

Как выбрать архитектурный паттерн или подход?

В прошлой [статье](#) мы разобрались, что не так с web-MVC и как его можно расширить. А сегодня будем разбираться, какой же архитектурный подход стоит выбирать и в какой ситуации. Разберём плюсы и минусы каждого подхода, в чём особенности и недостатки. Welcome под кат!

И так, какие основные критерии выбора?

- Базовая архитектура фреймворка - у **PHP** фреймворков (*Yii, Laravel, CodeIgnitor, Spiral*) зачастую это классический **web-MVC**, *Symfony* я бы отнёс к категории "условный MVC" (**MVC-like**) - из коробки он похож на **MVC** (слой модели представлен репозиторием, Entity и EntityManager) и даёт очень большую гибкость
- масштаб проекта - на сколько проект будет большим, какой у него план развития, даже какие нагрузки предполагаются
- Технический бэкграунд команды - особенно у тим/тех-лидов или архитекторов, которые и будут строить начальную архитектуру. От этого зависит не только выбор архитектуры, а ещё и шанс "выстрелить себе в ногу" в дальнейшем, например команда без опыта в **DDD** хочет сразу строить на **DDD** принципах и в дальнейшем проект превращается в ад, потому что за недостатком опыта были допущены плохие решения, которые в дальнейшем приходится "обмазывать" костылями, порождая технический долг
- Требования бизнеса - если сложная предметная область, то некоторые архитектурные паттерны будут "стрелять в ногу", и даже могут это делать на старте проекта
- Сроки и бюджет - нету смысла тратить много времени на анализ и выбор архитектуры, если бюджет - три копейки и срок - полтора месяца - просто бизнес и деньги потеряет, и проект не получит. Для такого рода проектов работает правило - **KISS - keep it simple, stupid**
- Требования к сторонним интеграциям - сколько и каких планируется интеграций со сторонними **API**, сервисами

Понятно, что это далеко не все критерии выбора. Но это основные критерии, по которым и строят архитектуры приложений на разный срок поддержки. Но сперва давайте зайдём чуть из-далека и поговорим про **стиль архитектуры** или **уровень разбиения приложения**, выбор которого стоит **выше фреймворка**. Какие уровни существуют?

Монолит

Классический монолит. Сразу сделаю ремарку, что мы говорим про сугубо backend, который не имеет под собой frontend'a. Да, монолиты это не только про "Бэкенд +

Фронтенд", а про сам стиль организации кода и бизнес-логики. Так что же такое, этот ваш **монолит**? **Монолитное приложение** - это единое целое приложение, где все слои "живут" в одной кодовой базе, вся логика и бизнес-код живут в одном процессе и кодовой базе. В классическом понимании монолит — это когда всё серверное (API, бизнес-логика, админка, или даже рендеринг HTML) лежит в одной кодовой базе и деплоится одним процессом. Как пример - starter kit в *Laravel*, где всё находится в общем **App** неймспейсе. Примерная архитектура кода выглядит вот так:

```
.
└─ app/
    ├── Controllers/
    │   ├── API/
    │   │   └─ MyApiController.php
    │   └─ DocController.php
    ├── Models/
    │   ├── Foo.php
    │   └─ Bar.php
    ├── Serializers/
    │   ├── SerializerInterface.php
    │   └─ MySerializer.php
    └─ Services/
        └─ Business/
            └─ MyBusinessService.php
```

Вся логика находится в одном месте, нету каких-либо разбиений по доменным сущностям. Реальный KISS.

Когда выбирать монолит? Тут всё очень просто - если срок поддержки проекта небольшой, проект маленький (условный *ToDo List* или *простенькое АПИ*), бюджет небольшой, срок и уровень поддержки - тоже небольшой и планируемая нагрузка невелика. Вообще, стоит сказать сразу, что при старте проекта - всегда выбирается монолит (за очень и очень редким исключением), но монолиты тоже бывают разные. И сейчас об этом поговорим.

Модульный монолит

Отличается от классического монолита тем, что вся логика разделена по доменным сущностям и находится в модулях. Но почему это монолит? Потому что признаки и условия всё те же - единая кодовая база, все слои находятся в одном приложении, просто разделены чуть глубже, единый пайплайн деплоя. Но чем же тогда отличается? Если коротко - то разделением бизнес-логики и инфраструктурного кода разделены по отдельным бизнес-доменам. Бывает полноценная модульность, и "компромиссная" модульность - когда модули - папка внутри *App*. А "компромиссная" потому, что теряется чёткое разделение на модули как *Bounded Context* - модули будут

зависимостью от корневого *App*, который зачастую играет роль **Core-module**.

Классический минус компромиссного подхода — когда из модуля можно дергать любой класс из **app/**, легко появляется неявная связанность и потеря модульности уже на старте. Так как же организовать модульность правильно? Первый и самый "лёгкий" путь - пойти на компромисс и сделать **App/Modules** структуру, второй - сделать всё чуть более красиво:

```
.
├─ app/
│   ├── Controllers
│   ├── Entities
│   └── Services
└─ modules/
    ├── Api/
    │   ├── Controllers
    │   ├── Entities
    │   └── Services
    └── Admin/
        ├── Controllers
        ├── Entities
        └── Services
```

И прописать как **PSR-4** в `composer.json`

```
{
    "autoload": {
        "psr-4": {
            "App/": "app/",
            "Modules/": "modules/"
        }
    }
}
```

И третий, самый каноничный путь, который поможет обеспечить более широкое масштабирование - **package-as-module**. Где каждый модуль - отдельный *composer package*, который физически находится в той же кодовой базе, но вайрится не напрямую через **PSR-4 autoload**, а как *symlink package* через тот же самый **composer.json**:

```
.
├─ app/
│   └── ...
└─ modules/
```

```

|   |─ api/
|   |   |─ src/
|   |       |─ Controllers
|   |       |─ Models
|   |       |─ Services
|   |       └─ composer.json
|   └─ admin/
|       |─ src/
|       |   |─ Controllers
|       |   |─ Models
|       |   |─ Services
|       └─ composer.json
└─ composer.json

```

composer.json в модуле:

```

{
    "name": "module/api",
    "type": "library",
    "version": "1.0.0",
    "autoload": {
        "psr-4": {
            "Module\\API\\": "src/"
        }
    }
}

```

корневой composer.json:

```

{
    "name": "team/our-monolith-app",
    "type": "project",
    "require": {
        "php": ">=8.1",
        "module/api": "*",
        "module/admin": "*"
    },
    "autoload": {
        "psr-4": {
            "App\\": "app/"
        }
    },
    "repositories": [
        {
            "type": "path",
            "url": "modules/*"
        }
    ]
}

```

```
}  
]  
}
```

Такой подход делается чуть сложнее чем предыдущие два, но в перспективе даст очень хорошую масштабируемость, так как в случае если какой-то модуль начинает разрастаться - то его кодовая база модуля просто выносится в отдельный сервис и с минимальными доработками имплементируется вызов отдельного сервиса в основную кодовую базу. Но это так же порождает некоторые ограничения - нужно минимизировать вызовы логики в других модулях (в т.ч. и в **Core**). Конечно, стоит сказать, что выбирая модульный монолит - всё равно стоит учитывать, что модули должны быть "**чёрными ящиками**" для других модулей, но при последнем стиле организации модульности - это становится не рекомендацией, а требованием. В идеале, модули общаются только через публичные интерфейсы/контракты, а все внутренние классы/сервисы модуля скрыты. Ну и ещё плюс последнего подхода - настоящая изоляция контекста доменной сущности, и в идеале - вообще убрать слой **app** и использовать вместо него **shared** или **core модули**. При **package-as-module** стиле внедрение новых команд и рефакторинг легче делать итеративно, а для событий или глобальных штук используется отдельный **shared/core** пакет — это **industry standard** (для битриксойдов/ворпдпрессников - "стандарт индустрии"). Когда стоит выбирать **module monolith**? Всегда, когда у проекта есть долгосрочные цели, даже если нужно просто "накидать MVP" для показа бизнеса, но и подготовка тоже будет чуть сложнее - нужно вникать в процессы бизнеса чуть глубже, но это не минус, а инвестиция. Но, повторюсь, в долгий срок это даст очень хорошую и структурированную базу. И вообще, такой подход - это и есть *Bounded Context* из **DDD**. Правда, если вам нужно сделать MVP для "вбрасывания", долгосрочная поддержка не планируется - то можно не заморачиваться, а просто писать всё в одном монолите. Но если проект - это не распил бабла инвесторов, то выбирайте **Module Monolith**, а если проект ещё и планируется большой - то **package-as-module**.

Микросервисы

Очень хороший подход, когда у вас проект вырос настолько, что масштабирование монолита (даже модульного) становится слишком затратным - горизонтальное масштабирование становится всё дороже, онбординг нового разработчика/тестера/менеджера растягивается на очень долгий срок, и вообще проект начинает трещать по швам - то вам сюда. Каждый микросервис - отдельное приложение, которое разделено по своему бизнес-домену (условно *Orders* - отдельно, *Cart* - отдельно), со своей базой данных, со своим кешем и поддержка с разработкой ложится на отдельную команду. Я ни разу не видел авантюристов, которые новый проект начинают сразу в микросервисах - это избыток, ведь затраты на поддержку всего этого становятся очень большими, даже больше чем у бизнеса заложен бюджет. Но где же тогда используются микросервисы? В очень крупных проектах, с миллионами пользователей и десятками и сотнями тысяч **RPS** (для битриксойдов - "**Request Per Seconds**", "Запросы в секунду").

Настоящий **Highload**, и только на таких проектах можно ощутить реальную выгоду от микросервисов. Да и порог чуть выше - микросервисы не вызывают "классы" друг из друга, а общаются по транспортным протоколам (может быть как **REST over HTTP**, так и разного рода **RPC** - тот же **JSON-RPC**, **GraphQL** или как стандарт - бинарный **gRPC** со строгими контрактами), или через общую шину данных (в качестве которой зачастую выступают **Message Brokers** - **RabbitMQ**, **Apache Kafka**) - **Event Bus**. Да, на первый взгляд складывается ощущение, что это избыток - ведь на транспорт данных тратится время, зачем если можно просто "купить ещё один сервак" и "раскатать туда инстанс прилаг для балансировщика нагрузки", но это ошибочно - ведь затраты на такой "сервак" будут очень большими, особенно придётся не сладко базе данных - придётся покупать отдельные сервера для базы, делать репликации, и затраты если не учетверятся, то утраиваются тоже. Поэтому самый логический шаг - выносить всё в отдельные "*микроприложения*", со своими базами данных и масштабировать их по мере нагрузки. Например, если в сервисе *User* пользователи меняют емейл раз в 10 секунд, а в корзину (сервис *Cart*) кладут тысячу товаров в секунду - то делать масштабирование можно независимо и только у сервиса *Cart*. При монолите пришлось бы масштабировать всю кодовую базу, которая на больших проектах может достигать вплоть до 10 миллионов строчек кода! А иногда и больше. И микросервисы - это путь развития, а не стартовый шаблон. Так же не стоит забывать о том, что мониторинги, трассировка, кросс-сервисные баги и пайплайн деплоя - это всё ложится на экспертность команды, само оно не появится. И логи хранить в файликах не получится - масштабы не те, дорого хранилище дергать будет, а логи и мониторинг держать не просто придётся - это становится необходимостью. Процесс эволюции - *из монолита - модульный монолит - монолит + критические модули в микросервисы - полные микросервисы*. И никак иначе. Если вам ваш техлид или СТО говорит - "давайте начнём с микровервисов" - то либо ваш этот СТО "немного" некомпетентен, либо он энтузиаст и авантюрист, либо он хочет вытянуть с бизнеса побольше бюджета. Примеры микросеврисного подхода - это и *Ozon*, и *Wildberries*, и сервисы *Яндекса*, кароче - всё что крупное и популярное (но за редким исключением). Чуть дальше мы поговорим про **SOA (Service-Oriented Architecture)**, но их нельзя путать - микросеврисы про какой-то локальный контекст, единственный бизнес-домен в сервисе, а **SOA** - более глобально. Ну и иногда не опытные разработчики называют просто мелкие куски кода - это заблуждение. Микросервис - отдельное приложение, со своим пайплайном деплоя, инфрой и они **слабо связаны** между собой.

SOA (Service-Oriented Architecture)

Об этом много говорить не будем - подход похож на микросервисный, но чуть всё же чуть глобальнее - разделяем не по конкретным доменным сущностям, а по целым бизнес-процессам. Разберём пример - флоу оплаты товара. В **микросервисах** - это отдельный микросервис с товарами *Products*, отдельный *Order*, отдельный *Cart*, отдельный *Payment*, отдельная *CRM* для "кривых" заказов. А в **SOA** - всё что связано с флоу покупки было бы одним сервисом - сервис с товарами и сервис с *Order*, в котором были бы и сам заказ, и корзина, платёжка и *CRM*. Такой подход для новых

проектов встречается чуть чаще, но как правило - в рамках уже существующей системы. Часто в **SOA** общие базы данных, более чёткие контракты и общая инфра/пайплайны. Например мой друг, редактор нашего канала, разрабатывает *самостоятельный* сервис по анализу цен на полёты - в его сервисе (хоть там и своя БД/кеш) происходит вызов внешних API, внешний REST API, админка. Его сервис вызывается в их общей CRM системе для анализа цен уже на уровне менеджеров и sales. Более упрощённый **SOA**, но соответствующий основным принципам. В микросервисной архитектуре его проект бы выглядел как три отдельных микросервиса - агрегация данных со сторонних АПИ, внешний REST API, админки и визуализации. Из минусов - контракты между сервисами должны быть жесткими, отдельная инфраструктура для шин, реестров и прочих (**ESB**), **event bus** - теперь bottle neck (для битриксидов-кубидов "бутылочное горлышко", узкое место). Ну и из плюсов - легко соединять эти сервисы между собой через шину данных и строгий контракт, так же общее место для управление политиками безопасности и аудитами. Эволюционный путь - "*из монолита в SOA*". Встречается во всяких "кровавых энтерпрайзов", у всяких больших гос. систем, у банков, крупных корпоратов и телекома. Такое самому строить с нуля будет сложно и крайне затратно. Вообще, все ноги растут из монолита.

Event-Driven

Сам по себе **Event Bus** - не целостная архитектура, а один из важных слоёв, уж особенно в **микросервисной/SOA** архитектуре. Слой этот предназначен для передачи событий между разными частями системы без "*жесткой связанности*" сервисов/компонентов. Механизм простой - существует общая "*шина*" (**Bus**), и сервисы, которые слушают определённые события, и эти же сервисы для общения друг с другом могут в эту "*шину*" эти события с данными отправлять (*публишить*). Как пример - сервис *Order* при успешном заказе публикует событие "*OrderCreated*" в очередь (топик, канал) "*orders*" внутри **Message Broker (RabbitMQ, Apache Kafka)**:

```
{
  "event": "OrderCreated",
  "user_id": 123,
  "order_price": 10000,
  "currency_id": 1
}
```

А вот сервис *Cashback* слушает (*субскрайбум*) "*orders*", и при получении события "*OrderCreated*" рассчитывает по *order_price* кешбэк и добавляет его пользователю по *user_id*. Ключевой момент - *Order* и *Cashback* ничего друг о друге не знают, и получают только дозированную информацию друг от друга. Даже о пользователе эти сервисы ничего не знают кроме его ID. Выше я привёл пример для **микросервисной** или **SOA** архитектуры, но сам **Event Bus** можно использовать ещё и для связанности между модулями внутри монолитного приложения. Пример с заказом и кешбэком можно адаптировать и для **module monolith**. Вот пример:

Модуль заказа *Order* при успешном заказе вызывает эвент (**Event**), который слушает **Listener** из модуля *Cashback*. И после получения этого эвента - уже рассчитывает кешбэк и начисляем его пользователю. Вот всё тоже самое, только без брокера очередей.

```
<?php
declare(strict_types=1);

namespace Module\Order\Events;

class OrderCreated
{
    public function __construct(
        public int $userId,
        public int $orderPrice,
        public int $currencyId,
    )
    {
    }
}
```

```
<?php
declare(strict_types=1);

namespace Module\Cashback\Listeners;

use Module\Order\Events\OrderCreated;
use Module\Cashback\Services\CashbackService;

class CashbackListener
{
    public function __construct(
        protected CashbackService $cashbackService,
    )
    {
    }

    public function handle(OrderCreated $order): void
    {
        $cashback = $this->cashbackService->calculateCashback($order->orderPrice);
        $this->cashbackService->updateUserCashback($order->userId, $cashback);
    }
}
```

```
<?php
declare(strict_types=1);
```



```
// config
return [
    Module\Order\Events\OrderCreated::class => [
        Module\Cashback\Listeners\CashbackListener::class,
    ]
];
```

Вот примерная структура, как бы это выглядело в модульном монолите. Да, это классический **Event Listener**, и он кстати описан в [PSR-14](#) стандарте.

Сам по себе такой подход очень легко комбинируется с другими архитектурными типами, ведь это позволит сделать слабосвязанными не только микросервисы, но и модули внутри одного монолита. А сами события могут быть как синхронными, так и асинхронными. В микросервисных архитектурах используется асинхронная модель, из-за того, что сам брокер сообщений - отдельное приложение. А вот в монолитах чаще всего - синхронная модель, асинхронная используется реже, хоть *Laravel* и позволяет делать Events асинхронными через *ShouldQueue* интерфейс. Использовать можно уже начиная с модульного монолита, обеспечит слабую связанность и в случае потребности в распиле на микросервисы будет практически безболезненно.

CQRS

CQRS - Command Query Responsibility Segregation - это паттерн проектирования, в котором операции по изменению (**Commands**) и чтения (**Queries**) разделяются между собой. Об этом паттерне мы поговорим дальше, когда будем говорить об самих технических подходах, а пока нам надо знать, что этот паттерн может использоваться не только в рамках одного сервиса, но и в рамках более глобальной архитектуры. В крупных проектах иногда это даже разделяется на уровне базы данных (чтение более легковесная операция), когда большая нагрузка идёт больше на чтение, чем на запись (всякие витрины, рекомендации через агрегированные состояния и прочее). Штука очень мощная, но так же, как и при микросервисной архитектуре - при старте проекта очень избыточно (сейчас речь идёт не об **CQRS** на уровне сервиса, а на более высоком уровне), внедрять надо только при необходимости и после тщательного изучения метрик и аналитики. В простых/маленьких проектах **CQRS** часто будет избыточным, зато в реально нагруженных системах позволяет упростить поддержку, расширяемость и повысить отказоустойчивость. Но всегда стоит иметь ввиду, что это и зачем нужно.

Вот мы и разобрали основные архитектурные стили и уровни деления приложений. Конечно, на самом деле таких стилей как собак не резанных, и мы затронули только основные, но остальные более нишевые, и в рамках этой статьи даже называть их не будем - архитекторы и так всё знают, а аудитория статьи больше рядовые разработчики (и иногда даже кодеры уровня битрикс/вордпресс). Поэтому сейчас переходим к блоку о технической реализации архитектуры на уровне приложения/

сервиса. И начнём с классического во фреймворках - **CDA** (**web-MVC**, почему именно с припиской web - почитать можно [ТУТ](#)).

CDA (web MVC)

CDA - Controller-Driven Architecture, он же **MVC - Model-View-Controller**, он же **MVCS** - как **MVC + Service** - де-факто стандарт в бэкенд разработке. Файловая структура выглядит примерно вот так (Views - опционально):

```
.
└─ src/
    ├── Controllers/
    │   └─ MyController.php
    ├── Model/
    │   └─ User.php
    └─ View/
        └─ my-view.php
```

Самая база, стандарт, классика, называйте как хотите. Но это не очень практичная архитектура, потому что мешаются слои, смешивается ответственность и вообще при росте проекта начинаются большие проблемы при поддержке приложения. Поэтому будем выделять слой бизнес-логики в отдельный слой - **Services**.

```
.
└─ src/
    ├── Controllers/
    │   └─ MyController.php
    ├── Model/
    │   └─ User.php
    ├── Services/
    │   └─ MyService.php
    └─ View/
        └─ my-view.php
```

Но это тоже не панацея, ведь слой модели всё равно остаётся смешанным, ведь там определяется и сущность, и сама работы с базой данных. Это тоже усложняет и тестируемость, и ответственность мешается, кароче - ужас. Как это фиксировать? Легко, теперь модель будет просто агрегатором данных, а всё общение будет происходить через репозитории и **Entity** сущности.

```
.
└─ src/
    └─ DTO/
```

```
|   └─ UserDTO.php
|── Controllers/
|   └─ MyController.php
|── Entity/
|   └─ User.php
|── Model/
|   └─ User.php
|── Repositories/
|   └─ Contracts/
|       └─ MyRepositoryInterface.php
|       └─ MyRepository.php
└─ Services/
    └─ MyService.php
```

Да, на первый взгляд структура усложняется в разы, но это только на первый взгляд. Такое "усложнение" позволяет разделить бизнес-логику от конкретных инфраструктурных реализаций, так разделять ответственность - **Entity** это бизнес-сущность, модель - хранение данных, а репозиторий - за доступ к этим самым данным. Ну а так же тестируемость, моки слоёв за данными. Так же такая структура очень хорошо ложится в **модульный монолит**, где каждый модуль - отдельный **CDA**. А если реализуем какой-нибудь **RPC** - то желательно **Controllers** называть как **Handlers**, тогда архитектура будет называться **HDA - Handler-Driven Architecture**. Ну не буду пересказывать свою прошлую статью, за подробностями - [сюда](#), а этот шаблон берите на вооружение, вещь можно сказать универсальная. И строить такую архитектуру - нужно даже если у вас проект с бюджетом "3 копейки" и сроком в месяц - за редким исключением, когда вы точно знаете, что код потом никому не понадобится.

DDD

DDD, он же **Domain-Driven Design**, подход в архитектуре приложений. Стоит сразу сказать, что **некорректно** называть **DDD** архитектурным паттерном, потому что **DDD** - это больше про *согласование языка бизнеса и кода, про глубокое моделирование процессов и понятий* проекта/компании, а не только как неймспейсы с классами в репозитории хранить. Да что уж говорить, при грамотном подходе опытные команды **DDD** архитектуру могут строить месяцами, ведь это требует очень глубокого понимания предметной области. А в рамках этого блока мы будем описывать лишь очень малую часть **DDD**, которая про "хранения кода в папках на проекте". В этом ключе основная идея **предметно-ориентированного проектирования (DDD)** довольно проста - вся архитектура завязывается на бизнес-домен (**Domain**) - вокруг реальных областей, с которыми работает бизнес, в котором реализованы все контракты (в контексте контрактов речь идет не только про интерфейсы, но ещё и всякие **DTO**, **Entity** и тому подобное). Самое главное - чтобы домен был изолирован от реализации. В слое **Application** находится вся бизнес-логика, которая по традиции называется юз-кейсами (**UseCases**, это грубо говоря тоже самое, что и **Service**).

UseCase - это отдельный бизнес-сценарий (например, “создать заказ”, “рассчитать доставку”), который оркестрирует взаимодействие доменных сущностей и сервисов, не смешивая детали инфраструктуры. В юзкейсах так же находятся и обработчики событий, и интеграции разных доменов между собой. А всё, что связано с данными - относится к инфраструктурному слою (**Infrastructure**), там и реализации репозиториев, вызовы внешних API, брокеров и так далее. А всё, что "смотрит наружу" - может находится либо в **UI** слое, либо **Interface**, либо же в **Transport** слое (если бэкэнд - **REST/RPC**). Стоит так же сказать, что методология **DDD не диктует структуру папок** и файлов, опять же, она вообще про другое, поэтому некоторые слои могут отличаться по названию, но главное - чтобы не по содержимому. Стоит так же заметить, что эта парадигма прекрасно ложится и на отдельные микросервисы, и на монолиты. Самое главное - чтобы все бизнес-домены были разделены по **Bounded Context** (по границе предметных областей), все сущности одного домена "живут" только в нём, а всё общее (**DTO**, **ValueObject** и прочие контракты) - выносим в **shared**. Вот примерная структура **DDD** приложения. Не забываем, что в **Domain** находятся контракты, а реализация бизнес-логики - в **Application** (хотя паттерн **Domain Service** позволяет в доменной области хранить реализации, например **Services**, когда бизнес-логика “не помещается” в одну **Entity** или **ValueObject**), в инфре (**Infrastructure**) - реализации баз, и прочего инфраструктурного, а в транспорте - всякие контроллер/хандлеры и прочее. А **Domain Service** требуется, если бизнес-логика более объёмна, чем для единой **Entity** или предполагает работу с несколькими сущностями напрямую, но остаётся “чистой” (не имеет кода инфраструктуры).

```
.
└─ src/
    └─ Order/
        └─ Domain/
            ├── Entity
            ├── ValueObject
            ├── Repository
            ├── Event
            └─ Services
        └─ Application/
            ├── Command
            ├── Query
            └─ Handler
        └─ Infrastructure/
            ├── Repository
            └─ Event
        └─ Transport/
            ├── Controller
            └─ Request
```

└ Shared/
└ ...

Но не забывайте, что перед тем как реализовывать **DDD** - сразу надо начать изучать все бизнес-процессы, а этот процесс не быстрый и требует **глубокого понимания** бизнеса (или плотную работу с людьми со стороны бизнеса). Просто взять и склепать **DDD** за неделю-две - зачастую не получится, можно не то, что в ногу себе выстрелить, а ещё куда повыше. Поэтому подходить к выбору этой парадигмы нужно крайне обдуманно, просто взять и "щас как навернём папочек под ди-ди-ди" *не получится*, и крайне губительно для проекта. Если не разбираетесь досконально в бизнесе, или у бизнеса нету средств на оказание такой поддержки в анализе, то лучше возьмите **CDA** и не морочьте себе голову. А перенос проекта на **DDD методологию** можно отложить до лучших времён - для будущего роста, и это абсолютно нормально.

Гексагональная архитектура

Hexagonal и **Clean Architecture** - архитектура, в котором вся бизнес-логика находится в "**ядре**", написана без учёта инфраструктуры (тут под инфраструктурой мы будем понимать вызовы в БД, кеш, внешние API и так далее. Ведь будем же понимать, да, битриксойды?). А весь инфраструктурный код пишется в **адаптерах (Adapters)** и "стыкуется" с "**ядром**" через **порты (Ports)** - специальные контракты. Условно, структура будет выглядеть так:

```

      [REST API]      [Console]
      |               |
[Input Adapter]  [Input Adapter]
      |               |
      [Port – интерфейс UseCase]
      |
[Application Layer] (ядро)
      |
[Port – интерфейс репозитория]
      |
[Output Adapter: PostgressRepo, RedisCache, S3Uploader]
```

В ядре происходит весь бизнес-процесс. Мы вызываем контракты (через интерфейсы) на получение данных, выполняем какую-нибудь логику, и снова вызываем контракты на сохранение данных в базу и кеш. Тут ключевой момент - всё идёт через контракты (где контракт - интерфейс в коде), ядро не знает о реализациях работы с базой, кешами и прочими транспортом. Зачем такое нужно? Всё просто. Изолируемость и тестируемость - можно менять хоть фреймворк, хоть протокол общения, хоть базы данных - основной функционал будет работать как прежде. Можно всю логику покрыть юнит тестами и замочать контракты - и даже если мы все адаптеры поменяем - тесты

не упадут. Но есть один ключевой момент - нельзя просто так взять и построить "гексагоналку" на условной *Laravel* или *Symfony* - мы завязываемся на конкретный фреймворк в нарушение принципов этой архитектуры. Вся бизнес логика строится вне зависимости даже от фреймворка. Применять её можно хоть на монолите, хоть на микросервисах - без разницы. А использовать - только от потребностей проекта, ибо время на продумывания контрактов и слоёв будет уходить чуть больше, чем "накидал крудов по АПИШке и пошёл". Ну и опять же, на "быстрых" проектах-единорогах, с маленьким бюджетом и сроками внедрять гексагоналку будет *overengineering*, затраты по времени и средствам больше при непонятных выхлопах. А на проектах, у которых есть большие цели и потенциал развития - само то, если одно из требований к проекту - полное покрытие тестами. А если проект эволюционирует - то поменять "обязку" (например, переехать с **REST** на **gRPC**, или заменить **Redis** на **Kafka**) - проще простого, потому что ядро не зависит от конкретных реализаций. Есть кстати ещё **Onion Architecture** - очень похожа на "гексагоналку", только с ещё большей изоляцией слоёв.

CQRS и Event Sourcing

Чуть выше мы рассматривали паттерн CQRS в рамках стилей архитектуры, а сейчас поговорим про него в рамках именно технической реализации в коде. Тут тоже самое что и на уровне выше, только организовано на уровне сервисов. И тут всё так же работает принцип разделения изменения и чтения данных на команды (Commands) и запросы (Queries). И делается это всё для того же, для чего и на уровне организации сервисов - разделение чтения и изменения для масштабных систем с нагрузкой на чтение. Зачем это делается - разделение ответственности (ибо Queries могут быть с очень большой логики), поддержка разных баз данных (особенно в аналитических системах, где данные могут записывать в реляционную базу по типу Postgres, а чтение - из колоночной базы по типу ClickHouse, и всё это для единых сущностей), моки и тесты, ну и меньшая связанность, да. Тоже стоит прибегать только в случае, если заранее известно что сервис будет нагружен *for read*, или в процессе эволюции проекта. Сам по себе паттерн не сложный, но требующий хорошего понимания архитектуры и на небольших проектах может быть сильно избыточен. Если хочется использовать сразу - то нужно чётко понимать, зачем это делать и какие будут преимущества. И иногда CQRS может быть неполноценным без Event Sourcing. А Event Sourcing в свою очередь - это паттерн, который позволяет не сразу записывать "срез" данных в БД, а вызывается эвент, который сохраняется в отдельную таблицу вместе с timestamp. И это нужно для того, чтобы хранить всю историю изменений в виде событий, и в случае чего откатить обратно. CQRS даёт масштабируемость, Event Sourcing - аудит и возможность восстановить состояние на любой момент времени. Вместе это мощная основа для систем, где важна как скорость, так и "историчность". Да, два этих паттерна могут существовать по отдельности, но бывают такие моменты, когда вместе им будет гораздо лучше. Так что это тоже нужно иметь в виду.

ADR

Современная альтернатива **MVC** - **Action-Domain-Responder**, популярна для **REST API** и "чистых" приложений. Если коротко - то:

- **Action** - это аналог контроллера, только занимается тем, что принимает данные. Ограничен тем, что он action per endpoint - делает только одну вещь
- **Domain** - вся бизнес-логика, работает с сущностями, сервисами, репозиториями
- **Responder** - формирует ответ, **JSON**, **XML** или **HTML** (в зависимости от приложения)

В слое **Domain** находятся вся бизнес-логика, как уже сказано выше, и идёт работа с данными. Вот как примерно выглядит структура:

```
.
└─ src/
    └─ Action/
        └─ CreateBurgerAction.php
    └─ Domain/
        └─ Services/
            └─ BurgerService.php
        └─ Entity/
            └─ Food.php
        └─ Repository/
            └─ Contracts/
                └─ FoodRepositoryInterface.php
            └─ FoodRepository.php
    └─ Responder/
        └─ CreateBurgerResponder.php
```

В отличие от **MVC**, где контроллер часто совмещает и бизнес-логику, и формирование ответа, в **ADR** каждый слой отвечает только за своё: **action** - только принимает и валидирует, **domain** - реализует бизнес-сценарий, **responder** - только формирует ответ. А ещё эта архитектура делает код прозрачней, что позволяет удобней тестировать бизнес-логику, не затрагивая *экшен или респонс*. А так же эта структура отлично ложится на модульную архитектуру, где каждый модуль будет не **CDA/MVC**, а **ADR** (к слову, тот же **CQRS** тоже на модульность ложится отлично). Всё чисто, прозрачно и удобно. Чёрт побери, да я и сам в таком подходе писать буду! Можно ли использовать в своих проектах? Конечно! Архитектура проста, слои понятны. Если вам при старте какого-либо проекта будет не лень потратить время на адаптацию кода из фреймворка (я сейчас больше про такие фреймворки, как *Laravel*, *Yii* или *CodeIgniter*, *Symfony* сам по себе гибкий) под **ADR** - то только рекомендую попробовать. А если вы начинаете новый проект вне фреймворка - то даже проще будет. Поэтому если есть привычка к **MVC** - то можно смело пробовать **ADR**, на первых парах это будет скорее необычно, но для *API* или *API-first* проектов это часто оказывается *прозрачнее и проще*.

Command Bus и Mediator

Последний в этой статье, но не последний по значимости, как говорится. **Command Bus** - архитектурный паттерн, особенностью которого является то, что все действия не вызываются в коде напрямую, а проходят через общую централизованную "шину" (**Bus**). **Mediator** - это более общий подход, в котором вызываются не только определённые команды, а вообще разные компоненты. Например - обработка сложного бизнес-процесса, где происходит вызов сразу нескольких *handlers* в одном *workflow*.

Общие особенности:

- **Command** - простой класс DTO, который описывает действие (например, создать заказ, отправить письмо)
- **CommandHandler** - класс, который реализует обработку этой команды (бизнес-логика)
- **CommandBus** - принимает команду, ищет нужный handler и вызывает его
- **Middleware** - "обёртки" вокруг шины, чтобы добавить логи, транзакции, очереди, валидацию и т.д.
- **Query/QueryHandler** - для реализации CQRS и централизованной обработки запросов

```
.
└─ src/
    └─ Commands/
        └─ CreateBurgerCommand.php
    └─ CommandHandler/
        └─ CreateBurgerHandler.php
    └─ CommandBus/
        └─ CommandBus.php
        └─ Middleware/
            └─ LoggingMiddleware.php
    └─ Query/
        └─ GetBurgersQuery.php
```

Стоит ли начинать проект именно из **Command Bus** или **Mediator**? Вряд ли, тут как в случае и с **CQRS** (они в целом связаны) - нужна очень большая экспертиза, хорошая аналитика проекта на счёт нагрузки и масштабов. Но если проект начинает разрастаться - то помнить об этом не самом популярном паттерне стоит, ибо позволит связать разные компоненты или модули без особой головной боли, и в случае попила на микросервисы - заменить вызовы будет ещё проще. А в большинстве современных **PHP** фреймворков **Command Bus** часто внедряется как отдельная библиотека или компонент (*Symfony Messenger*, *Laravel Bus chain* (как часть *Jobs*), *Tactician*). Это

позволит внедрить этот паттерн на уже готовую архитектуру тогда, когда понадобится, не меняя всю архитектуру проекта.

Ну что же, спустя 4768 слов мы разобрали все популярные типы архитектур и архитектурные паттерны и определили, что и в каких случаях лучше или хуже использовать. Теперь резюмируем - всё зависит от проекта, как бы банально это не звучало. Если проект-"единорог", или нету времени/бюджета - то можно вообще не париться и писать так, как предлагает фреймворк. Но в моменте это может быть очень опасно - а вдруг продукт выстрелит, или бизнес даст денег на развaитие - и всё, либо плодить легаси/тех. долг, либо стопать все новый фиши и заняться рефакторингом. Поэтому если есть точное понимание, что проект полетит в "топку" - то пофигу. Но если есть хоть малейшие подозрения на успех - то лучше использовать **CDA/MVCS с разбиением слоя модели** или в качестве эксперимента и чистоты **ADR**. Вместо **монолита - модульный монолит** где используется подход **package-as-module**. Если проект начинается как "*младенец энтерпрайза*" и у бизнеса есть достаточно средств и людей с компетенциями - то стоит потратить достаточно времени и начать проектировать всё по **DDD**. На таком проекте это точно даст хороший задаток на будущее развитие. Но не стоит сразу проектировать под **микросервисы**, или **SOA**, а код организовывать по **CQRS/Command Bus** (вернее стоит только в том случае, если вы точно знаете, что делаете), это черевато проблемами на старте и потерей денег/времени. С **гексагональной архитектурой** всё примерно так же, как и с **DDD** - стоит начинать если у проекта есть весомое требование к тестируемости или к независимости слоёв, и эта ситуация тоже требует хорошего продумывания абстракций, опытную команду и хорошую экспертизу.

А на этом всё, в следующей статье мы разберём протоколы передачи данных и API интерфейсы. Всем пока.