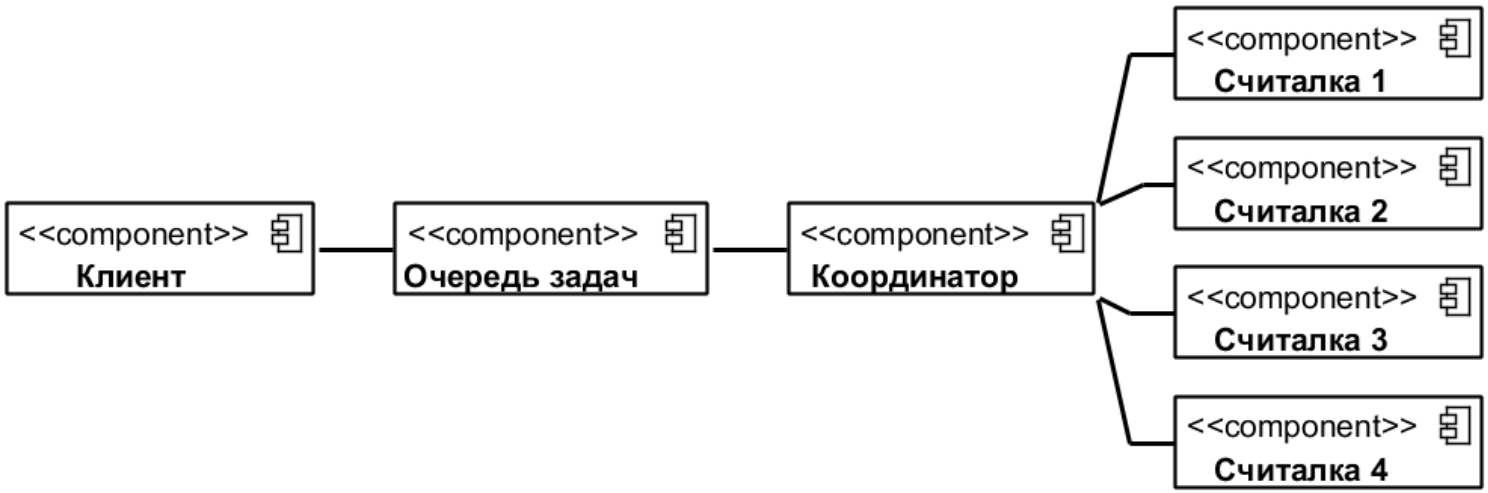
Лекция 14: Проектирование распределённых приложений (3)

Стратегические вопросы

Юрий Литвинов
yurii.litvinov@gmail.com

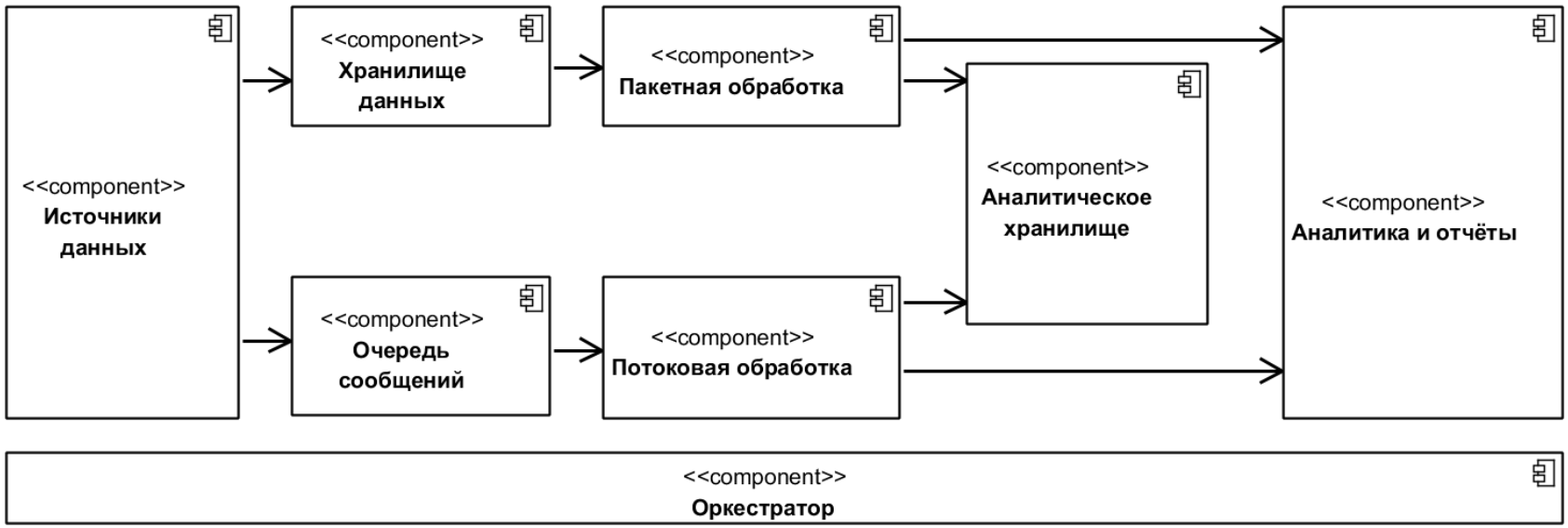
25.04.2022

Big Compute



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

Big Data

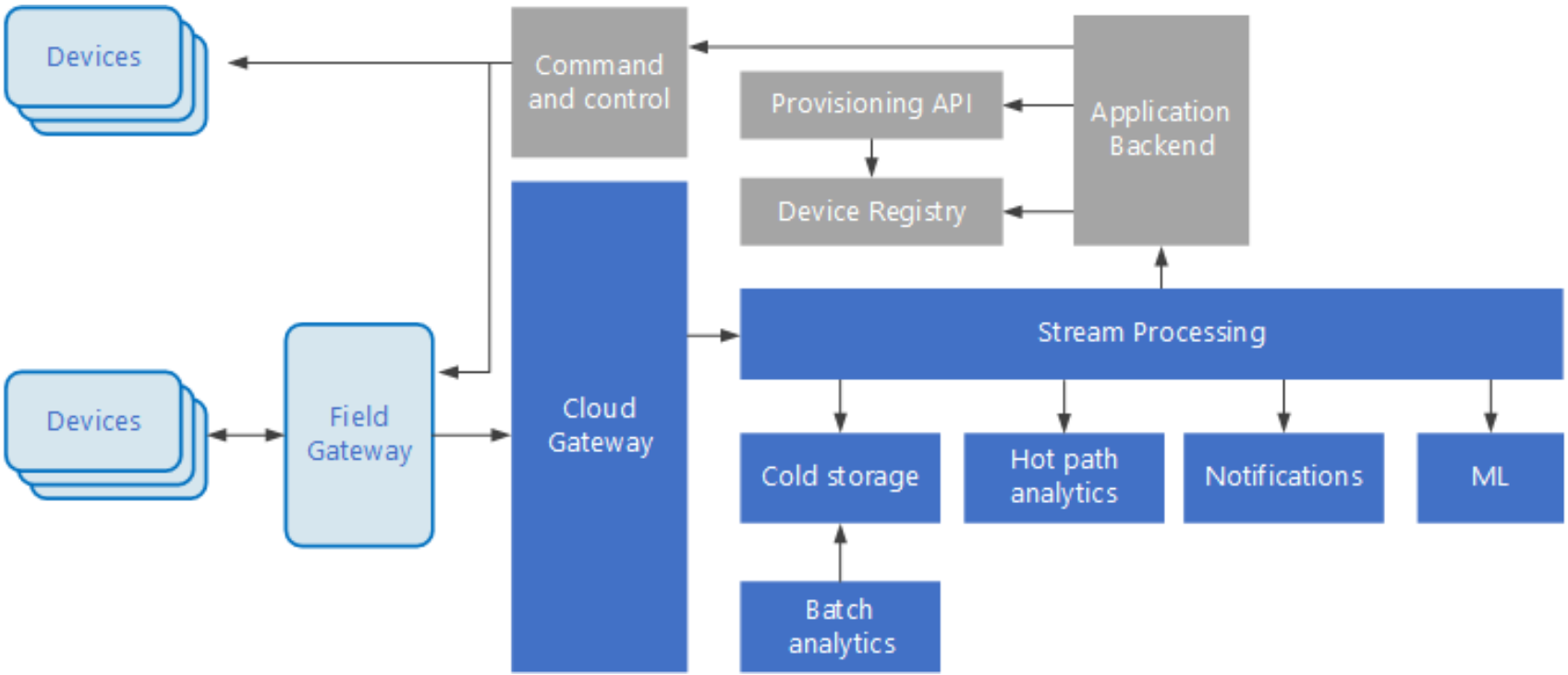


- Для аналитики над большими данными
  - Либо данных много и их можно обрабатывать неторопливо
  - Либо данных много и их надо обрабатывать в реальном времени
- Данные не лезут в обычную СУБД

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

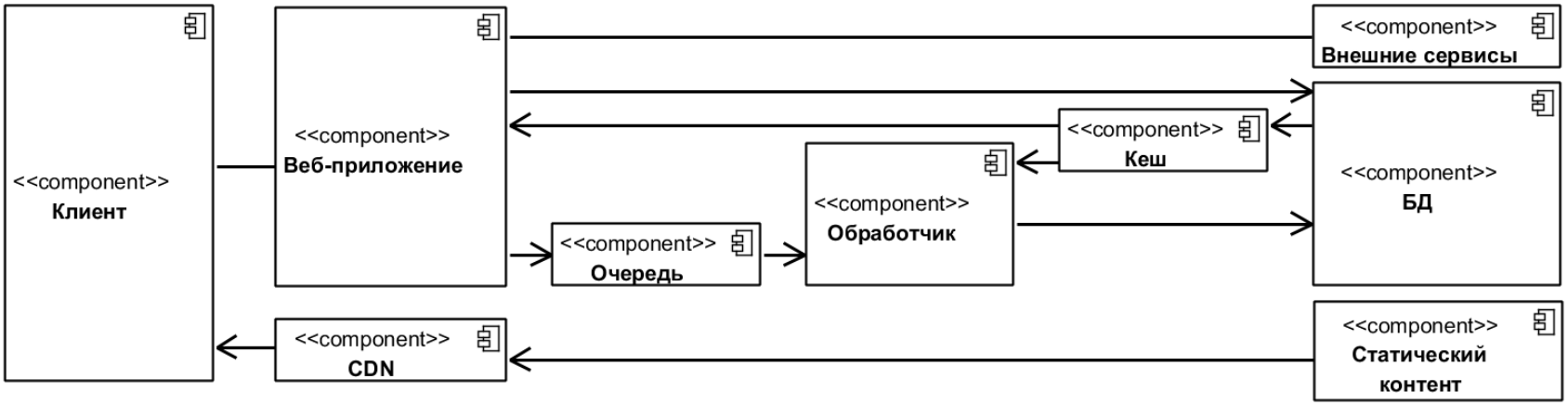


© https://github.com/MicrosoftDocs/architecture-center/blob/main/docs/guide/architecture-styles/big-data.md

Событийно-ориентированная архитектура

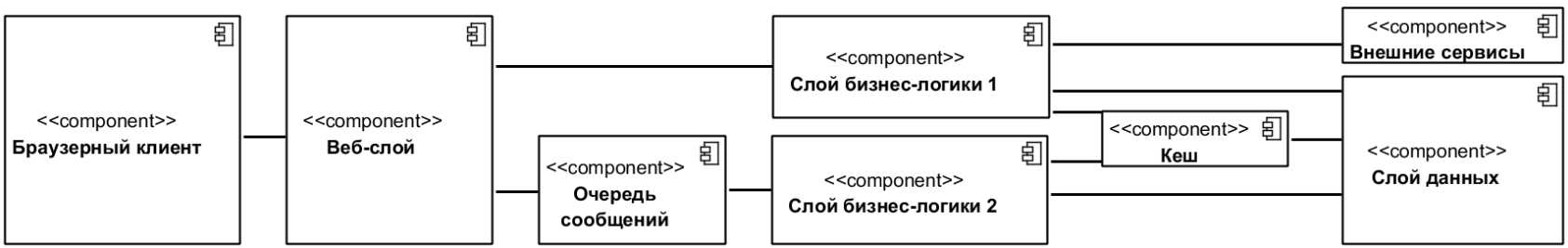
- Для обработки событий в реальном времени
- Бывает двух видов:
  - Издатель/подписчик (например, RabbitMQ)
  - Event Sourcing (например, Apache Kafka)

Web-queue-worker



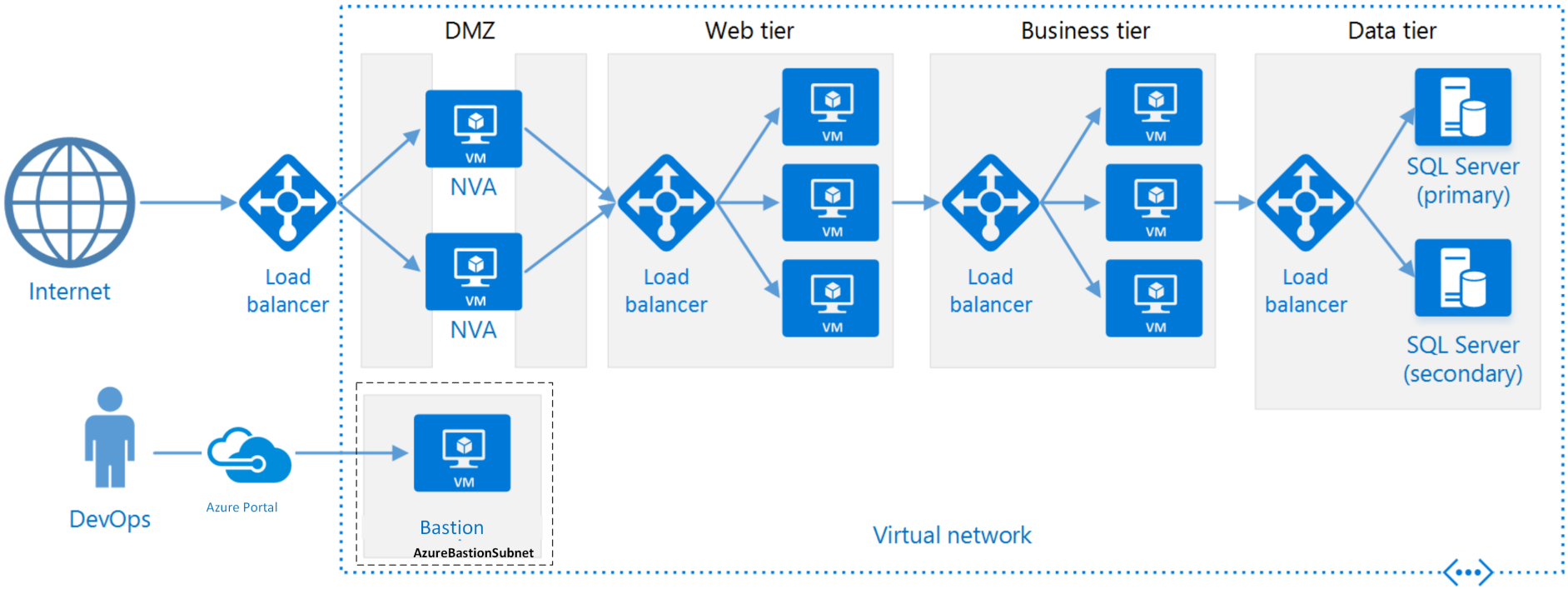
- Для вычислительно сложных задач в несложной предметной области
- Позволяет эффективно использовать готовые сервисы
- Независимое масштабирование фронтенда и обработчика
- Может превратиться в Big Ball of Mud

N-звенная архитектура



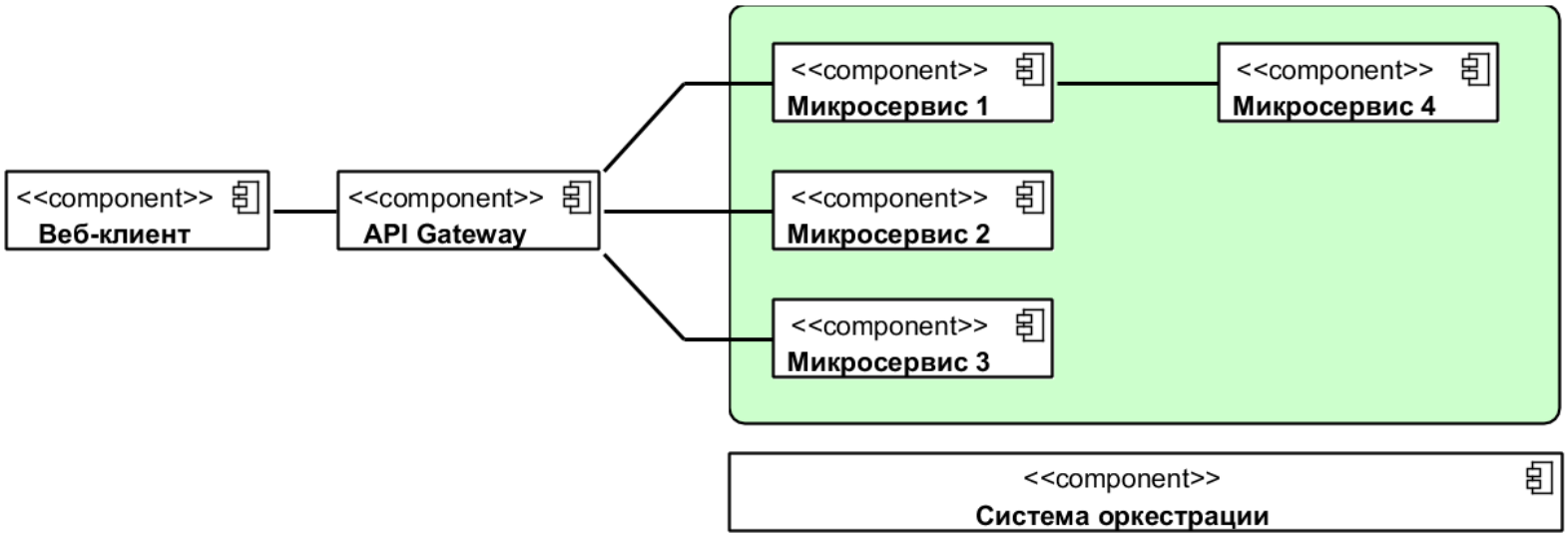
- Для быстрого переноса монолита в облако
- Для простых веб-приложений
- Проблемы с масштабированием и сопровождаемостью

Пример: N-звенное приложение на Azure



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

Микросервисная архитектура

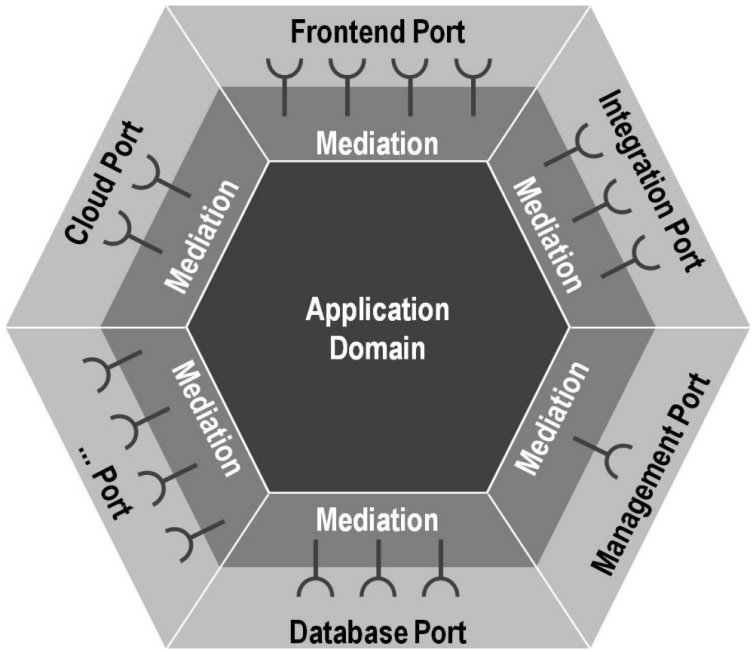


- Для приложений со сложной предметной областью
- Альтернатива монолиту, со своими достоинствами и недостатками
- Микросервис пишется одним человеком за две недели
  - На самом деле, пишется и поддерживается небольшой командой
- Микросервис — ограниченный контекст в смысле DDD

Гексагональная архитектура

“Порты и адаптеры”

- Другая точка зрения на уровни: самый нижний — уровень предметной области
- Всё остальное поставляется ему как внешние зависимости
- Активно используется Dependency Inversion
- Порт — по сути, интерфейс, предоставляемый или потребляемый
- Адаптер — паттерн “Адаптер” для “подгонки” интерфейсов



© B Butzin et al, Microservices Approach for the Internet of Things

Плюсы и минусы

Плюсы:

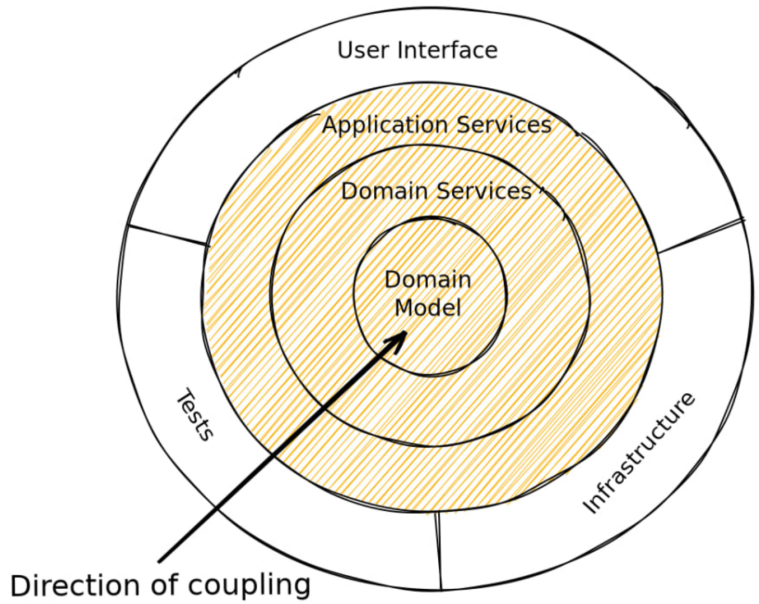
- Изоляция механизмов доставки
- Изоляция вспомогательных механизмов
- Лёгкость тестирования, моки
- Чистая бизнес-логика и модель предметной области
  - Максимальная простота
  - Возможность валидации и конвертирования данных

Минусы:

- Довольно тяжеловесна
- Непонятно, что делать с фреймворками
- Не очень подробна

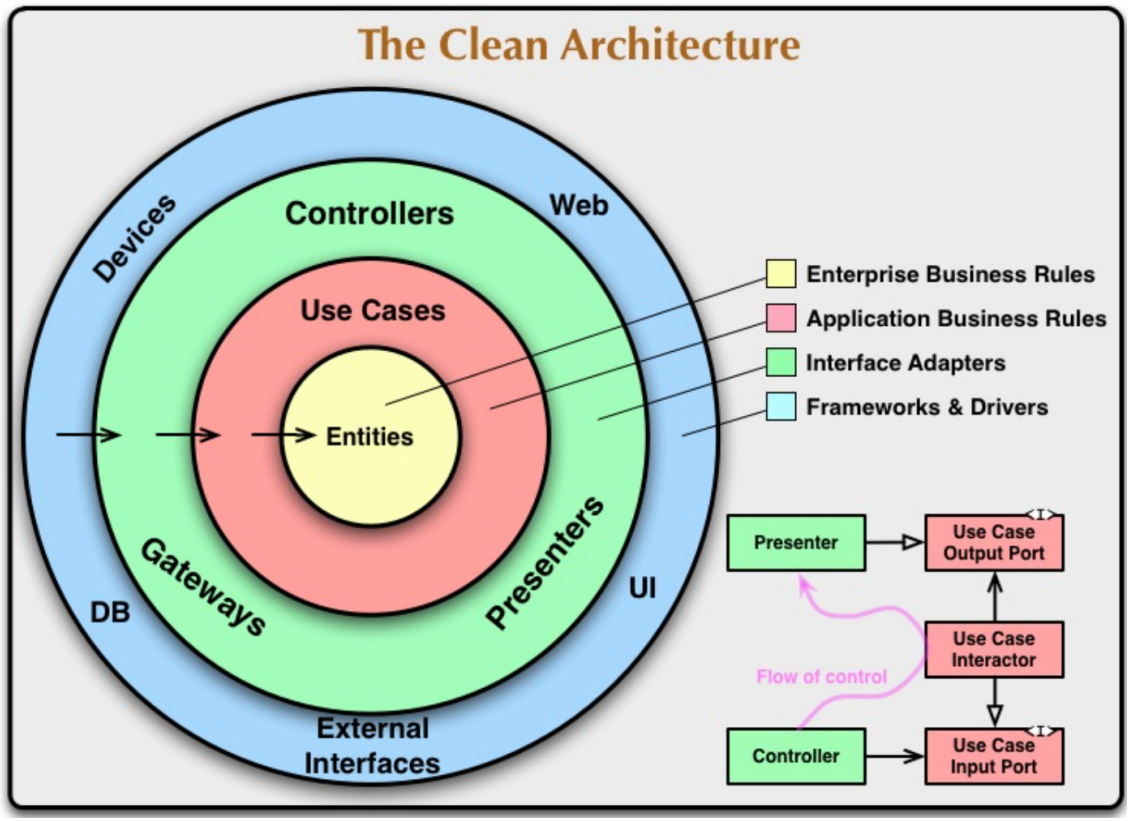
Луковая архитектура

- Дальнейшее развитие гексагональной — определяет внутреннюю структуру ядра
- Внутренние слои не знают о внешних, доменная модель вообще ни о ком не знает
- Внутренние слои определяют интерфейсы, внешние их реализуют
- Уровневость нестрогая — слой может использовать все слои под ним



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

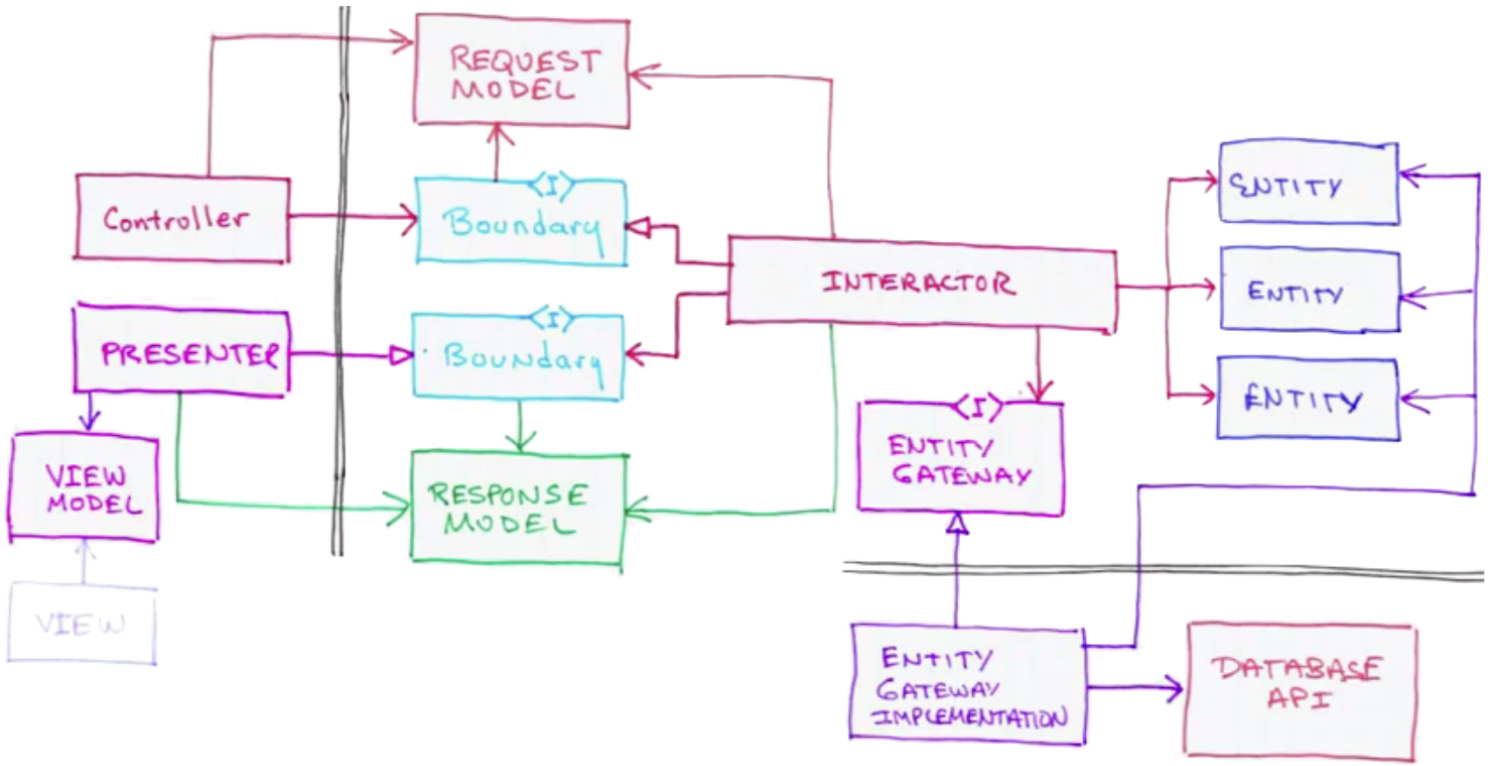
Чистая архитектура



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

- Дальнейшее развитие луковой — определяет поток управления

Чистая архитектура, обработка запроса



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

Дизайн REST-интерфейса

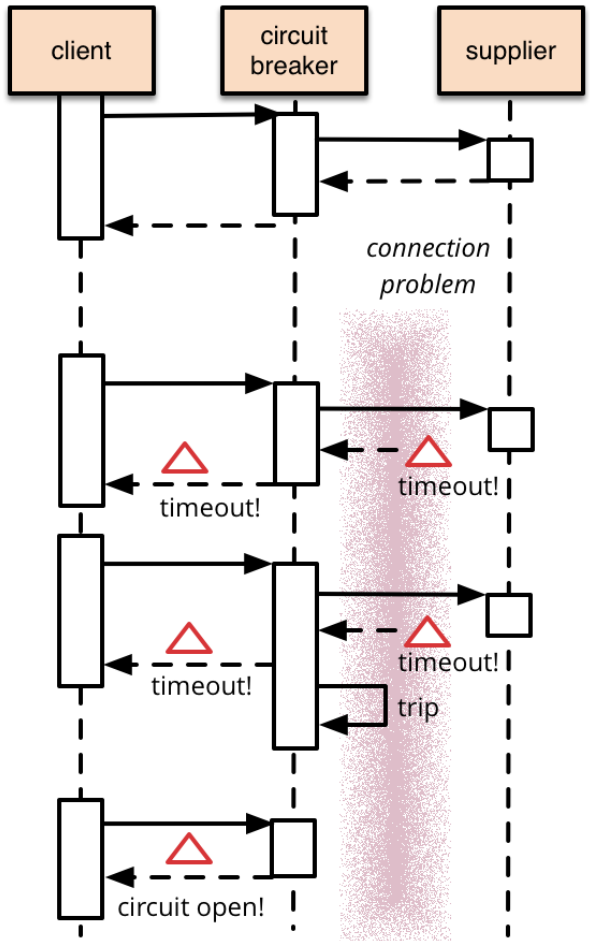
- API строится вокруг ресурсов, не действий
  - http://api.example.com/customers/ — хорошо
  - http://api.example.com/get_customer/ — плохо
- Отношения между сущностями: http://api.example.com/customers/5/orders
  - Максимум одно отношение — надо будет, сделают ещё запросы
- API — модель предметной области, 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

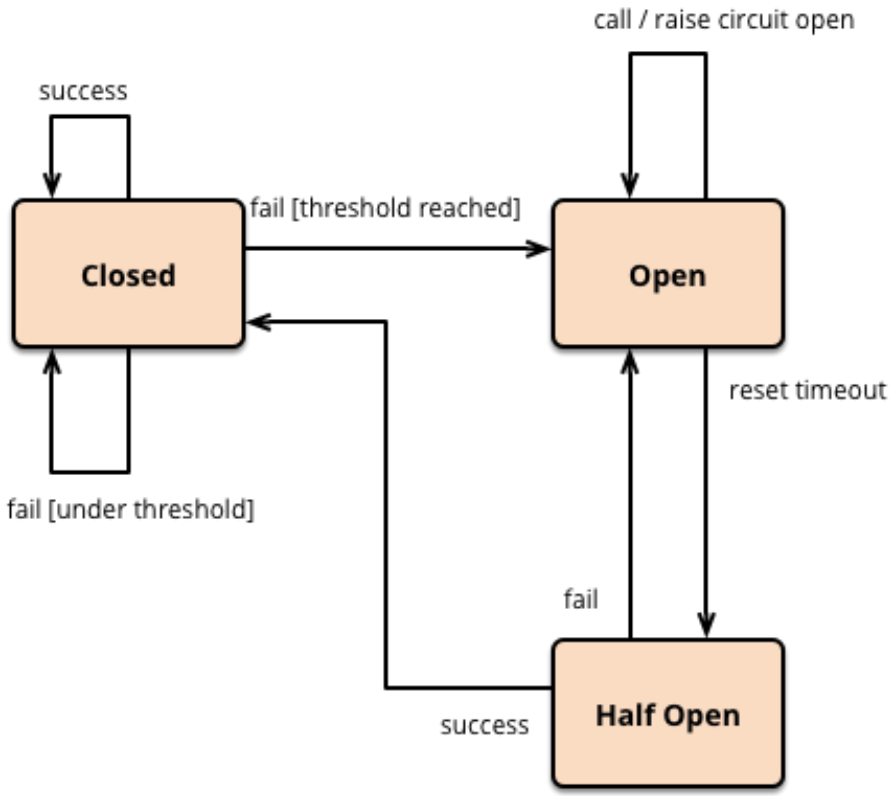
- Повтор при временном отказе
- API для самодиагностики
- Разделение на изолированные группы ресурсов
- Буферизация запросов
- Автоматическое переключение на резервный экземпляр, ручное обратно
- Промежуточное сохранение
- Плавная потеря работоспособности (graceful degradation)
- Тестирование отказов,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

Circuit Breaker, поведение



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

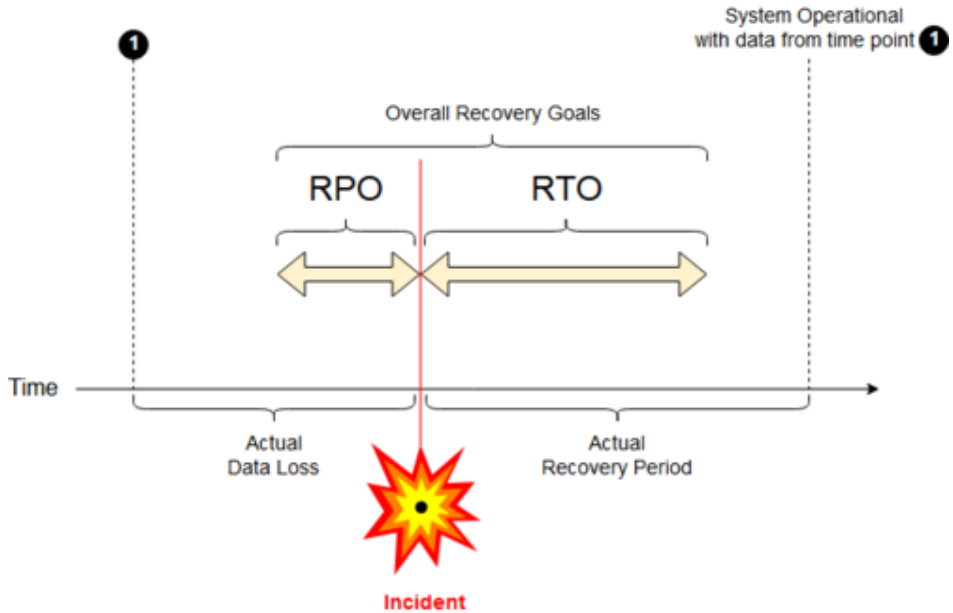
Circuit Breaker, состояния



© 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



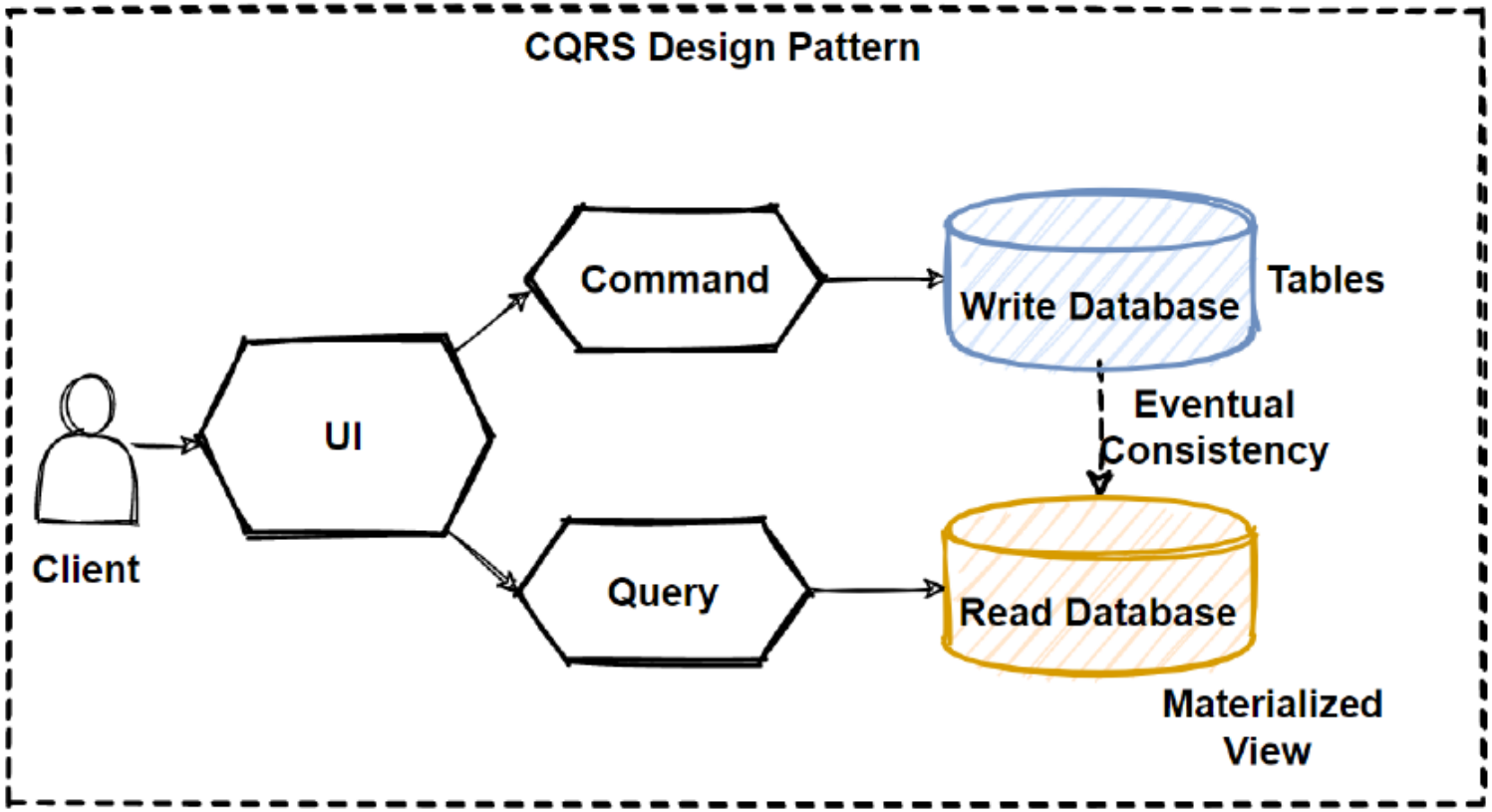
©

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

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

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