Патерны проектирования
https://habr.com/ru/companies/ruvds/articles/699648/#anchorid

Многоуровневая архитектура
N-layer/N-tier архитектура

https://youtu.be/WYh5khUsv_o?si=Gzp4Mz4-Cxyo5woo

N-Layer — N слойная архитектура — описывает логическую структуру компонентов(код, разделенный на слои)

N-tier — N уровневая архитектура — описывает физические компоненты системы (процессы)

Примеры:

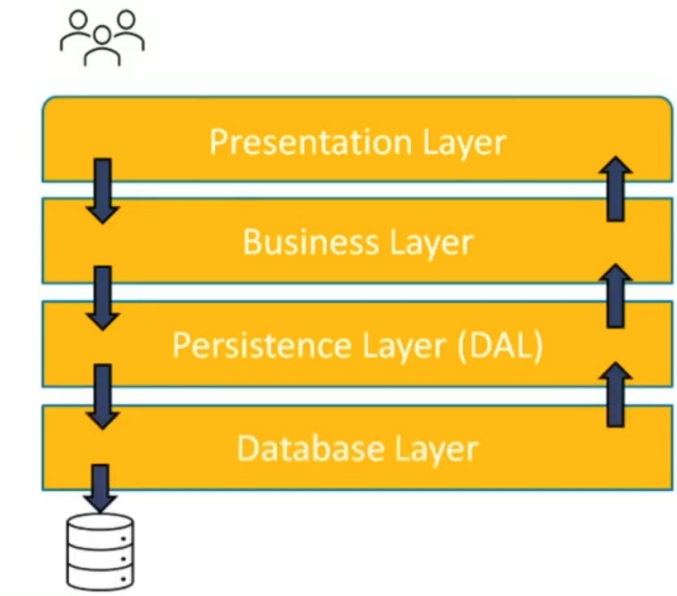
В N — tier — работает с физическими компонентами(WEB-сервера/браузеры — комп пользователя/БД-сервера)

В N-layer — это уже логическое разбиение — говорим о коде …

1. **Presentation Layer —** содержит код пользователя интерфейса — UI + взаимодействие с WEB-сервером
2. **Business Layer — бизнес-логика** вычисление обработки данных, согласно бизнес правилами организации. (У калькулятора — это вычисления)
3. **Persistence Layer —** слой доступа к данным — принимает запросы от бизнес логики и формирует запрос к БД, потом получает данные от БД и отправляет Бизнес логике

ФОРМИРУЕТ SQL запросы или взаимодействет через ORM

4. **Database Layer** — сам БД, драйверы базы данных



Здесь поддерживается принцип — ***разделение ответственности*** *— где каждый слой работает в рамках своей ответственности*

Слои должны быть закрытыми, т. е. должны последовательно переходить от верхнего слоя к нижнему(перепрыгивает через слои запрещается)

Преимущество в том, что при модифицировании приложения чаще всего надо будет менять только один слои(например БД было ORACLE стало POSTGRESQL тогда надо передалать только PERSISTENCE LAYER будет)
20% - запросов могут попадать в воронку — это принцип при котором один из слоев ниченр не делает → Если больше 20%, то разрешаем перепрыгнуть через один из слоев

Каналы и фильтры

https://learn.microsoft.com/ru-ru/azure/architecture/patterns/pipes-and-filters

Разбиение сложной задачи на **независимые этапы(**фильтры**),** которые соединены через каналы.

Каждый этап(фильтр) — обрабатывает данные и передает их дальше
Канал — пути передачи данных

Примеры:

Пайплайн 1: [Загрузить] → [Обрезать] → [Черно-белое] → [Сохранить]
Пайплайн 2: [Проверка] → [Расчет] → [Скидка] → [Отправка]

Фильтры могут работать параллельно:

```
                → [Фильтр A] →
[Данные]  → [Фильтр B] → [Объединить] → [Результат]
                → [Фильтр C] →
```

Итог: Паттерн «Каналы и фильтры» помогает разбить монолитную задачу на простые шаги, которые легко тестировать, масштабировать и переиспользовать. Главное — правильно спроектировать порядок фильтров и каналы передачи данных!

Клиент Серверное взаимодействие

https://youtu.be/JlTQujO6cbA?si=ZwDOApVr-VVncJM_

Логика работы делиться на 2 независимых уровня:

Толстый клиент — вся работа по обработки данных и отображению на клиенте, а серевер — это просто хранилище данных
Тонкий клиент — вся работа по обработки информации лежит на сервере

Клиент — пользователи работают на своем ПК (ПК - client)
Сервер - ПК клиента обращается к серверу (который отвечает 100% за хранение данных, но не факт, что обрабатывают)

Вся цепочка работает от сервера…

MVC

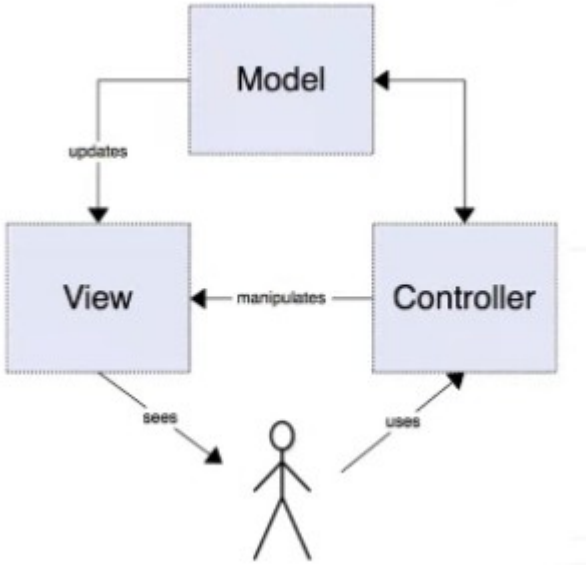
https://skillbox.ru/media/code/chto_takoe_mvc_bazovye_kontseptsii_i_primer_prilozheniya/

Логика приложения делится на 3 части:

•**Controller (контроллер)**. Обрабатывает действия пользователя, проверяет полученные данные и передаёт их модели. (отчечает за проверку корректности полученных данных)

•**Model (модель)**. Получает данные от контроллера, выполняет необходимые операции и передаёт их в вид.(основные функции обработчики)

•**View (вид или представление)**. Получает данные от модели и выводит их для пользователя.



В **приложении:**
Вид — интерфейс
Контроллер - обработчик событий (смотрит, что сделал пользователь)
Модель — метод, вызываемый обработчиком

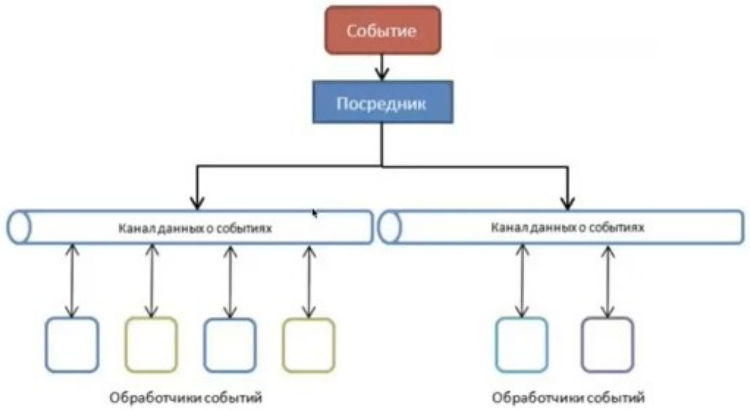
MVC может быть разным, например view и модель могут общаться между собой через controller

При этом паттерне программисты могут работать параллельно, каждый пилит свою часть

EVENT-DRIVEN модель

ТО что нам рассказывали:

- Компоненты должны быть слабо связаны, которые не понимают что происходит, они просто генерируют события



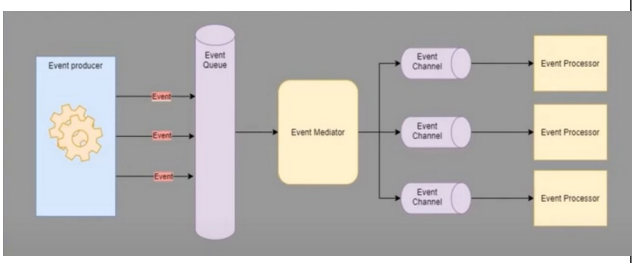
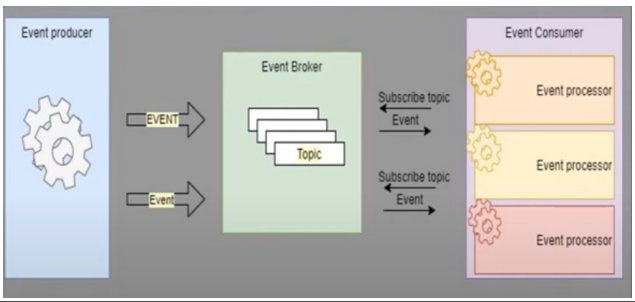
События публикуются на шину данных
Сервисы, слушают эту шину данных.
И когда сервис видит определенное событие, которое для него. Он берет это событие
На основе события начинает, что-то делать

Благодаря этому сервисы могут работать параллельно (+)
Все завязано на шине данных (-)
Такие системы сложно разрабатывать и квалификация специалистов очень важна становистя (-)

Дополнительный материал

событийно ориентированный патрен
https://youtu.be/Wh_wHJH79Fs?si=_grm5xbJPQ7P5Ipa
https://tproger.ru/articles/event-driven-arhitektura--principy-i-primery-ispolzovaniya
- асинхронная обработка событий

Event — событие — триггер для запуска определенный логики
Broker и Mediator — приложении посредники между event и приложенирем принимающий event (это 2 типа топологии)
Event Log — централизованное хранилище данных (события сохраняются сюда)

Топология Mediator / посредник	Топология Broker/Продюсер
 **Основные компоненты топологии:**	Похожа на Mediator, но тут нет **центрального распределителя событий** 

Event Queue **Event Mediator —** отвечает за организацию шагов из начального события **Event Channel / каналы-** нужны для распараллеливания (походу) **Event Processor —** содержат бизнес логику приложения (обработчики)	

Антипатерны:

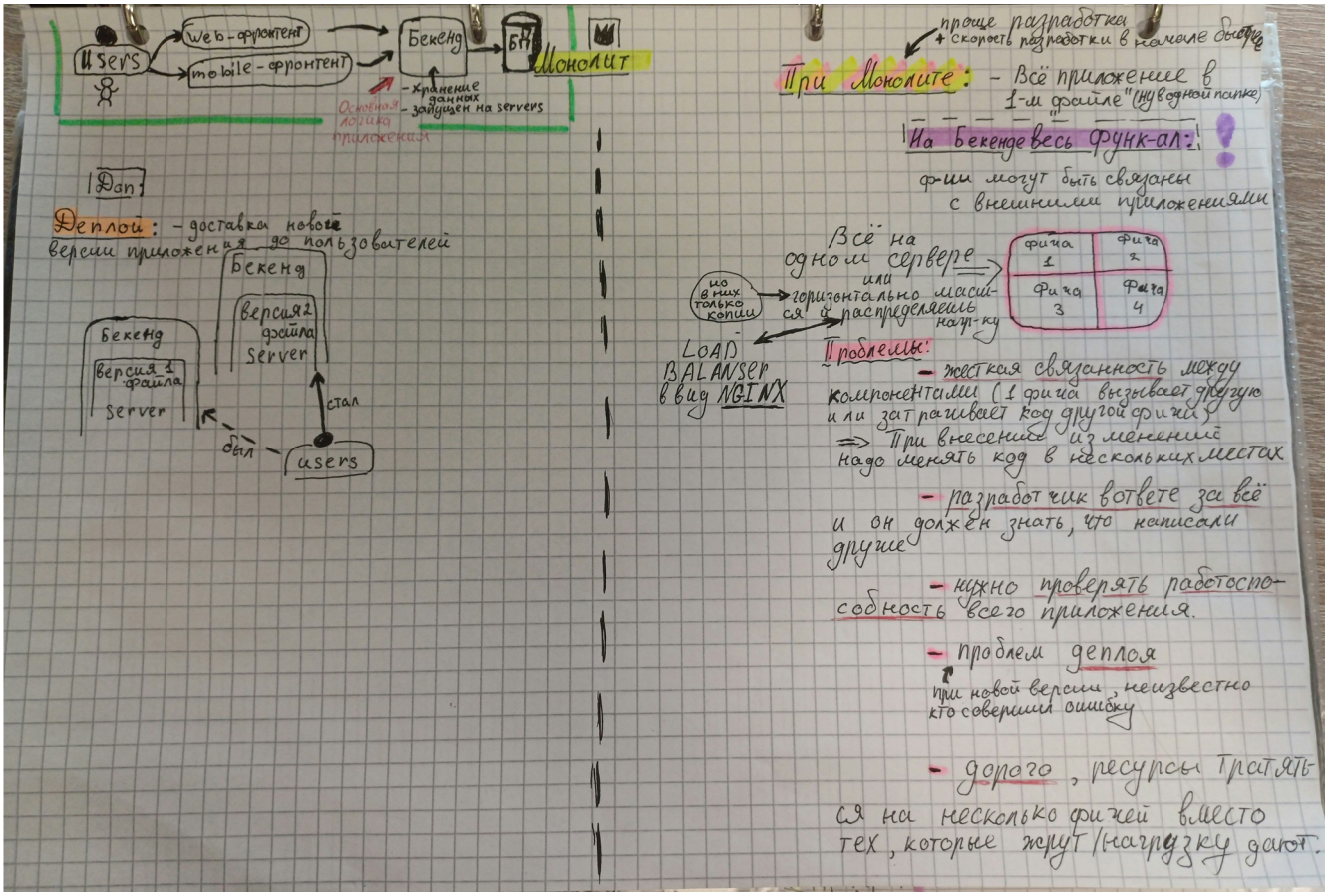
- **Events** должны быть атомарны (у них не должны быть зависимости от других event)
- Асинхронный обмен сообщениями — нет порядка обработки событий
- События должны быть автономны — Broker не зависит от того, как обрабатываются события на потребителе
- и т.д

Компоненты:
Produccer/Производители — отправляет сообщение, после какого-то действия в системе
Event — само сообщение / пакет данных (о изменении системы, что что-то произошло)
Каналы — по ним доставляются события до потребителей
Маршрутизатор — добавляет ID к событиям. Ни как не обрабатывает событие, а просто отправляет план действий потребителю

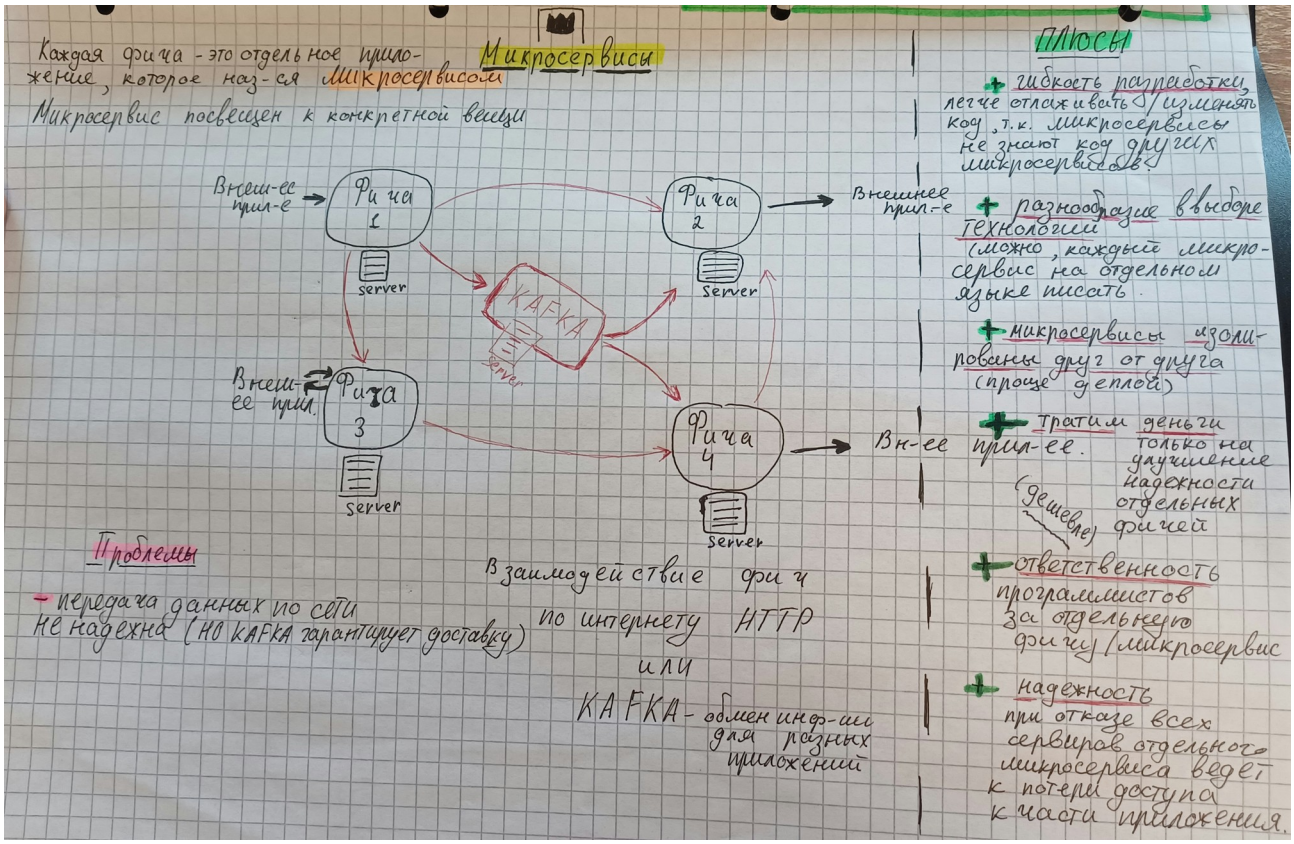
Брокер — каналы + маршрутизатор, запускает подтверждение о том, что событие произошло и генерит ответ

Потребитель — ответ на сообещение (или какая-то ответная реакция в самой системе)

Монолит



Микросервисы



SOLID

https://youtu.be/ihBN8dQ7XAM?si=--YwvNb4uCk_vYDK

S — принцип единной ответственности — 1 класс решает одну задачу

Плохой пример:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print(self.name, "makes sound...")

    @classmethod
    def get_from_db(cls, name):
        print("find", name, "in db", cls)

    def save_to_db(self):
        print("save to db", self)
```

Хороший пример:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print(self.name, "makes sound...")

class AnimalDB:

    @classmethod
    def get_from_db(cls, name):
        print("find", name, "in db", cls)

    @classmethod
    def save_to_db(self, animal):
        print("save to db", animal)
```

O — принцип открытости и закрытости — классы/функции и т. д. - должны быть открыты для расширения, но закрыты для модификации

т. е. нельзя изменять уже существующии компоненты, но можно с помощью наследования добавлять новый функционал

```python
class Animal:
    def __init__(self, name, sound):
        self.name = name
        self.sound = sound

    def make_sound(self):
        print(self.name,
              "makes sound", self.sound)


lion = Animal("lion", "roar")
cat = Animal("cat", "meow")
print(lion.sound)
# roar
lion.make_sound()
# lion makes sound roar
print(cat.sound)
# meow
cat.make_sound()
# cat makes sound meow
```

L — принцип подстановки Барбары Лисков — необходимо, чтобы подклассы могли заменять свои суперклассы

```python
class Animal:
    def get_tail_info(self):
        return ""


class Lion(Animal):
    def get_tail_info(self):
        return "Lion has a big and heavy tail..."


class Cat(Animal):
    def get_tail_info(self):
        return "Cat has a nice and furry tail..."


def get_animal_tail_info(animal: Animal):
    return animal.get_tail_info()
```

* - не используй isinstance — в основном это про это

I — принцип разделения интерфейса — нужен для того, чтобы интерфейсы были проще

Интерфейс — должен решать одну задачу

D — принцип инверсии зависимостей — ЧТО-то конкретное должно зависить от абстрактного, а не наоборот

ЧИСТАЯ АРХИТЕКТУРА

- код который легко модифицировать

- затрагивает и другие патерны MVC

Есть пользовательский интерфейс — UI
Есть репозитории
и есть сущности с которыми мы работаем — Users Cases и что-то там еще написано..

1. **Уровень представления** - где появляется сетевые протоколы …
2. **Уровень приложения** — контролеры в MVC — координация по монолитам
3. **Уровень домена —** это предметная область(что-то конкретное, то с чем мы работаем)
4. **Уровень инфраструктур —** БД + API — сервера всякие

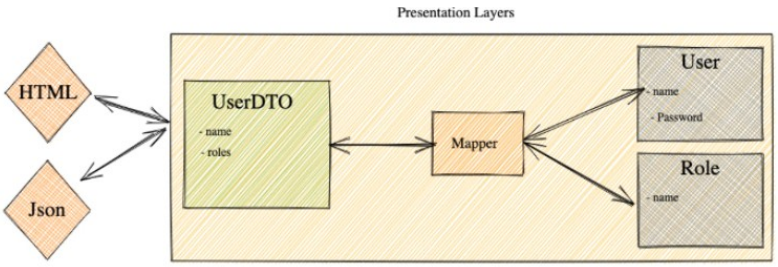
DTO

- это объект в БД, отображение сущности в БД на наши объекты

DTO

Шаблон проектирования, который используется для передачи данных между слоями приложения.

DTO представляет собой объект, который содержит данные, необходимые для выполнения операции или запроса в приложении.



Паттерн DTO полезен в системах с удаленными вызовами между компонентами, так как позволяет стандартизировать параметры обрабатываемых сущностей от компонента к компоненту.

Паттерн DTO позволяет иметь несколько вариантов Кастомизацию объекта, таким образом можно отдавать например, укороченный и полный объект.

При реализации нескольких доменов, можно четко понимать, какой именно объект DTO к какому домену принадлежит и ограничить его нужным домену набором полей

DTO

Шаблон проектирования, который используется для передачи данных между слоями приложения.

DTO представляет собой объект, который содержит данные, необходимые для выполнения операции или запроса в приложении.

```
namespace BookService.Models
{
    public class BookDto
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string AuthorName { get; set; }
    }
}

namespace BookService.Models
{
    public class BookDetailDto
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public int Year { get; set; }
        public decimal Price { get; set; }
        public string AuthorName { get; set; }
        public string Genre { get; set; }
    }
}
```

```
// GET api/Books
public IQueryable<BookDto> GetBooks()
{
    var books = from b in db.Books
                select new BookDto()
                {
                    Id = b.Id,
                    Title = b.Title,
                    AuthorName = b.Author.Name
                };

    return books;
}

// GET api/Books/5
[ResponseType(typeof(BookDetailDto))]
public async Task<IHttpActionResult> GetBook(int id)
{
    var book = await db.Books.Include(b => b.Author).Select(b =>
        new BookDetailDto()
        {
            Id = b.Id,
            Title = b.Title,
            Year = b.Year,
            Price = b.Price,
            AuthorName = b.Author.Name,
            Genre = b.Genre
        }).SingleOrDefaultAsync(b => b.Id == id);
    if (book == null)
    {
        return NotFound();
    }

    return Ok(book);
}
```

Далее рассматривались темы, которые предназначены для создания микросервисной архитектуры

1. **Разбиение по бизнес возможностям — 1 бизнес логика — 1 микросервис + иерархия такая же должна быть**
2. **Разбиение на поддомены DDD— берем и разбиваем все модель на поддомены, у каждого поддомена свои данные**
3. **Душитель** — пилим монолит на микросервисы --- есть фасад, который перенаправляет запросы пользователей на уже готовые микросервисы
4. и т.д.