

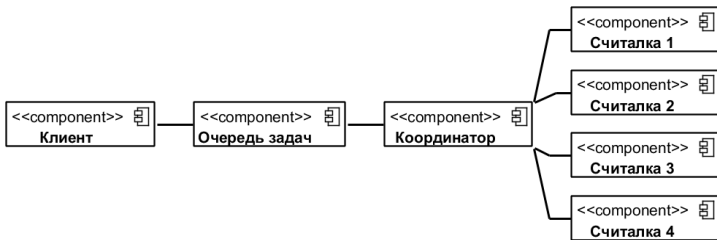
# Лекция 10: Проектирование распределённых приложений

Часть вторая: архитектурные вопросы

Юрий Литвинов  
y.litvinov@spbu.ru

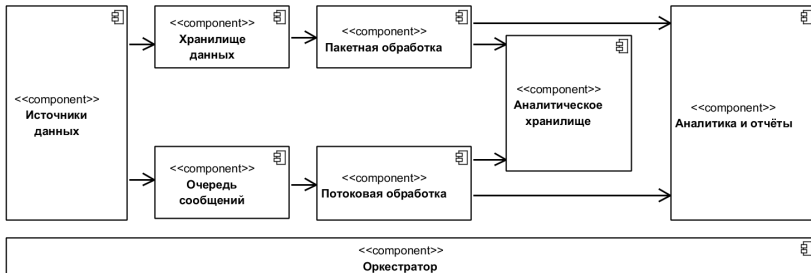
19.05.2026

# Big Compute



- ▶ Для сверхсложных задач, предполагающих тысячи вычислительных узлов
- ▶ Требуется «embarrassingly parallel» задачу
- ▶ Предполагает использование весьма продвинутых (и дорогих) облачных ресурсов

# Big Data

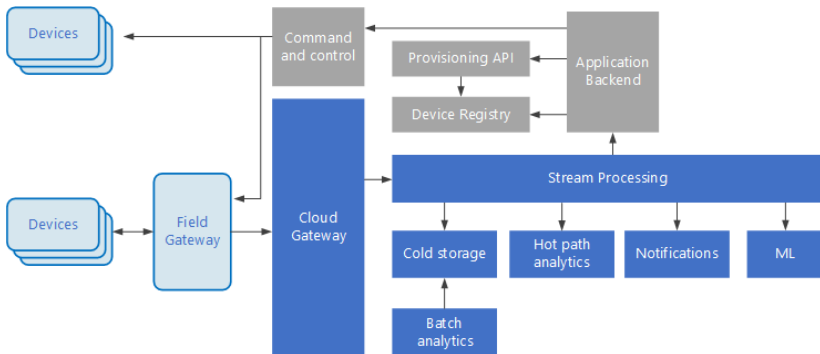


- ▶ Для аналитики над большими данными
  - ▶ Либо данных много и их можно обрабатывать неторопливо
  - ▶ Либо данных много и их надо обрабатывать в реальном времени
- ▶ Данные не лезут в обычную СУБД

## Big Data, хорошие практики

- ▶ Распределённые хранение и обработка
  - ▶ Например, Apache Hadoop, Apache Spark
- ▶ Schema-on-read
  - ▶ Data lake — распределённое хранилище слабоструктурированных данных
- ▶ Обработка на месте (TEL вместо ETL)
- ▶ Разделение данных по интервалам обработки
- ▶ Раннее удаление приватных данных

# Пример: IoT



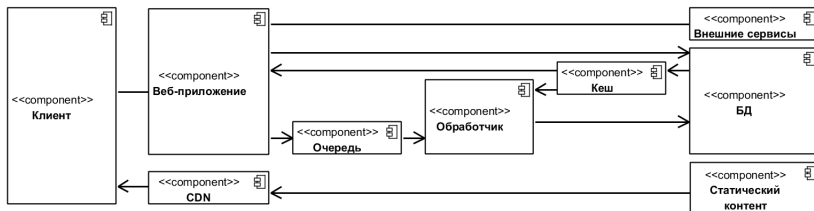
© <https://github.com/MicrosoftDocs/architecture-center/blob/main/docs/guide/architecture-styles/big-data.md>

# Событийно-ориентированная архитектура



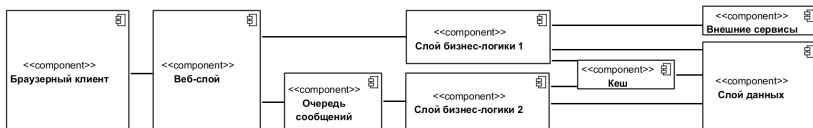
- ▶ Для обработки событий в реальном времени
- ▶ Бывает двух видов:
  - ▶ Издатель/подписчик (например, RabbitMQ)
  - ▶ Event Sourcing (например, Apache Kafka)

# Web-queue-worker



- ▶ Для вычислительно сложных задач в несложной предметной области
- ▶ Позволяет эффективно использовать готовые сервисы
- ▶ Независимое масштабирование фронтенда и обработчика
- ▶ Может превратиться в Big Ball of Mud

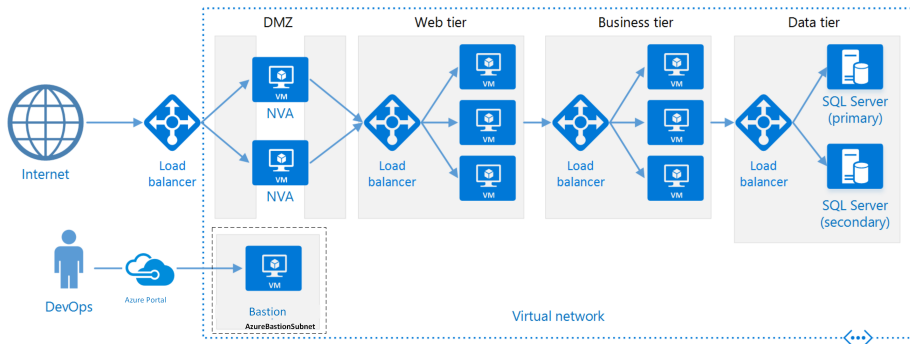
# N-звенная архитектура



- ▶ Для быстрого переноса монолита в облако
- ▶ Для простых веб-приложений
- ▶ Проблемы с масштабированием и сопровождаемостью

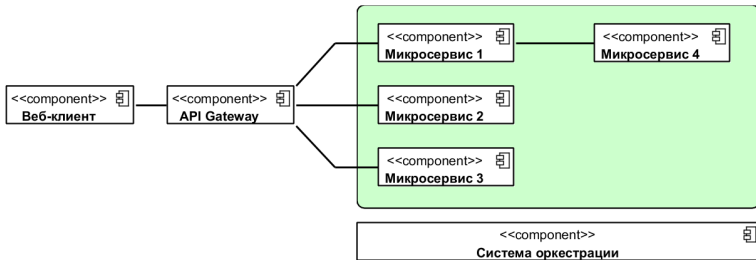


# Пример: N-звенное приложение на Azure



© <https://github.com/MicrosoftDocs/architecture-center/blob/main/docs/guide/architecture-styles/n-tier.md>

# Микросервисная архитектура



- ▶ Для приложений со сложной предметной областью
- ▶ Альтернатива монолиту, со своими достоинствами и недостатками
- ▶ Микросервис пишется одним человеком за две недели
  - ▶ На самом деле, пишется и поддерживается небольшой командой
- ▶ Микросервис — ограниченный контекст в смысле DDD

# Особенности

- ▶ Каждый микросервис — отдельное приложение
  - ▶ Независимость языков и технологий
  - ▶ Имеет своё хранилище данных, не имеет права шарить данные
    - ▶ В каком-то смысле, объект из ООП
    - ▶ Каждому сервису наиболее подходящая СУБД
- ▶ Мелкозернистая масштабируемость
- ▶ Независимое развёртывание
- ▶ Изоляция ошибок
- ▶ Маленькая и простая кодовая база

# Проблемы

- ▶ Сложность перекладывается с реализации на оркестрацию
  - ▶ Неочевидно, неразвитые инструменты
  - ▶ В целом сложнее, чем рассмотренные выше стили
  - ▶ Сложное управление и мониторинг, требуется развитая культура DevOps
  - ▶ Сложная в плане управления зависимостями разработка
- ▶ Технологический зоопарк
- ▶ Нагрузка на сеть
- ▶ Сложно поддерживать целостность данных
  - ▶ Eventual Consistency

# Дизайн REST-интерфейса

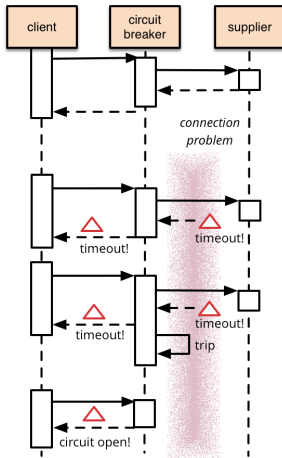
- ▶ API строится вокруг ресурсов, не действий
  - ▶ <http://api.example.com/customers/> — хорошо
  - ▶ [http://api.example.com/get\\_customer/](http://api.example.com/get_customer/) — плохо
- ▶ Отношения между сущностями:  
<http://api.example.com/customers/5/orders>
  - ▶ Максимум одно отношение — надо будет, сделают ещё запросы
- ▶ API — модель предметной области, не данных
- ▶ Семантика HTTP
  - ▶ Заголовки Content-Type, Accept
  - ▶ Коды возврата (200, 204, 404, 400, 409)
- ▶ Механизмы фильтрации и «пагинации»
- ▶ Поддержка Partial Content
- ▶ Hypertext as the Engine of Application State (HATEOAS)
- ▶ Версионирование — не ломать обратную совместимость

# Общие принципы дизайна распределённых приложений

## Самовосстановление

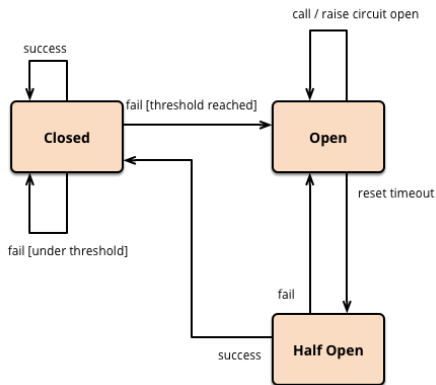
- ▶ Повтор при временном отказе
- ▶ API для самодиагностики
- ▶ Разделение на изолированные группы ресурсов
- ▶ Буферизация запросов
- ▶ Автоматическое переключение на резервный экземпляр, ручное обратно
- ▶ Промежуточное сохранение
- ▶ Плавная потеря работоспособности (graceful degradation)
- ▶ Тестирование отказов, Chaos engineering

# Circuit Breaker, поведение



© <https://martinfowler.com/bliki/CircuitBreaker.html>

# Circuit Breaker, состояния

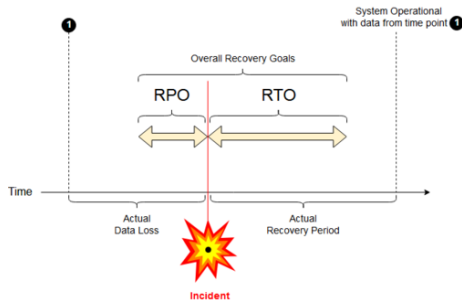


© <https://martinfowler.com/bliki/CircuitBreaker.html>



# Избыточность

- ▶ Бизнес-требования к надёжности
  - ▶ Recovery Time Objective, Recovery Point Objective, Maximum Tolerable Outage
- ▶ Балансировщики нагрузки
- ▶ Репликация БД
- ▶ Разделение по регионам
- ▶ Шардирование



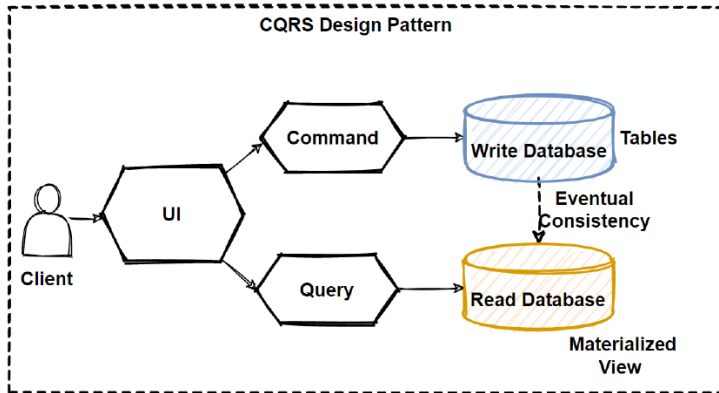
©

[https://en.wikipedia.org/wiki/Disaster\\_recovery](https://en.wikipedia.org/wiki/Disaster_recovery)

# Минимизация координации

- ▶ Доменные события (domain events)
- ▶ Event Sourcing
- ▶ Асинхронные, идемпотентные операции
- ▶ Шардирование
- ▶ Eventual Consistency, компенсационные транзакции

# Command and Query Responsibility Segregation



© <https://medium.com/design-microservices-architecture-with-patterns/cQRS-design-pattern-in-microservices-architectures-5d41e359768c>

# 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 ответы
  - ▶ Этот пункт в распределённых системах должен быть обеспечен всегда, потому что отказы неизбежны. Остаётся выбрать один из двух

# ACID vs 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

# Проектирование для обслуживания

- ▶ Делать всё наблюдаемым
  - ▶ Трассировка, в т.ч. распределённая
  - ▶ Логирование
- ▶ Мониторинг, метрики
- ▶ Стандартизация форматов логов и метрик
- ▶ Автоматизация задач обслуживания
- ▶ Конфигурация — это код