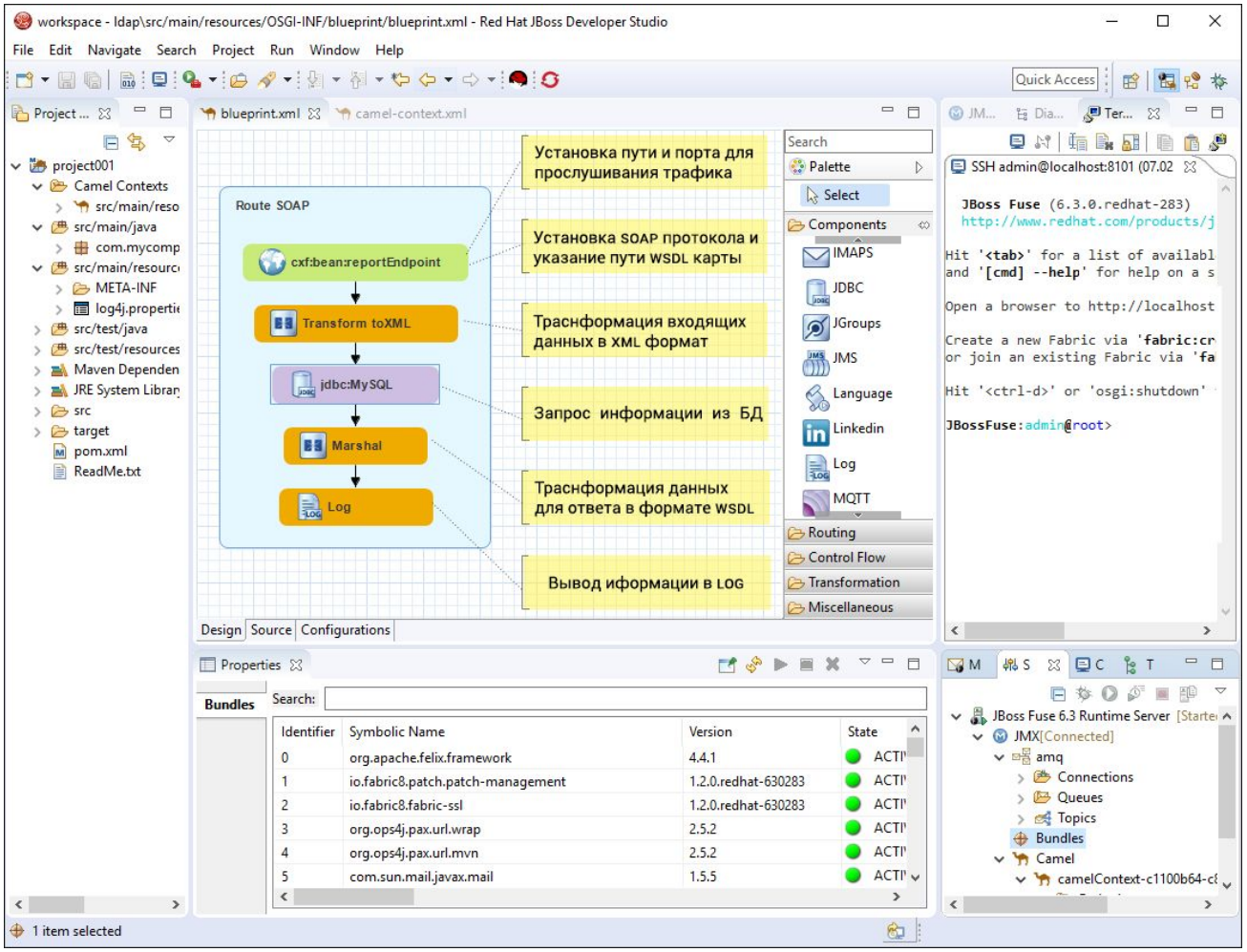
В состав платформы входит средство разработки, позволяющее не только задать правила маршрутизации, но также, при необходимости, определить собственные коннекторы, в т.ч. с использованием внешних процедур, реализованных на языках Java, C, C++, C#, Python и др.

К основным преимуществам современных ESB-решений относятся:

-широкий набор коннекторов и масштабируемость решения;
-гибкая маршрутизация данных;
-гарантированная доставка информационных сообщений;
-организация безопасного канала передачи;
-централизованное управление;
-возможность мониторинга и диагностики состояния передачи;
-возможность интеграции с очередью сообщений стороннего производителя.

К настоящему времени на рынке представлено более двух десятков шин данных, однако наибольшее распространение получили следующие решения:

-Integration Bus (IBM);
-Oracle Service Bus (Oracle);
-BizTalk (Microsoft);
-ActiveMatrix Service Bus (TIBCO);
-MuleESB (MuleSoft);
-JBoss Fuse ESB (Red Hat).



Red hat JBoss Developer Studio

Внедрение Шины Данных в IT-ландшафт организации позволяет не только структурировать, привести к единому стандарту и упростить поддержку процедур обмена информацией между системами, но также снизить временные затраты на интеграцию новых подсистем и, как следствие, сократить стоимость поддержки и развития всей IT-инфраструктуры компании.

Протоколы сообщений между сервисами

В каждой отрасли бизнеса, каждой компании, используется разнообразнейшее ПО. За десятилетия существования веба как отрасли сформировались следующие практики межсетевого взаимодействия:

-Обмен файлами по FTP.
-Неструктурированные HTTP-запросы, договорённости между разработчиками.
-Веб-сервисы.
-Экзотика: сокеты, порты, бинарные объекты.

Веб-сервисы (или веб-службы) — это технология, позволяющая системам обмениваться данными друг с другом через сетевое подключение. Обычно веб-сервисы работают поверх протокола HTTP или протокола более высокого уровня. Веб-сервис — просто адрес, ссылка, обращение к которому позволяет получить данные или выполнить действие.

Главное отличие веб-сервиса от других способов передачи данных: стандартизированность. Приняв решение использовать веб-сервисы, можно сразу переходить к структуре данных и доступным функциям. Например, В SOAP (как более строгом протоколе), уже решён вопрос уведомления об ошибках.

Самые известные способы реализации веб-сервисов:
-XML-RPC (XML Remote Procedure Call) — протокол удаленного вызова процедур с использованием XML.
-SOAP (Simple Object Access Protocol) — стандартный протокол по версии W3C.
-JSON-RPC (JSON Remote Procedure Call) — более современный аналог XML-RPC.
-REST (Representational State Transfer) — архитектурный стиль взаимодействия компьютерных систем в сети основанный на методах протокола HTTP.
-Специализированные протоколы для конкретного вида задач, такие как GraphQL.
-Менее распространенный, но более эффективный gRPC, передающий данные в бинарном виде и использующий HTTP/2 в качестве транспорта.

SOAP

SOAP (Simple Object Access Protocol) — Данные передаются в формате XML.

Преимущества:
-отраслевой стандарт по версии W3C;
-наличие строгой спецификации;
-широкая поддержка в продуктах Microsoft,
-однозначность.

Недостатки:
-сложность реализации;
-сложность/ресурсоемкость парсинга XML-данных.

Любое сообщение в протоколе SOAP — это XML документ, состоящий из следующих элементов (тегов):

Envelope. Корневой обязательный элемент. Определяет начало и окончание сообщения.
Header. Необязательный элемент — заголовок. Содержит элементы, необходимые для обработки самого сообщения. Например, идентификатор сессии.
Body. Основной элемент, содержит основную информацию сообщения. Обязательный.
Fault. Элемент, содержащий информацию об ошибках, возникающих в процессе обработки сообщения. Необязательный.

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="https://www.w3.org/2003/05/soap-envelope/"
               soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding/">
    <soap:Body>
      <m:GetPrice xmlns:m="https://www.w3schools.com/prices">
        <m:Item>Apples</m:Item>
      </m:GetPrice>
    </soap:Body>
</soap:Envelope>
```

Пример SOAP запроса

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="https://www.w3.org/2003/05/soap-envelope/"
               soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding/">
    <soap:Body>
      <m:GetPriceResponse xmlns:m="https://www.w3schools.com/prices">
        <m:Price>1.90</m:Price>
      </m:GetPriceResponse>
    </soap:Body>
</soap:Envelope>
```

Пример SOAP ответа

REST

REST (Representational State Transfer) — на самом деле архитектурный стиль, а не протокол. В отличие от SOAP, REST не подкреплен официальным стандартом. Фактически, он основывается на соглашениях.

Веб-сервис, построенный с учетом всех требований и ограничений архитектурного стиля, можно назвать RESTful веб-сервисом.

REST не использует конвертацию данных при передаче, данные передаются в исходном виде — это снижает нагрузку на клиент веб-сервиса, но увеличивает нагрузку на сеть.

Управление данными происходит с помощью методов HTTP:

-GET — получить данные;
-POST — добавить данные;
-PUT — изменить данные;
-DELETE — удалить данные.

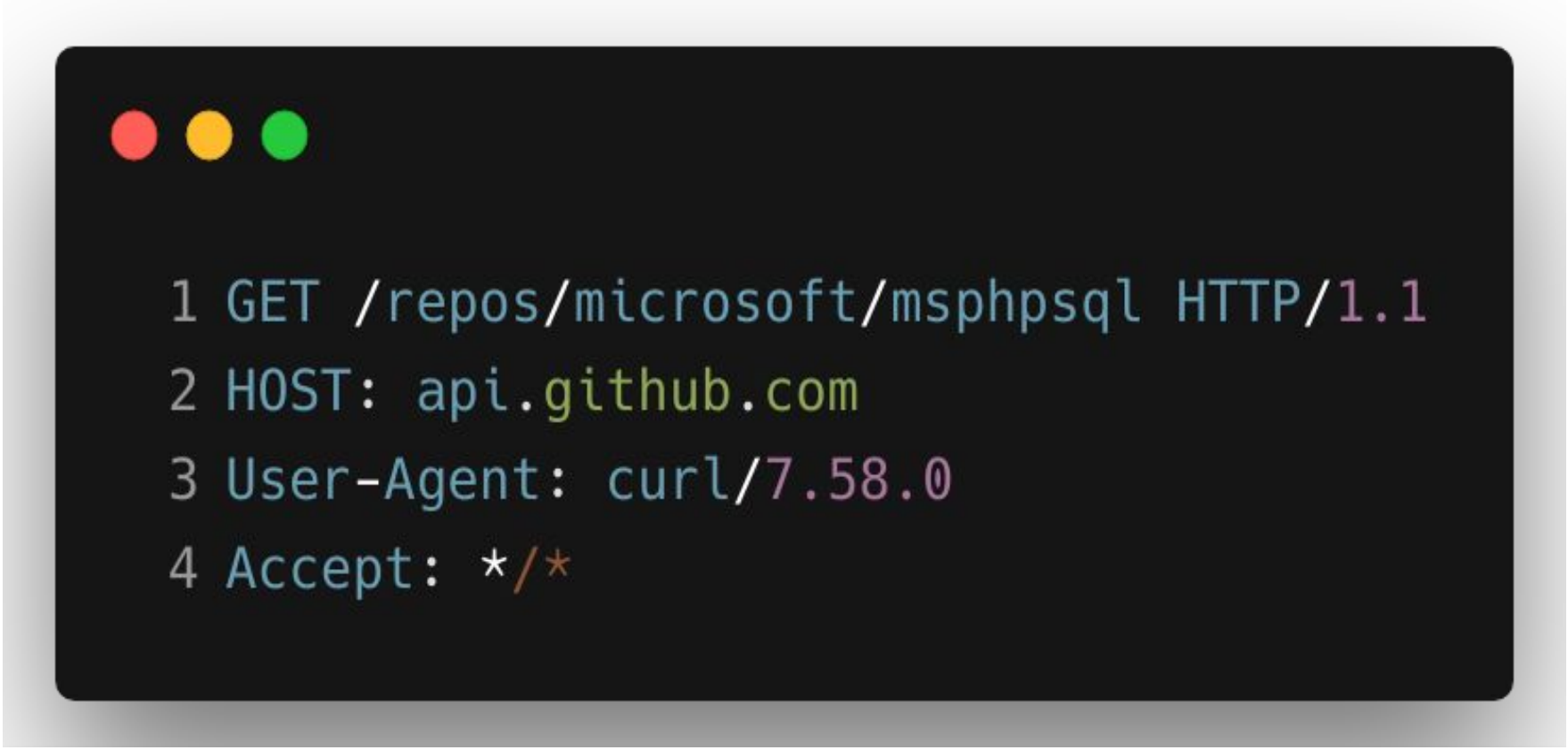
Использование этих методов позволяет реализовать типичный CRUD (Create/Read/Update/Delete) для любой информации. Но это лишь соглашение: часто используются только 2 метода: GET для получения и POST для всего остального. Разобраться поможет такое понятие, как REST-Patterns . Паттерны связывают HTTP методы с тем, что они делают.

Преимущества:

-простота реализации;
-экономичность в плане ресурсов;
-не требует программных надстроек (json_decode есть почти в каждом языке).

Недостатки:

-отсутствие спецификации;
-неоднозначность методов управления данными.



Пример REST запроса

Пример REST ответа

SOAP используется в крупных корпоративных системах со сложной логикой, когда требуются четкие стандарты, подкрепленные временем. XML-RPC устарел и не имеет смысла ввиду наличия JSON-RPC. RPC-протоколы подойдут для совсем простых систем с малым количеством единиц информации и API-методов.

Если же вы разрабатываете публичное API и логика взаимодействия во многом покрывается четверкой методов CRUD — подойдет REST. Он наиболее популярен в WEB. Яндекс, Google и другие используют именно его для своего API.

ЛЕКЦИЯ 4_2. ОРКЕСТРАЦИЯ, ОБНАРУЖЕНИЕ МИКРОСЕРВИСОВ, K&S.

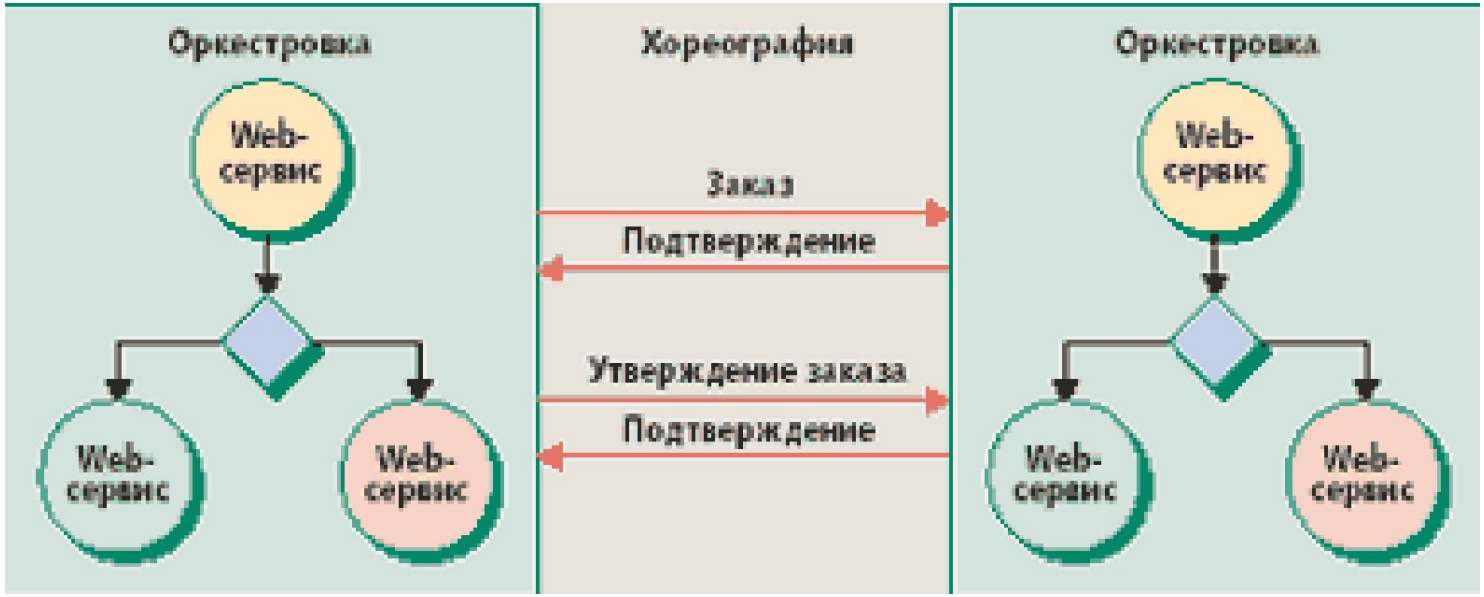
Оркестровка представляет собой единый централизованный исполняемый бизнес-процесс (Orchestrator), который координирует взаимодействие между различными службами.

В сервис-ориентированной архитектуре оркестровка сервисов реализуется согласно стандарту Business Process Execution Language (WS-BPEL).

… **Хореография** описывает взаимодействие между несколькими службами, в то время как **оркестровка** представляет контроль с точки зрения одной из сторон.

Термины оркестровка и хореография описывают два аспекта разработки бизнес-процессов на основе объединения Web-сервисов.

На рисунке в общем виде показана взаимосвязь этих аспектов, которые в какой-то мере дополняют друг друга.



Оркестровка относится к исполняемому процессу, а хореография позволяет отслеживать последовательности сообщений его участников

Оркестровка относится к исполняемому бизнес-процессу, который может взаимодействовать с внешними и внутренними Web-сервисами.

Взаимодействия на основе обмена сообщениями включают в себя бизнес-логику и порядок выполнения задач.

Они могут выходить за границы приложений и предприятий, определяя многошаговую транзакционную бизнес-модель.

Системы обнаружения сервисов автоматизируют процесс, позволяя получить ответ на вопрос, где работает нужный сервис, и изменить настройки в случае появления нового или отказа. Обычно под этим подразумевают набор сетевых протоколов (Service discovery protocols), обеспечивающих нужную функцию, хотя в современных реализациях это уже часть архитектуры, позволяющей обнаруживать связанные компоненты.

На сегодня существует несколько решений, реализующих хранение информации об инфраструктуре, — как относительно сложных, использующих key/value-хранилище и гарантирующих доступность (ZooKeeper, Doozer, etcd),

так и простых (SmartStack, Eureka, NSQ, Serf). Но, предоставляя информацию, они не слишком удобны в использовании и сложны в настройках.

Стандарты оркестровки и хореографии должны удовлетворять ряду технических требований, обеспечивающих разработку бизнес-процессов с привлечением Web-сервисов.

Эти требования касаются как языка, служащего для описания последовательности действий в бизнес-процессе, так и инфраструктуры для его выполнения.

Во-первых, для обеспечения надежности и универсальности, необходимых современным вычислительным средам, важна возможность асинхронного обращения к сервису.

Повысить производительность процесса позволяет возможность одновременного обращения к нескольким сервисам.

Во-вторых, необходимо, чтобы архитектура бизнес-процесса обеспечивала управление исключительными ситуациями и целостностью транзакций.

Кроме обработки ошибок и тайм-аутов оркестрованные Web-сервисы должны гарантировать доступность ресурсов при выполнении длительных распределенных транзакций.

Традиционные транзакции обычно не вполне пригодны для реализации длительных распределенных бизнес-операций, поскольку не позволяют достаточно долго блокировать необходимые ресурсы.

В-третьих, оркестровка Web-сервисов должна быть динамичной, гибкой и адаптивной, чтобы отвечать изменяющимся потребностям бизнеса.

Достижению гибкости способствует четкое разделение логики процесса и используемых Web-сервисов, обычно обеспечиваемое механизмом оркестровки.

Он обрабатывает поток работ бизнес-процесса, вызывая соответствующие Web-сервисы и определяя, какие шаги следует выполнить.

Если в традиционных вариантах сервис-ориентированной архитектуры модули могут быть сами по себе достаточно сложными программными системами, а взаимодействие между ними зачастую полагается на стандартизованные тяжеловесные протоколы (такие, как SOAP, XML-RPC), в микросервисной архитектуре системы выстраиваются из компонентов, выполняющих относительно элементарные функции, и взаимодействующие с использованием экономичных сетевых коммуникационных протоколов (в стиле REST с использованием, например, JSON, Protocol Buffers, Thrift).

Свойства, характерные для микросервисной архи-тектуры:

– модули можно легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления каждого из микросервисов;

– модули организованы вокруг функций: микросервис по возможности выполняет только одну достаточно элементарную функцию;

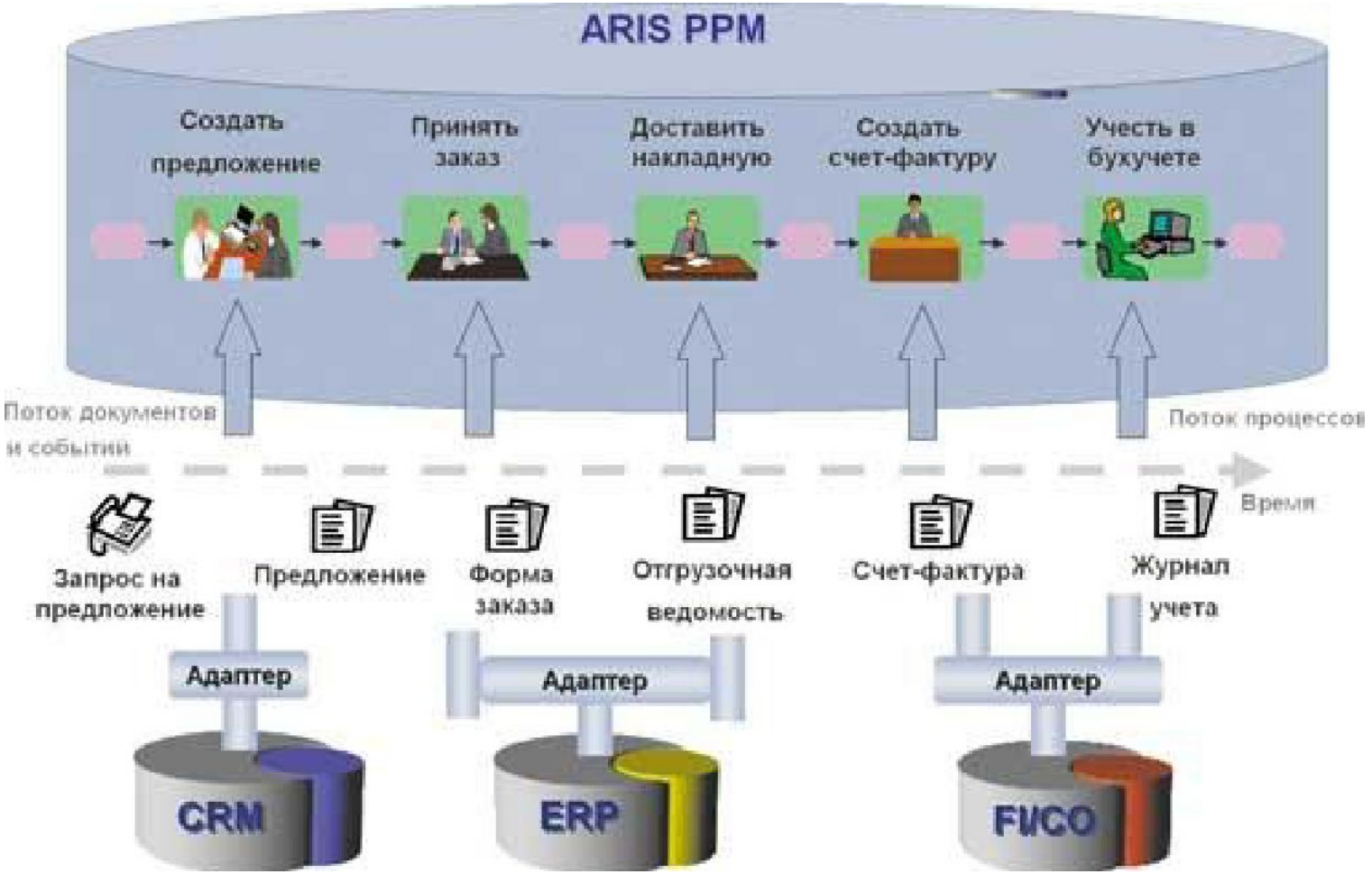
-модули могут быть реализованы с использованием различных языков программирования, фреймворков, связующего программного обеспечения, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных аппаратных платформах: приоритет отдаётся в пользу наибольшей эффективности для каждой конкретной функции, нежели стандартизации средств разработки и исполнения;

– архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые.

Наиболее популярная среда для выполнения микросервисов — системы управления контейнеризованными приложениями (такие как **Kubernetes** и её надстройки OpenShift и CloudFoundry, Docker Swarm, Apache Mesos), в этом случае каждый из микросервисов как правило изолируется в отдельный контейнер или небольшую группу контейнеров, доступную по сети другим микросервисам и внешним потребителям, и управляется средой оркестрации, обеспечивающей отказоустойчивость и балансировку нагрузки.

В последнее время получили развитие альтернативные подходы к созданию веб-сервисов, основанные на архитектурном стиле REST («RESTful-веб-сервисов»). Разработка методов создания грид-сервисов на основе этого архитектурного стиля позволяет упростить интерфейсы грид-сервисов, тем самым расширяя возможности их прямого использования из прикладных программ.

Грид-сервисы обеспечивают «оркестровку», то есть последовательный или одновременный запуск отдельных шагов композитных заданий в соответствии с заданной логикой и отслеживание процесса их выполнения.

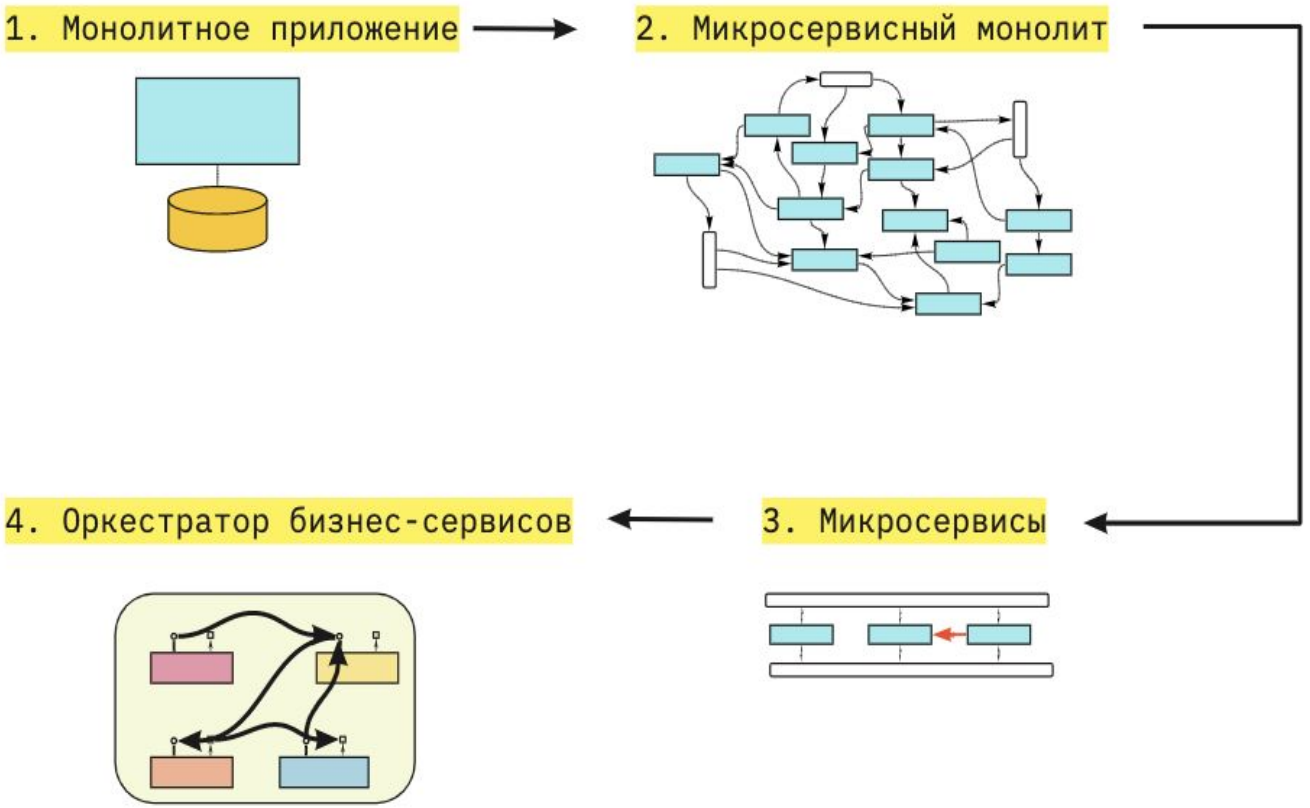


Технология работы процессно-ориентированной аналитической системы.

От микросервисного монолита к оркестратору

Когда компании решают разделить монолит на микросервисы, в большинстве случаев они последовательно проходят четыре этапа:

- монолит,
- микросервисный монолит,
- микросервисы,
- оркестратор бизнес-сервисов.



Четыре этапа перехода от монолита к микросервисам

Этап №1. Монолит

1.1 Характеристики

Обычно монолитную архитектуру можно описать так:

-Единая точка разработки и деплоя;

-Единая база данных;

-Единый цикл релиза для всех изменений;

-В одной системе реализовано несколько бизнес-задач.

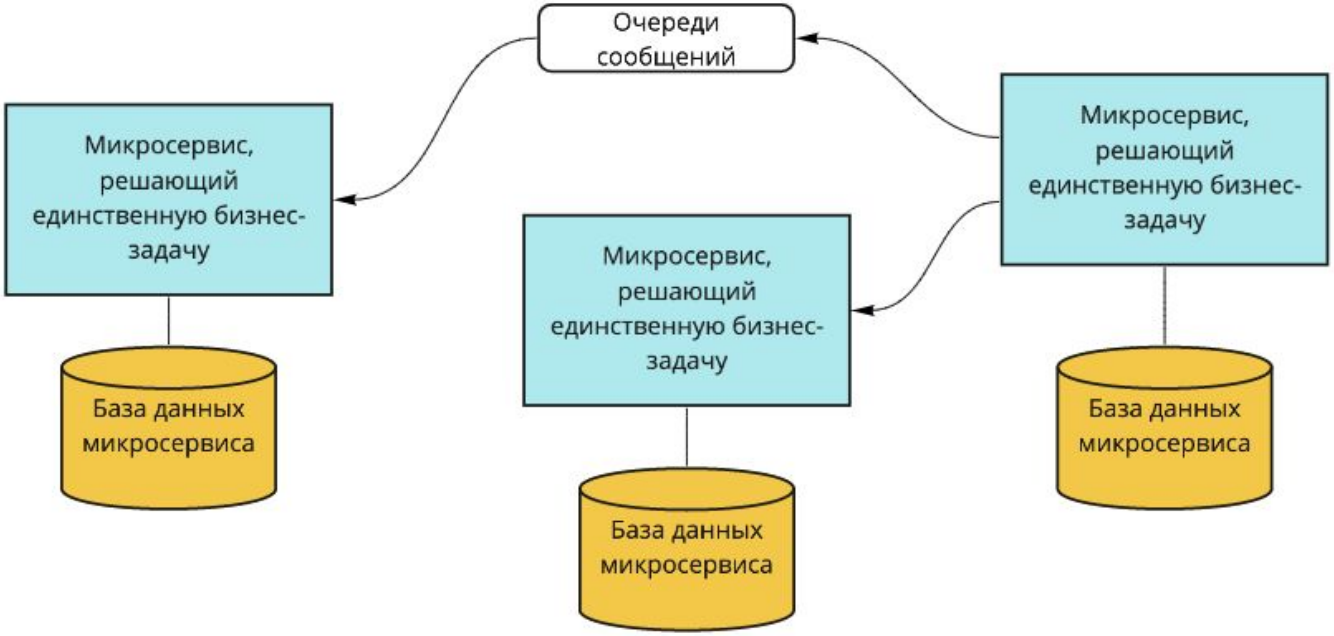
Монолитное приложение

Как перейти на следующий этап

В основе процесса выделения микросервисов лежит вынесение бизнес-задач из монолита в отдельные сервисы.

При этом нужно руководствоваться принципом единственности ответственности, который перефразируется так: у микросервиса должна быть только одна причина для изменения. Этой причиной является изменение бизнес-логики той единственной задачи, за которую он отвечает.

Первые микросервисы



Постепенно у вас образуется набор из микросервисов, где каждый отвечает лишь за свою бизнес-задачу

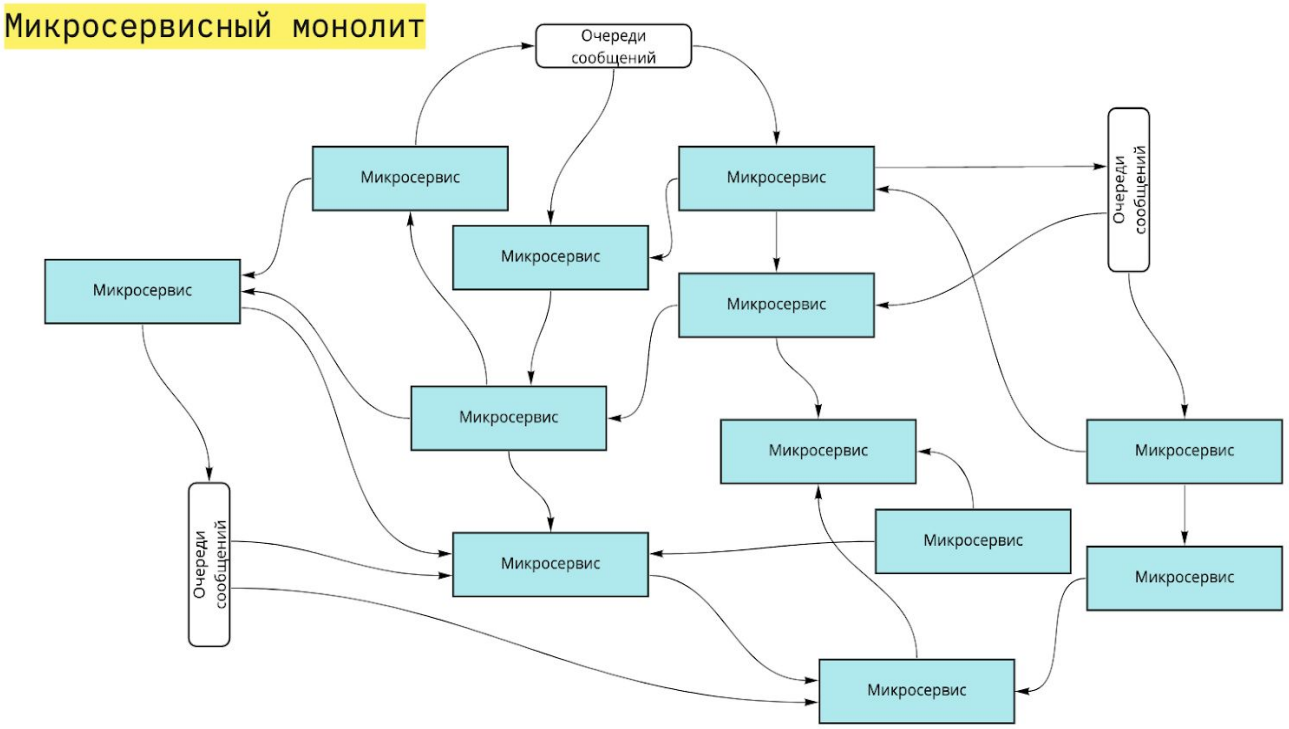
Этап №2. Микросервисный монолит

Характеристики

Все части монолита стали независимыми микросервисами и эти микросервисы должны общаться между собой. Если раньше, находясь внутри одного процесса, сервисы вызывали методы друг друга напрямую, то теперь нужно интегрироваться.

Из четырех способов интеграции в микросервисной архитектуре обычно не используют обмен файлами и стараются не использовать shared database, зато активно работают с RPC и очередью сообщений.

Получается, что все части монолита распались на микросервисы, а их обратно соединили паутиной синхронных и асинхронных интеграций:



По факту, получился тот же монолит, но с б*о*льшим количеством новых проблем.

Проблемы

-Прямые связи между микросервисами усложняют анализ проблем. Например, запрос может пройти через 5 микросервисов, прежде, чем вернуться с ответом. Что если на третьем микросервисе запрос завис? Что если там была ошибка? Что если на втором шаге должно было создаться сообщение в очередь, но оно не появилось? Возникает сложность с разбором проблем.

-Предыдущий пункт усложняется, если у микросервиса много экземпляров. Тогда возникает ситуация, что запрос пришел на экземпляр, который завис.

-Архитектуру сложно понять и, чем больше сервисов вы добавляете, тем запутанней всё становится. В целом, добавление новых сервисов нелинейно повышает сложность архитектуры.

-Неизвестно, кто потребители вашего API, что добавляет сложности в проектировании API и его изменении.

Если на пути рефакторинга монолита вы остановитесь на этом этапе, то, вполне резонно, сделаете вывод, что с монолитом было лучше и дешевле.

Как перейти на следующий этап

Основные идеи: локализовать точки интеграции и контролировать все потоки данных. Чтобы этого добиться, надо использовать:
-API Gateway для локализации синхронных взаимодействий и моннитоpинг/логирование трафика между микросервисами. В идеале, надо иметь визуализацию трассировки любого запроса.
-Service Discovery для отслеживания работоспособности экземпляров микросервиса и перенаправление трафика на "живые" экземпляры.
-Запретить прямые вызовы между микросервисами.

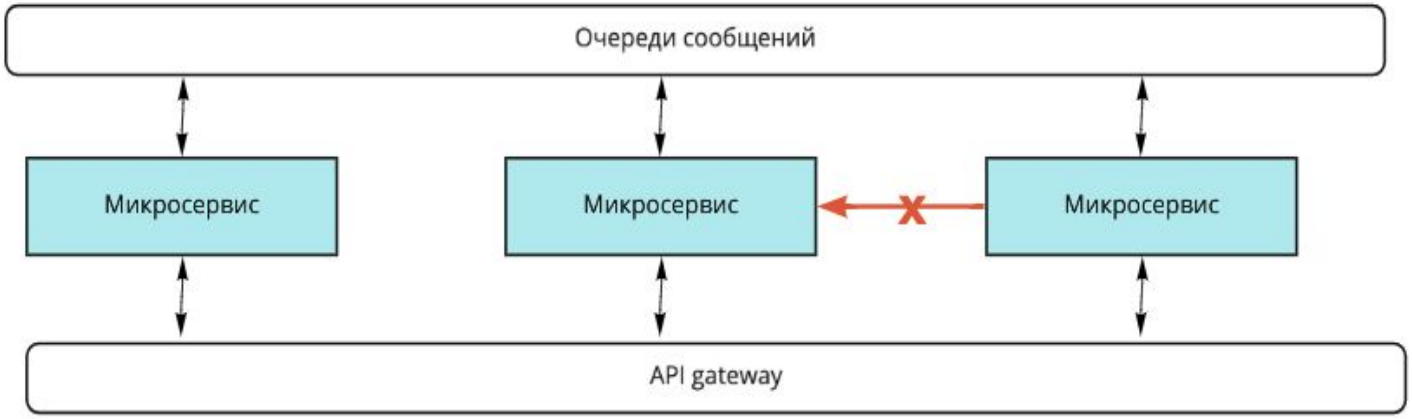
Этап №3. Микросервисы

Характеристики

Микросервисы ничего не знают о существовании друг друга: работают со своей базой данных, API и сообщениями в очереди.

Каждый микросервис решает только одну бизнес-задачу и старается делать это максимально эффективно, за счет выбора технологий, стратегии масштабирования.

Микросервисы



Становится заметна главная черта хорошей архитектуры: сложность системы растет линейно с увеличением количества микросервисов.

Проблемы

На этом этапе сложные технические задачи решены, поэтому начинаются проблемы на уровне бизнес-задач:

Среди сотен микросервисов и разных API бизнес не может понять, какие инструменты есть у него в руках. Пазл складывается в стройные картинки только у энтерпрайз архитекторов, а их, как известно, очень мало на Земле.

Бизнес хочет увидеть лес за деревьями, чтобы понимать, какие есть детали и как из них можно собирать новые продукты, не прибегая к разработке.

Сборку новых продуктов из существующих кубиков, хочется совместить с продуктовой разработкой, чтобы Владелец продукта сам ориентировался, какие ему доступны ресурсы.

Как перейти на следующий этап

Многие компании не идут дальше, потому что на текущем этапе бизнес-задачи могут решаться уже достаточно быстро и эффективно.

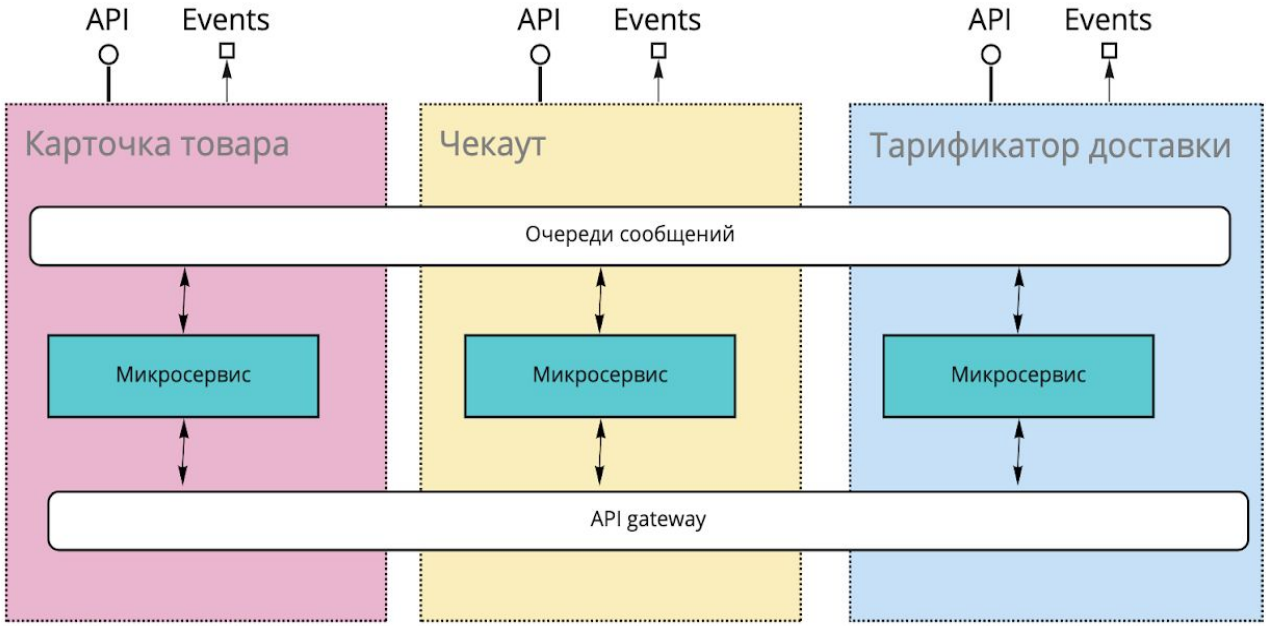
Тем, кто решают двигаться дальше:

Изучите концепцию Citizen Integrator. Для наглядного примера заведите себе пару процессов в Zapier. Опишите микросервисы в виде блоков, решающих бизнес-задачу, и сделайте из них конструктор. Это можно сделать: 1) на готовых инструментах, 2) обернуть BPM-движки типа Camunda, 3) написать всё самим с нуля как, например, сделали в Леруа Мерлен.

Все три подхода жизнеспособны. Выбор подхода зависит от стратегии вашей компании и наличии у вас ИТ-архитекторов и хороших программистов.

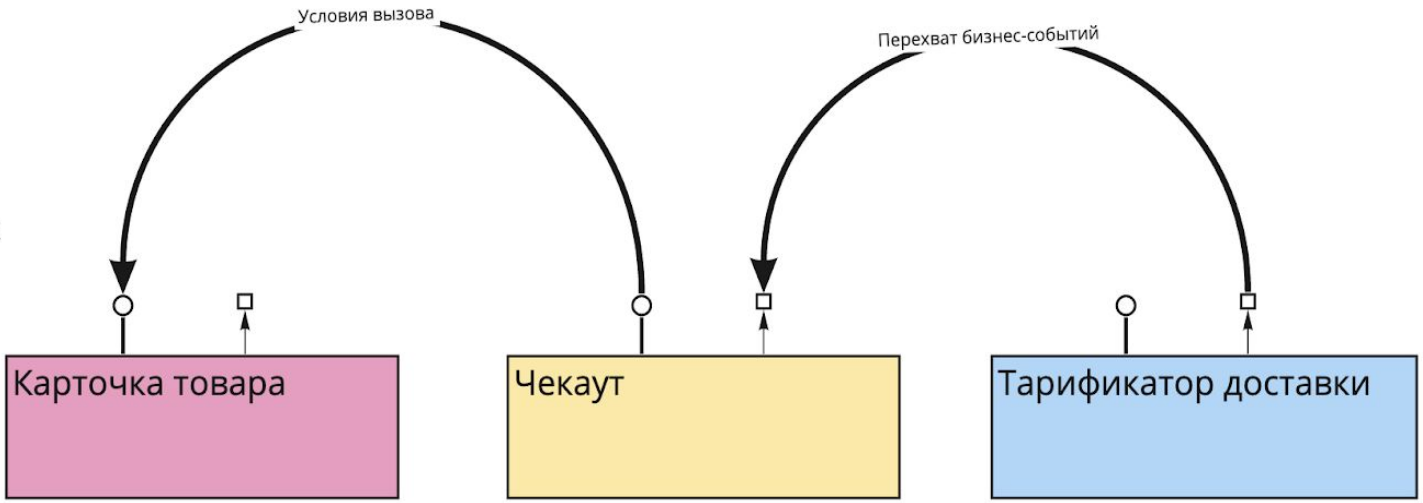
Бизнес-сервисы

Энтерпрайз
архитектор



Оркестратор бизнес-сервисов

не айтишник



Переход к заключительному этапу

Этап №4. Оркестратор бизнес-сервисов

Характеристики

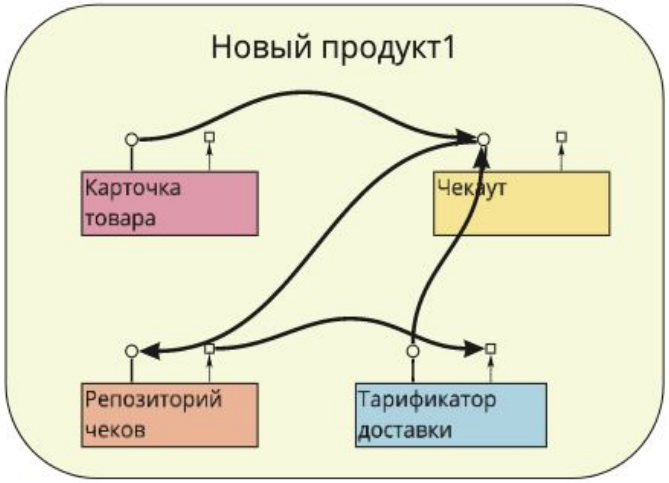
Оркестратор бизнес-сервисов обычно является визуальной платформой, где соединяются сервисы, выставляются триггеры и условия ветвления, контролируются все потоки данных: реализована трассировка запросов, логирование событий, автомасштабирование по условиям.

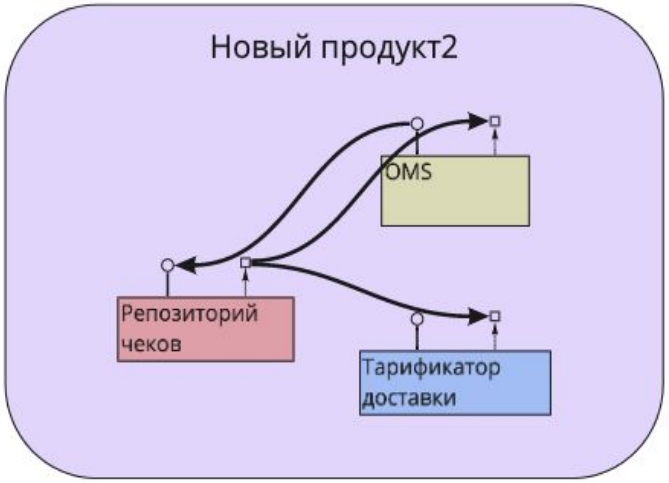
Сам оркестратор ничего не знает о специфике бизнес-процессов, которые на нем крутятся.

На этом этапе можете решить задачу создания продукта в визуальном редакторе.

Если нужных "квадратиков" не хватает, то программисты создают микросервис, учитывая правила описания сервиса для оркестратора, публикуют API и "кубик" появляется в визуальном редакторе, гото вый соединяться с другими участниками бизнес-задачи.

Создание продукта из готовых "кирпичиков"





Создание нового продукта

Проблемы

-Создание, внедрение и развитие оркестратора бизнес-процессов является дорогим удовольствием.

-Если ослабить архитектурный контроль, оркестратор может превратиться в узкое место систем, созданных на нем.

-Чем больше систем создается на оркестраторе, тем больше бизнес зависит от этого решения. В целом, это начинает напоминать проблемы монолита.

Эти четыре этапа показывают естественный ход вещей:

- Вначале приложение небольшое и решает одну бизнес-задачу. Со временем в него добавляют много всего и оно превращается в неповоротливый монолит.

При первой попытке разделить монолит многие команды не готовы к возрастающей сложности. Монолит делится на много микросервисов, но из-за большого количество взаимосвязей получается тот же монолит, только с новыми проблемами: простейшие задачи типа трейсинга запроса или мониторинга инфраструктуры становятся вызовом для команды разработки.

Когда сложности решаются, получается стройная и масштабируемая архитектура. Добавление новых микросервисов линейно повышает сложность.

На последнем этапе приходит бизнес и резонно говорит, что раз есть готовые решения бизнес-задач, то давайте делать новые продукты без разработки. Будем соединять готовые независимые блоки в новые бизнес-процессы через оркестратор.

kubernetes

Kubernetes (K8s) – это программное обеспечение для автоматизации развёртывания, масштабирования и управления контейнеризированными приложениями. Поддерживает основные технологии контейнеризации (Docker, Rocket) и аппаратную виртуализацию.

Kubernetes необходим для непрерывной интеграции и поставки программного обеспечения (CI/CD, Continuos Integration/ Continuos Delivery), что соответствует DevOps-подходу. Благодаря «упаковке» программного окружения в контейнер, микросервис можно очень быстро развернуть на рабочем сервере (production), безопасно взаимодействуя с другими приложениями.

Наиболее популярной технологией такой виртуализации на уровне операционной системы считается Docker, пакетный менеджер которого (Docker Compose) позволяет описывать и запускать многоконтейнерные приложения.

Однако, если необходим сложный порядок запуска большого количества таких контейнеров (от нескольких тысяч), как это бывает в Big Data системах, потребуется средство управления ими – инструмент оркестрации. Именно это считается основным назначением Kubernetes.

При этом **кубернетис – это не просто фреймворк для оркестрации контейнеров**, а целая платформа управления контейнерами, которая позволяет параллельно запускать множество задач, распределённых по тысячам приложений (микросервисов), расположенных на различных кластерах (публичном облаке, собственном датацентре, клиентских серверах и т.д.).

АРХИТЕКТУРА КУБЕРНЕТИС

Kubernetes устроен по принципу *master/slave*, когда ведущим элементом является подсистема управления кластером, а некоторые компоненты управляют ведомыми узлами.

Под узлом (node) понимается физическая или виртуальная машина, на которой работают контейнеры приложений. Каждый узел в кластере содержит инструменты для запуска контейнеризированных сервисов, например, Docker, а также компоненты для централизованного управления узлом.

Также на узлах развернуты **поды (pods) — базовые модули** управления и запуска приложений, состоящие из одного или нескольких контейнеров. При этом на одном узле для каждого пода обеспечивается разделение ресурсов, межпроцессное взаимодействие и уникальный IP-адрес в пределах кластера.

Это позволяет приложениям, развёрнутым на поде, без риска конфликта использовать фиксированные и предопределённые номера портов. Для совместного использования нескольких контейнеров, развернутые на одном поде, их объединяют в **том (volume)** – **общий ресурс хранения**.

Помимо подов, **на ведомых узлах также работают следующие компоненты Kubernetes**:

-Kube-proxy – комбинация сетевого прокси-сервера и балансировщика нагрузки, который, как сервис, отвечает за маршрутизацию входящего трафика на конкретные контейнеры в пределах пода на одном узле. Маршрутизация обеспечивается на основе IP-адреса и порта входящего запроса.

-Kubelet, который отвечает за статус выполнения подов на узле, отслеживая корректность выполнения каждого работающего контейнера.

Управление подами реализуется через API Kubernetes, интерфейс командной строки (Kubectl) или специализированные **контроллеры (controllers)** – процессы, которые переводят кластер из фактического состояния в желаемое, оперируя набором подов, определяемого с помощью селекторов меток.

Селекторы меток (label selector) – это запросы, которые позволяют получить ссылку на нужный объект управления (узел, под, контейнер).

На ведущем компоненте (master) работают следующие элементы:

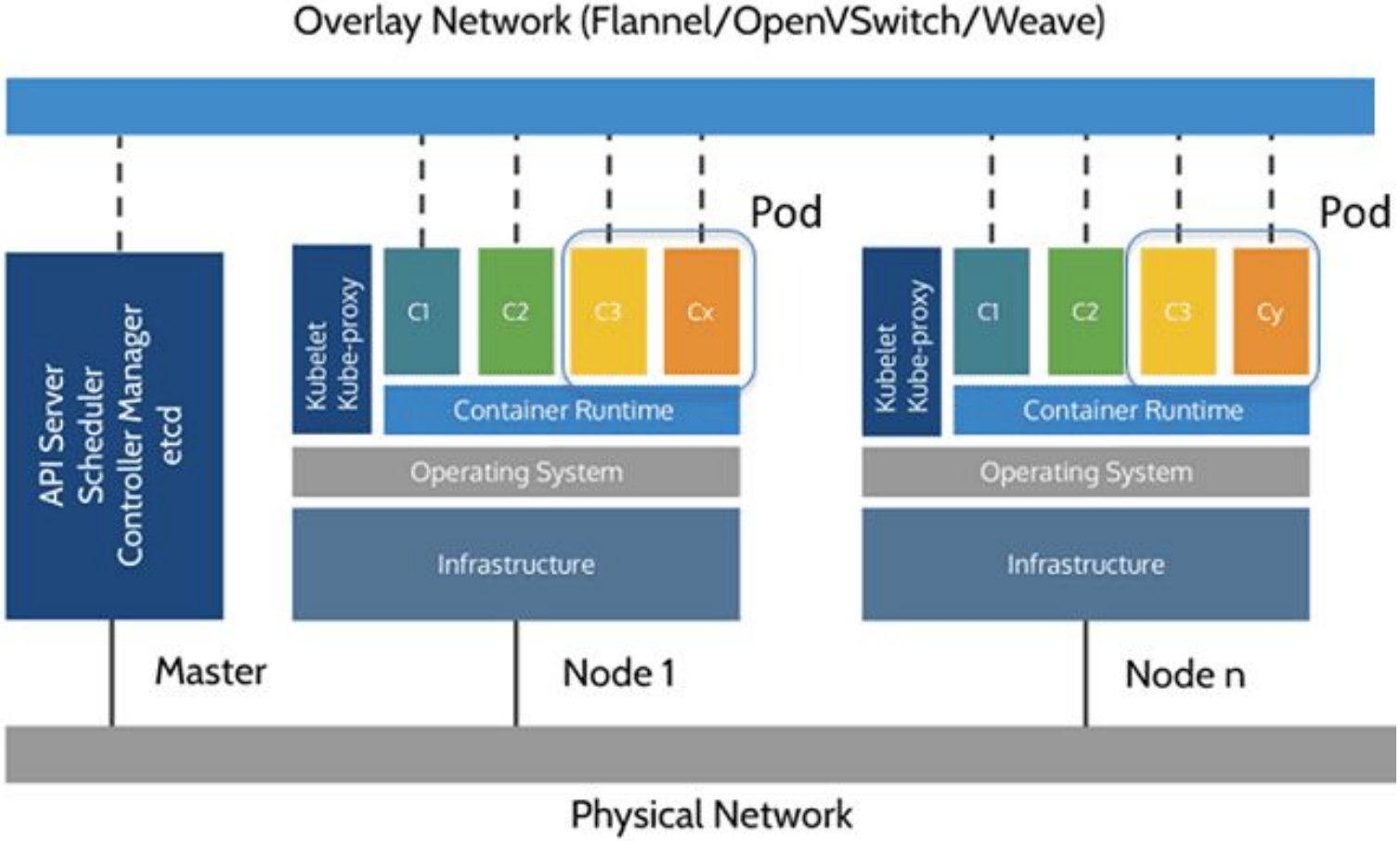
-Etcd - легковесная распределённая NoSQL-СУБД класса «ключ-значение», которая отвечает за согласованное хранение конфигурационных данных кластера.

-Сервер API-ключевой компонент подсистемы управления, предоставляющий интерфейс программирования приложений в стиле REST (в формате JSON поверх HTTP-протокола) и используемый для внешнего и внутреннего доступа к функциям Kubernetes. Сервер API обновляет состояние объектов, хранящихся в etcd, позволяя своим клиентам управлять распределением контейнеров и нагрузкой между узлами системы.

-Планировщик (scheduler), который регулирует распределение нагрузки по узлам, выбирая узел выполнения для конкретного пода в зависимости от доступности ресурсов узла и требований пода.

-Менеджер контроллеров (controller manager) – процесс, выполняющий основные контроллеры Kubernetes (DaemonSet Controller и Replication Controller), которые взаимодействуют с сервером API, создавая, обновляя и удаляя управляемые ими ресурсы (поды, точки входа в сервисы и т.д.).

Kubernetes Architecture



Архитектура Kubernetes

В Kubernetes контейнер – это программный компонент самого низкого уровня абстракции. Для межпроцессного взаимодействия нескольких контейнеров они инкапсулируются в поды.

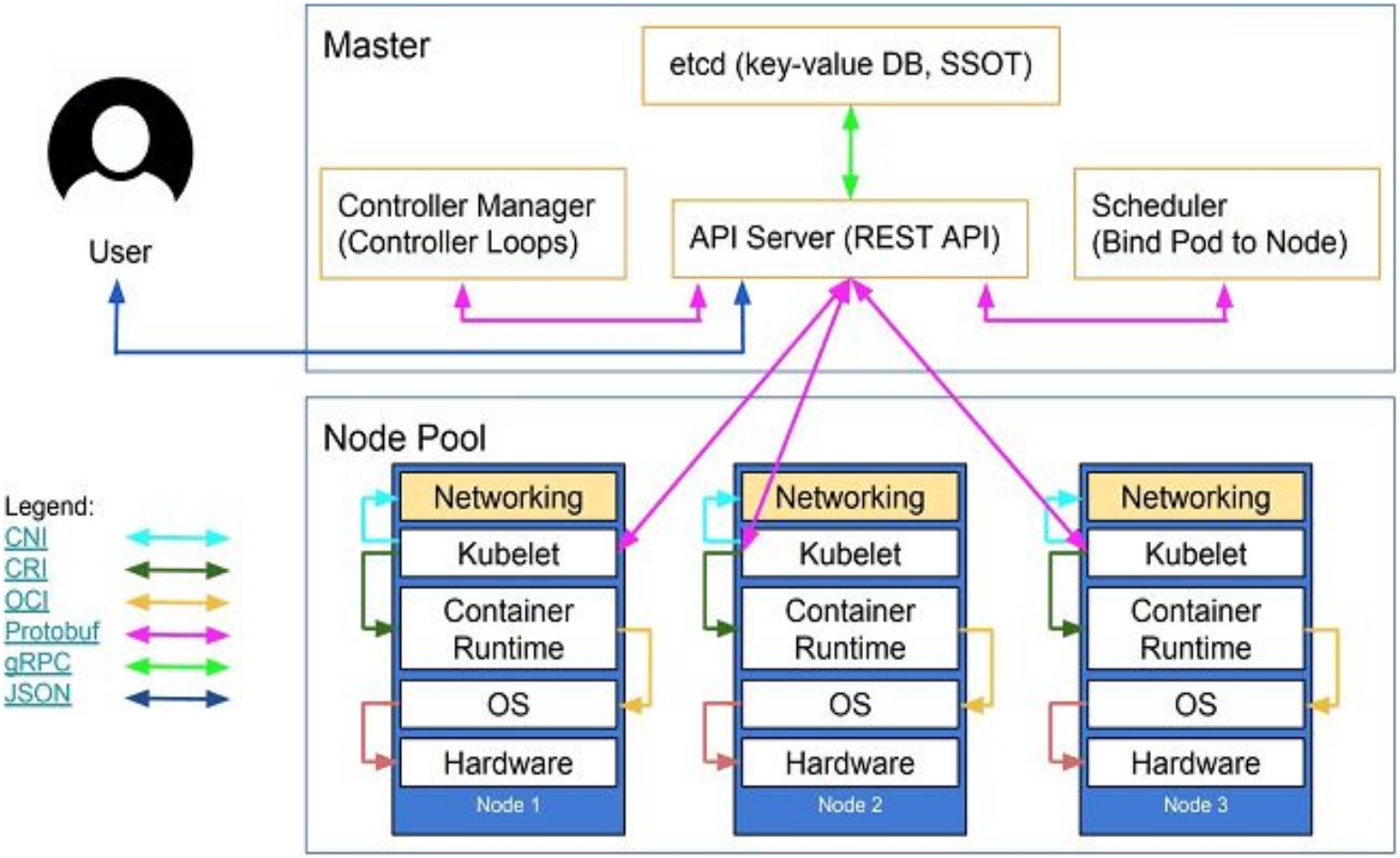
Задачей Kubernetes является динамическое распределение ресурсов узла между подами, которые на нем выполняются. Для этого на каждом узле с помощью встроенного агента внутреннего мониторинга Kubernetes cAdvisor ведется непрерывный сбор данных о производительности и использовании ресурсов (время работы центрального процессора, оперативной памяти, нагрузок на файловую и сетевую системы).

Что особенно важно для Big Data проектов, Kubelet – компонент Kubernetes, работающий на узлах, автоматически обеспечивает запуск, остановку и управление контейнерами приложений, организованными в поды.

При обнаружении проблем с каким-то подом Kubelet пытается повторно развернуть и перезапустить его на узле.

Аналогично HDFS, наиболее популярной распределенной файловой системе для Big Data-решений, реализованной в Apache Hadoop, в кластере Kubernetes каждый узел постоянно отправляет на master диагностические сообщения (*heartbeat message*).

Если по содержанию этих сообщений или в случае их отсутствия мастер обнаруживает сбой какого-либо узла, процесс подсистемы управления Replication Controller пытается перезапустить необходимые поды на другом узле, работающем корректно.



Принципы работы Kubernetes

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ КУБЕРНЕТИС

Поскольку **K8s** предназначен для управления множеством контейнеризированных микросервисов, неудивительно, что эта технология приносит максимальную выгоду именно в Big Data проектах.

Например, **Кубернетис** используют популярный сервис знакомств Tinder, телекоммуникационная компания Huawei, крупнейший в мире онлайн-сервисом поиска автомобильных попутчиков BlaBlaCar, Европейский Центр ядерных исследований (CERN) и множество других компаний, работающих с большими данными и нуждающимися в инструментах быстрого и отказоустойчивого развертывания приложений.

В связи с цифровизацией предприятий и распространением DevOps-подхода, спрос на владение Kubernetes растет и в отечественных компаниях.

Как показал обзор вакансий с рекрутинговой площадки HeadHunter, в 2019 году для современного DevOps-инженера и Big Data разработчика данная технология является практически обязательной.

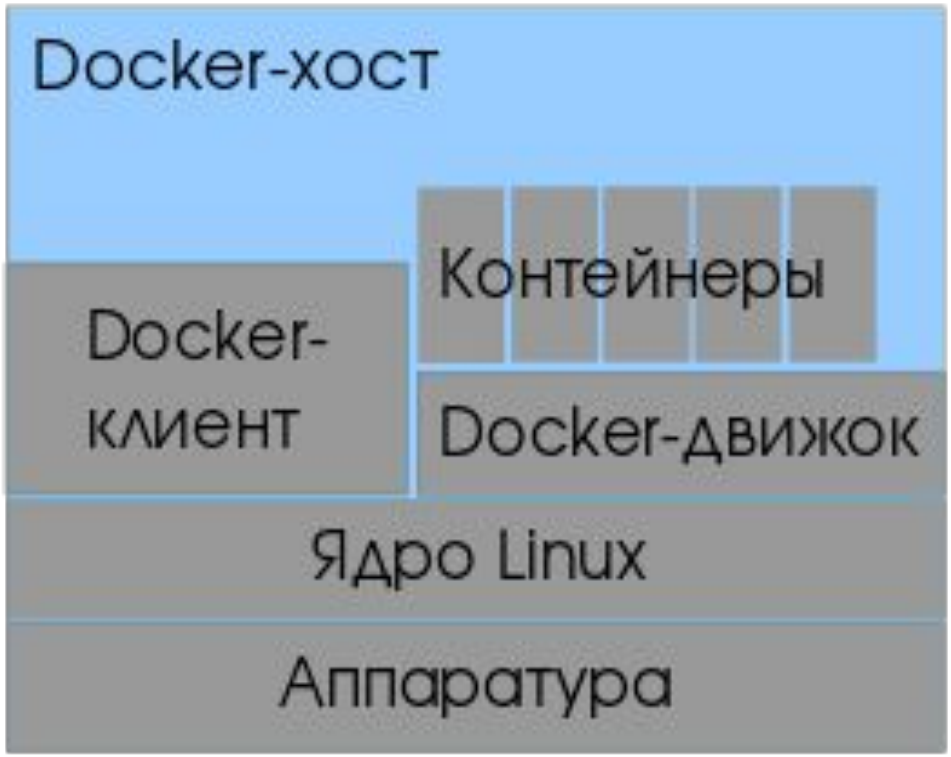
Kubernetes – настоящий must have для современного DevOps-инженера.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут на любой Linux-системе с поддержкой cgroups в ядре, а также предоставляет набор команд для управления этими контейнерами.

С появлением Open Container Initiative начался переход от монолитной к модульной архитектуре.

Разрабатывается и поддерживается одноимённой компанией-стартапом, распространяется в двух редакциях — общественной (*Community Edition*) по лицензии Apache 2.0 и для организаций (*Enterprise Edition*) по проприетарной лицензии.

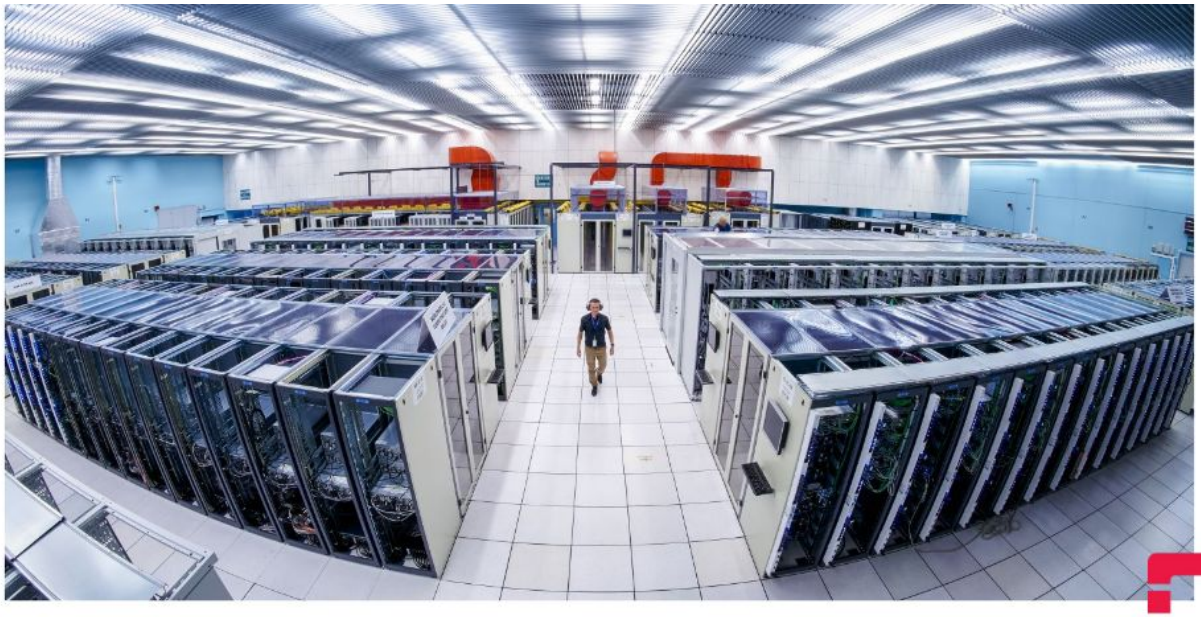
Написан на языке Go.



Docker на физическом Linux-сервере

Программное обеспечение функционирует в среде Linux с ядром, поддерживающим контрольные группы и изоляцию пространств имён (*namespaces*); существуют сборки только для платформ x86-64 и ARM.

Начиная с версии 1.6 (апрель 2015 года) возможно использование в операционных системах семейства Windows.



320 000 **Cores** 4 300 **Projects** 250 **Petabytes**

3 300 **Users** 10 000 **Hypervisors** 210 **Kubernetes Clusters**

Литература:

1. Ньюмен С. Создание микросервисов. — СПб.: Питер, 2016. — 304 с.: ил. — (Серия «Бестселлеры O’Reilly»).

2. Ричардсон Крис Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019. — 544 с.: ил. — (Серия «Библиотека программиста»).