C# Developer. Professional

Архитектура проекта

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Архитектура проекта

Елена Сычева

Team Lead Full-Stack Developer

Об опыте:

Более 15 лет опыта работы разработчиком (C#, Angular, .Net, React, NodeJs)

Телефон / эл. почта / соц. сети:
https://t.me/lentsych

Правила вебинара

Активно
участвуем

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально

Время, необходимое
на активность

Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

Маршрут вебинара

- Что это? Зачем это?
- Слои и связи
- API интеграция
- Выбор проектной архитектуры
- Практика

Цели вебинара

К концу занятия вы сможете

1.	Узнать об основных аспектах архитектуры приложений в контексте выпускной работы
2.	Понять, с чего начинать проект выпускной работы

Смысл

Зачем вам это уметь

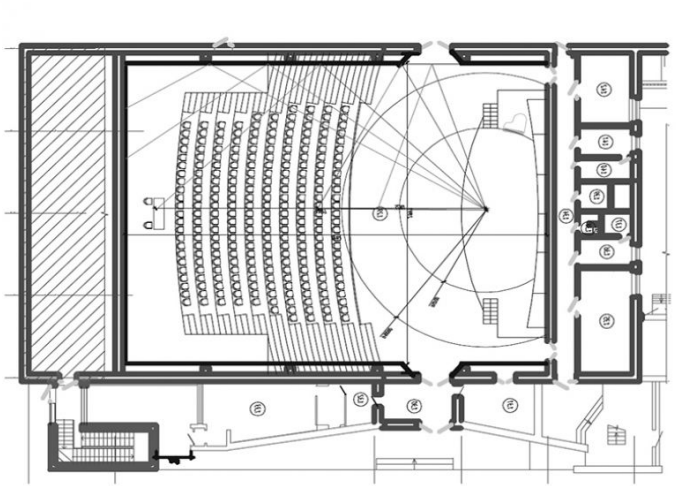
1. Понимать с чего начать
2. Понимать какая архитектура у существующего проекта

1. Какие архитектуры Вы уже знаете?

Архитектура. Что это?

Что такое архитектура проекта?

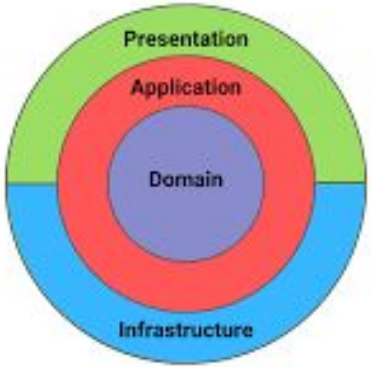
- Архитектура проекта относится к фундаментальной структуре приложения, описывающей, как компоненты взаимодействуют друг с другом, и принципы, управляющие этими взаимодействиями. Она действует как план для разработки, развертывания и обслуживания приложения.



Цель четко определенной архитектуры

Ясность:

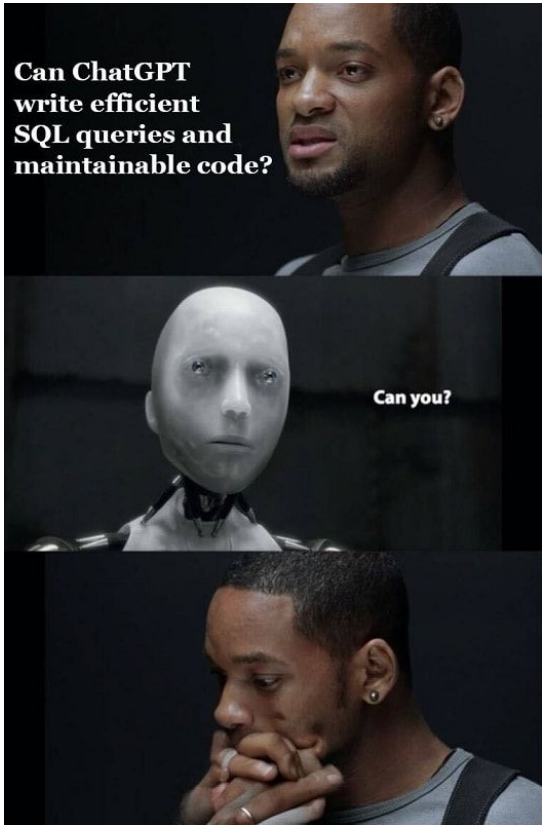
Четко определенная архитектура помогает разработчикам и командам понять, как работает система, что упрощает прием новых членов команды или устранение неполадок.



Цель четко определенной архитектуры

Поддерживаемость:

Хорошая архитектура гарантирует, что кодовая база останется организованной и ее будет легко изменять. Например, при внедрении новых функций или исправлении ошибок изменения могут быть реализованы без непреднамеренных последствий в других частях приложения.



Цель четко определенной архитектуры

Масштабируемость:

Хорошо структурированная архитектура поддерживает рост. По мере расширения пользовательской базы или функциональности приложения архитектура может вмещать изменения с минимальной реструктуризацией.

Цель четко определенной архитектуры

Эффективность:

Четкое разделение задач (например, разделение задач между различными компонентами) обеспечивает эффективную производительность и использование ресурсов.

Цель четко определенной архитектуры

Повторное использование и стандартизация:

Компоненты, созданные с использованием принципов архитектуры, часто можно повторно использовать в разных проектах или модулях, что экономит время и усилия.

1. С чего начинать?

Порядок действий

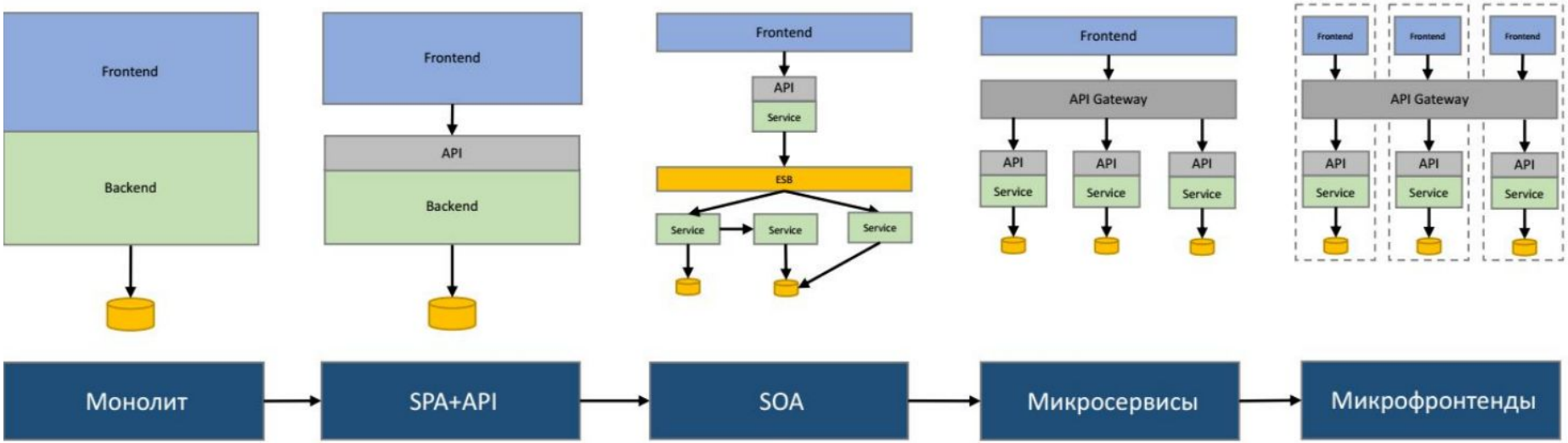
1. Определите требования

   Определите нефункциональные требования: рассмотрите потребности в производительности, масштабируемости, безопасности и удобстве обслуживания. Пример: для приложения блога определите такие функции, как аутентификация пользователей, создание постов, обработка комментариев и ожидания производительности для одновременных пользователей.

2. Выберите тип архитектуры (*Микросервисы*)
3. Определите логический дизайн (*трехслойная, луковая, чистая…*)
4. Определите ключевые компоненты и абстракции
5. Определите структуру API и связи

Тип архитектуры

Хронология развития архитектур

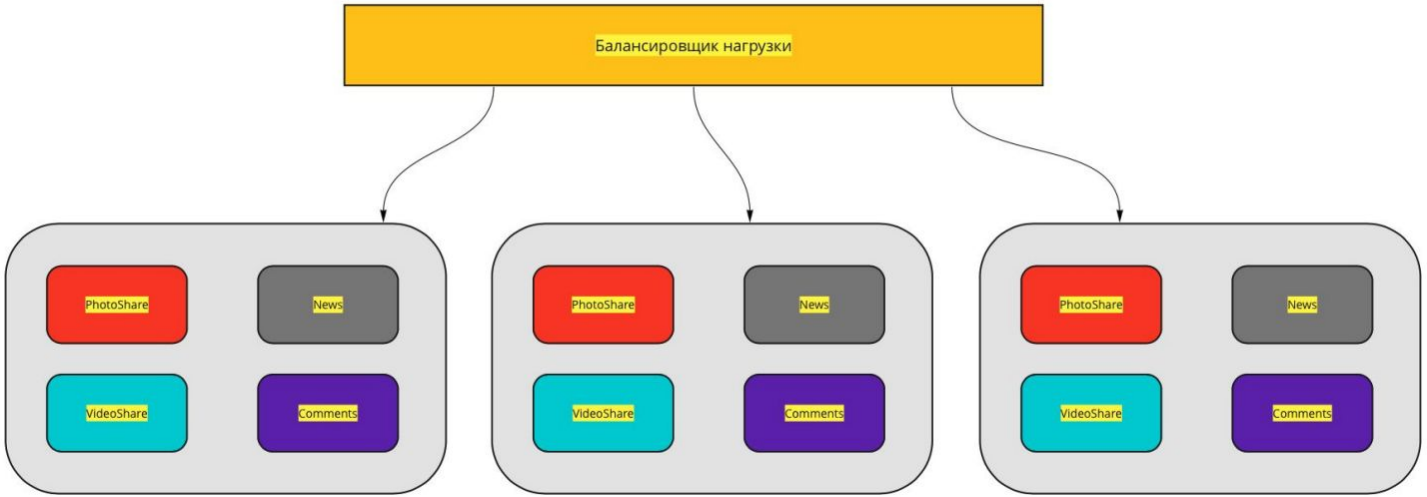


Типы системной архитектуры

1. Монолитная архитектура:
    - Определение: все приложение построено как единое, унифицированное устройство.
    - Преимущества: простота разработки и развертывания для небольших проектов.
    - Проблемы: сложность масштабирования и поддержки по мере роста приложения.
2. Микросервисная архитектура:
    - Определение: приложение разделено на более мелкие независимые службы, которые взаимодействуют через API.
    - Преимущества: масштабируемость, модульность и гибкость для разработки и развертывания.
    - Проблемы: повышенная сложность управления и оркестровки служб.

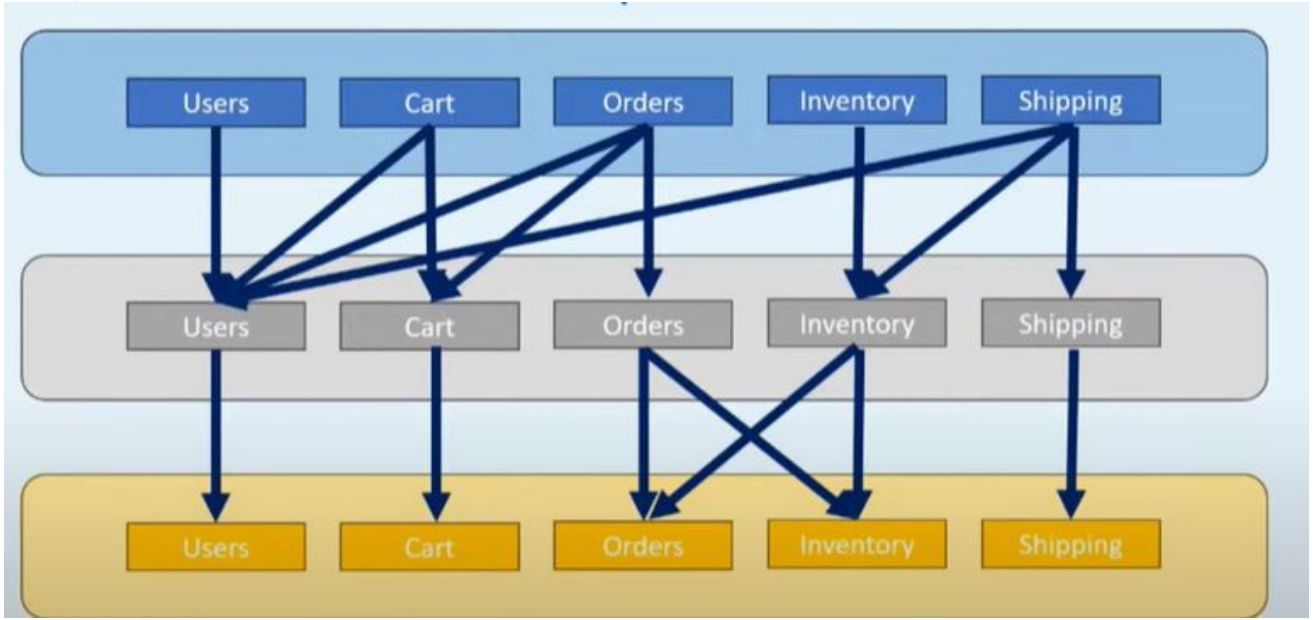
Проблемы масштабируемости в монолитах

Масштабирование всего приложения по сравнению с отдельными службами.
Неэффективность ресурсов.



Проблемы разработки в монолитах

- Взаимозависимости, замедляющие разработку.
- Длительные циклы тестирования и развертывания.

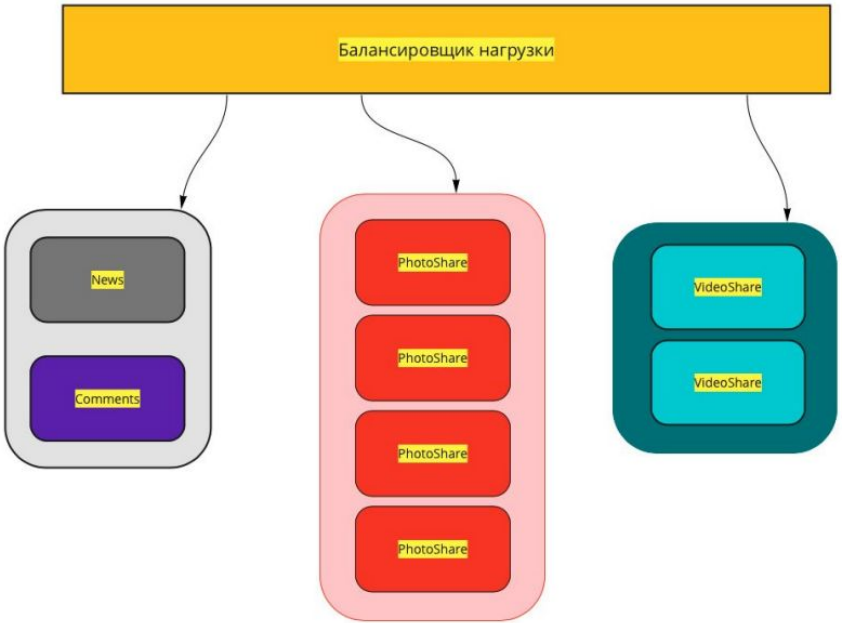


Поддерживаемость и гибкость. Проблемы в монолитах

- Сложность внесения изменений в большой монолит.
- Риск внесения ошибок при каждом обновлении.
- Сложность тестирования.

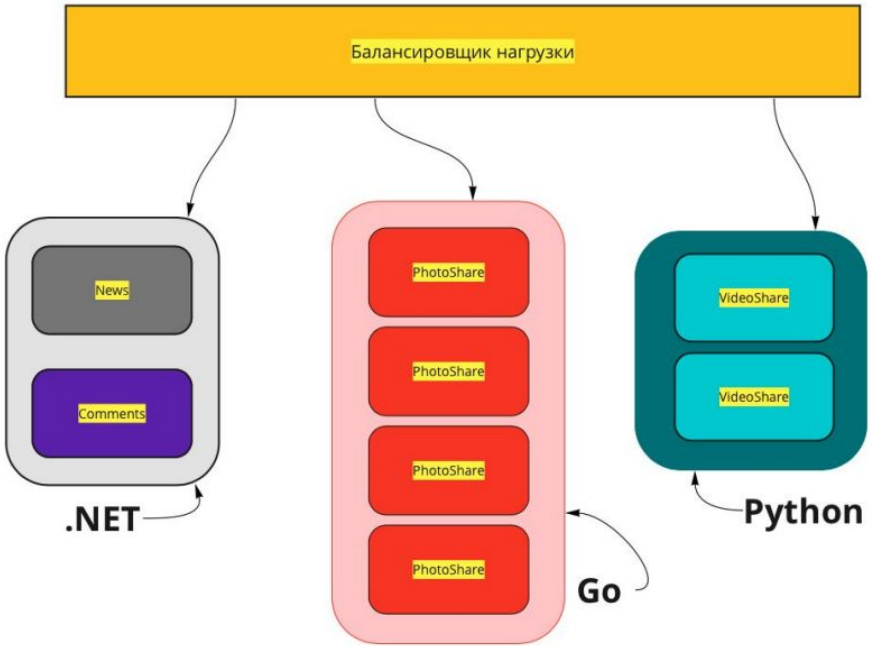
Микросервисы: Независимое масштабирование

Мелкозернистая масштабируемость.



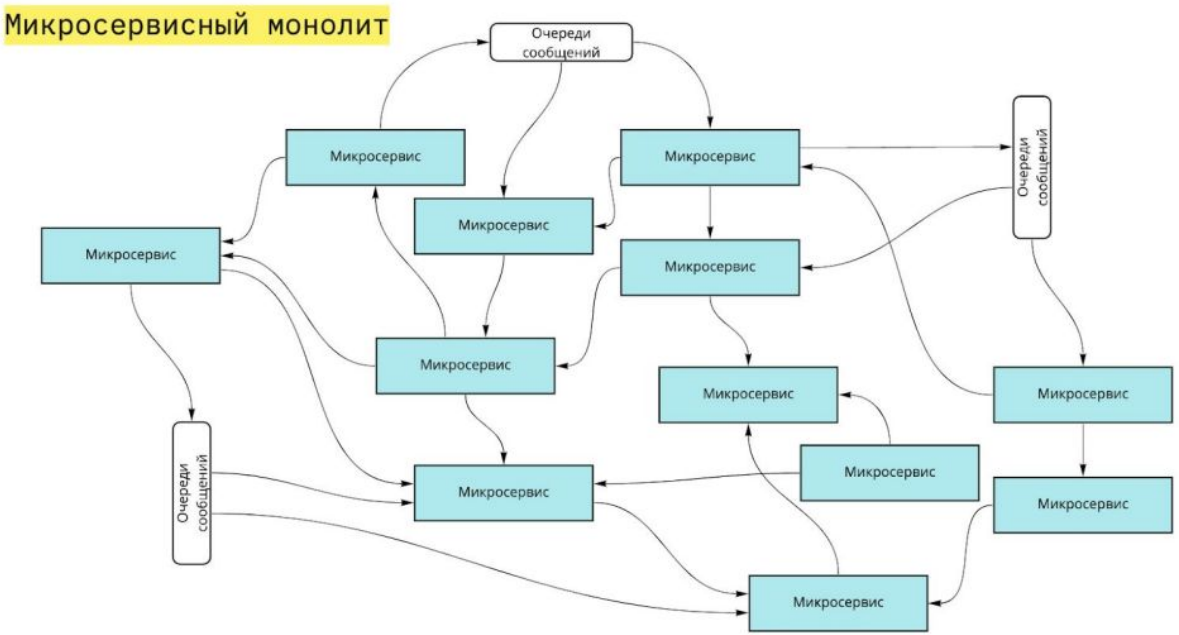
Микросервисы: Независимая разработка

- Независимая разработка и развертывание.
- Разнообразие технологий и простота обновлений.



Как общаются микросервисы

Синхронное общение



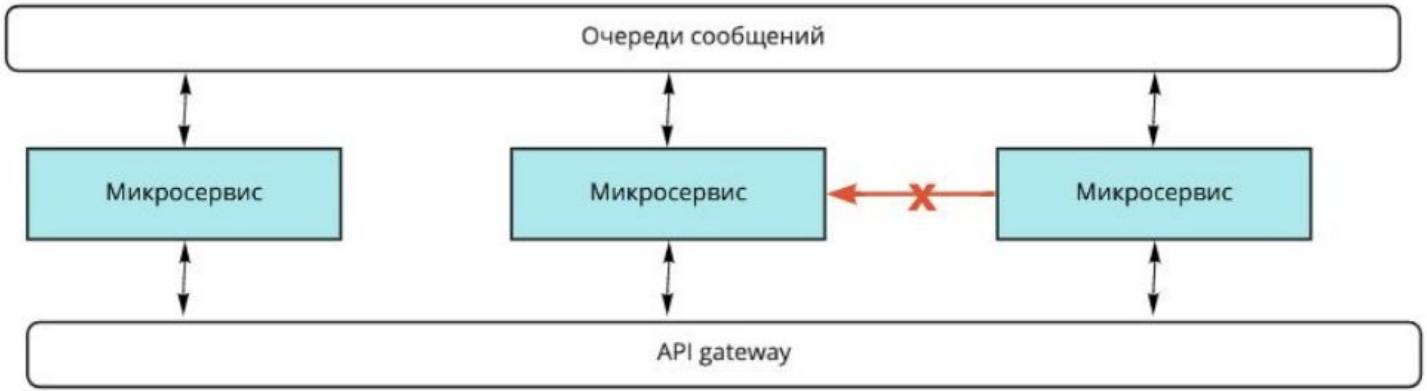
- ТТР
- gRPC
- GraphQL
- Когда нужен мгновенный ответ

Синхронное общение. Минусы

- ➔ А если вызываемый микросервис сам делегирует ответ другому микросервису синхронно?
- ➔ Чем больше микросервисов в пути обработки запроса, тем сложнее понять что происходит
- ➔ А если баг в микросервисе на N+1 шаге обработки запроса?

Асинхронное общение

Микросервисы



Асинхронное общение. Плюсы

- ➔ Микросервисы ничего не знают друг о друге
- ➔ МС занимается обработкой своей задачи по запросу
- ➔ Интеграция через очереди сообщений

Асинхронное общение. Минусы

- ➔ Если микросервисов в системе over9000, то сложно понять систему
- ➔ Можно случайно разработать новый микросервис под задачу, не зная, что уже есть решение у другой команды
- ➔ Бизнес тоже часто не может упомнить все свои инструменты

Типы системной архитектуры

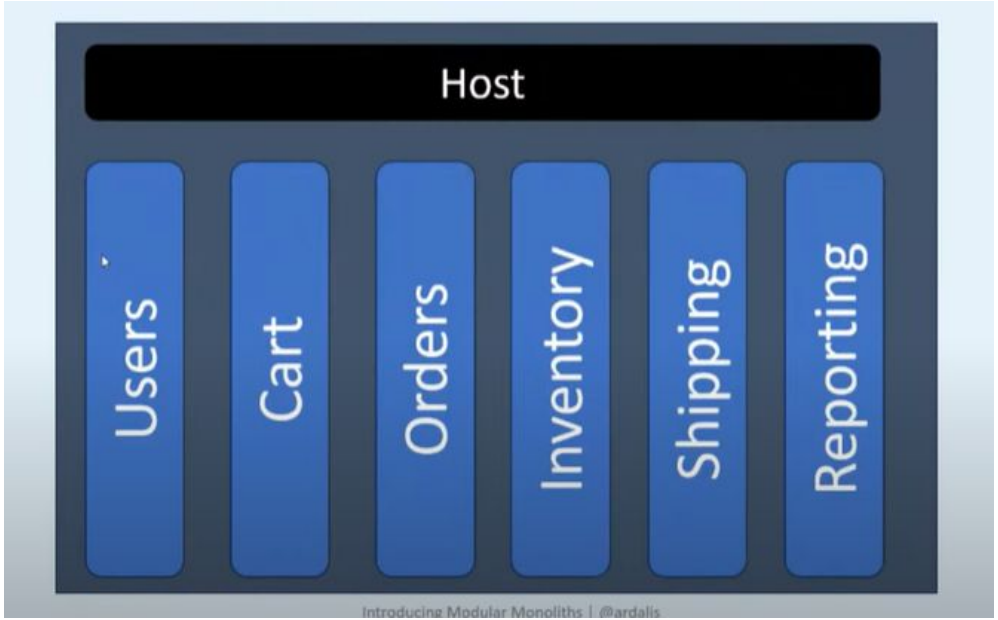
Modulith по сути является монолитом с:

- Четкими границами модулей: приложение разделено на связные модули, которые инкапсулируют определенную функциональность.
- Межмодульное взаимодействие: модули взаимодействуют друг с другом через четко определенные интерфейсы, что снижает тесную связь.
- Независимость в разработке: разработчики могут работать над модулями независимо, почти так же, как если бы они были отдельными службами, но без накладных расходов на управление распределенными системами.

Модульный монолит для управляемости

- Модульные границы и инкапсуляция.
- Постепенный переход к микросервисам.

.



Микросервисы: ускоренная разработка

- Параллельная разработка и более быстрая доставка.
- Сокращение времени выхода на рынок.

Модульный монолит: целевые команды

- Целевые команды разработки.
- Сокращение накладных расходов в модульных монолитах.

Что ещё бывает

Event-Driven Architecture (Управляемая событиями архитектура):

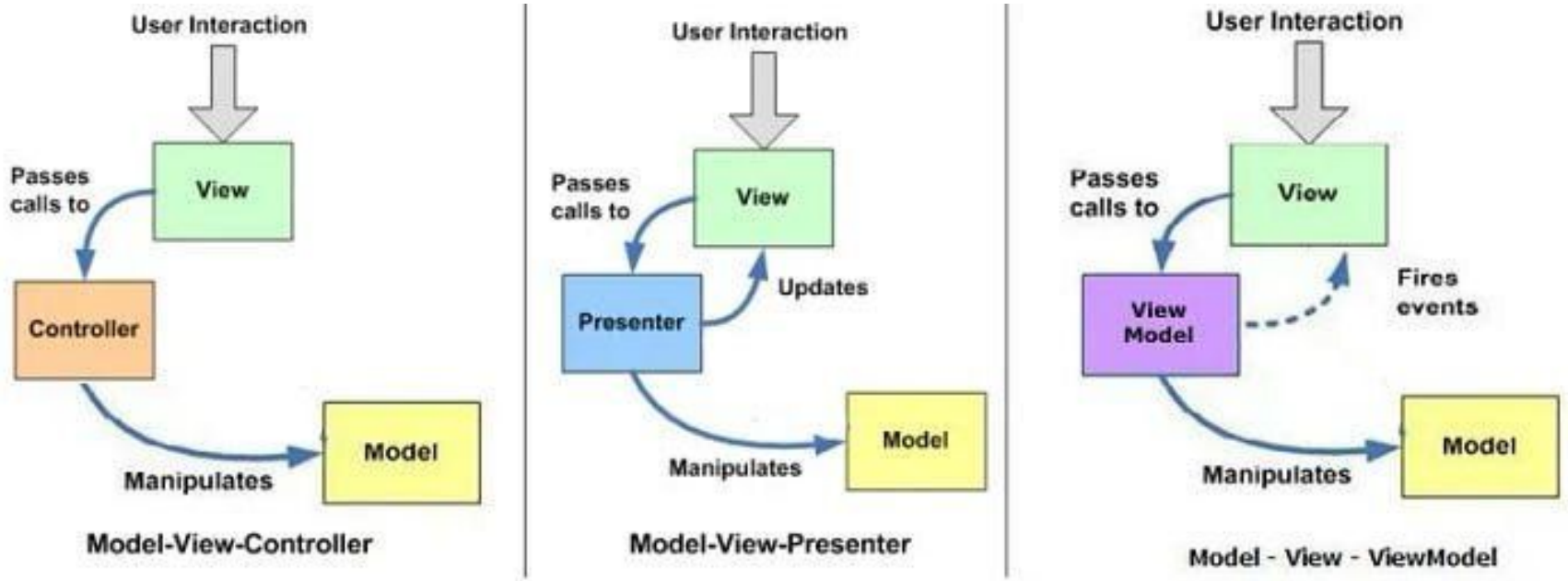
- Определение: компоненты взаимодействуют через события (сообщения) вместо прямых вызовов.
- Преимущества: асинхронность, масштабируемость и устойчивость к сбоям.
- Случаи использования: приложения реального времени, такие как биржевая торговля или платформы IoT.

Serverless (Бессерверная архитектура):

- Определение: приложение работает в инфраструктуре облачного провайдера, при этом разработчики сосредоточены на написании функций, а не на управлении серверами.
- Преимущества: экономичность, масштабируемость и снижение эксплуатационных расходов.
- Проблемы: Привязка к поставщику и задержки при холодном запуске.

Внутренняя архитектура (Логический дизайн)

Начало


User Interaction
Passes calls to
View
Controller
Model
Manipulates
Model-View-Controller
User Interaction
Passes calls to
View
Presenter
Updates
Model
Manipulates
Model-View-Presenter
User Interaction
Passes calls to
View
View Model
Fires events
Model
Manipulates
Model - View - ViewModel

2002 год: n-слойная

В своей книге Patterns of Enterprise Application Architecture *(«Шаблоны корпоративных приложений»)* он описал n-слойную архитектуру.

Подробно о слоях

Уровень представления:

- Цель: обрабатывает взаимодействие с пользователем, включая элементы пользовательского интерфейса и пользовательского опыта.
- Обязанности: отображение данных из уровня бизнес-логики и сбор ввода пользователя.
- Примеры:
- Веб: представления или страницы в фреймворках MVC.
- Настольный компьютер: приложения WinForms или WPF.

Уровень бизнес-логики:

- Цель: реализует основные функции и бизнес-правила приложения.
- Обязанности: обработка данных, полученных из уровня представления, и координация с уровнем доступа к данным.
- Примеры: вычислительные механизмы, правила проверки, рабочие процессы.

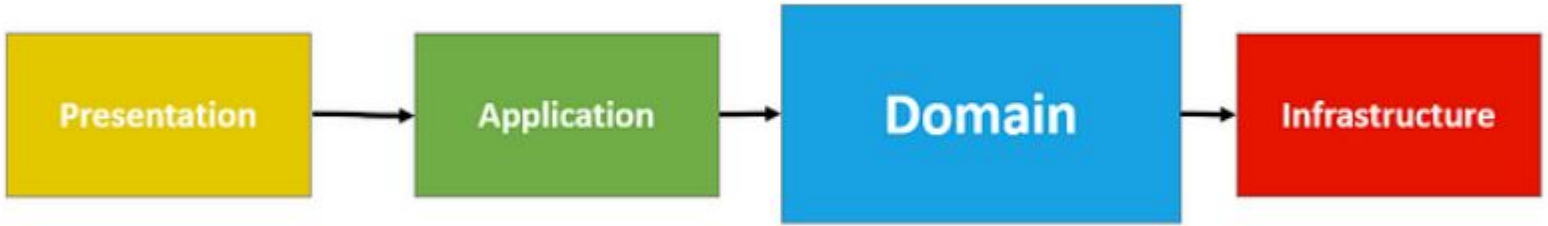
Уровень доступа к данным:

- Цель: взаимодействует с базой данных или другими системами хранения.
- Обязанности: извлечение, сохранение и обновление данных. Гарантирует, что уровень бизнес-логики не будет напрямую манипулировать источниками данных.
- Примеры: репозитории, инструменты ORM (например, Entity Framework).

2003 год: DDD

В 2003 Эрик Эванс опубликовал книгу Domain-Driven Design: Tackling Complexity in the Heart of Software

(«Предметно-ориентированное проектирование. Структуризация сложных программных систем»)



Принцип инверсии зависимостей (DIP)

```
public class OrderManager {
    public void Process(Order order) {
        var repository = new OrderRepository();
        var notificationSender = new NotificationSender();

        if(order.IsValid()) {
            repository.Save(order);
            notificationSender.Send(order);
        }
    }
}
```

```
public class OrderRepository {
    public void Save(Order order) {
        //Сохранение заказа в БД
    }
}

public class NotificationSender {
    public void Send(Order order) {
        //Отправка уведомления на почту пользователя
    }
}
```

Здесь OrderManager зависит от OrderRepository и NotificationManager

Принцип инверсии зависимостей (DIP)

```
public interface IMailSender {
    void Send(Order order);
}

public interface IOrderRepository {
    void Save(Order order);
}
```

```
public class OrderRepository : IOrderRepository {
    public void Save(Order order) {
        //Сохранение заказа в БД
    }
}
```

```
public class MailSender : IMailSender {
    public void Send(Order order) {
        //Отправка уведомления на почту пользователя
    }
}
```

```
public class OrderManager {

    private readonly IOrderRepository _orderRepository
    private readonly IMailSender _mailSender;

    public OrderManager(IOrderRepository orderRepository, IMailSender mailSender) {
        _orderRepository = orderRepository;
        _mailSender = mailSender;
    }

    public void Process(Order order) {
        if(order.IsValid()) {
            repository.Save(order);
            notificationSender.Send(order);
        }
    }
}
```

Здесь OrderManager зависит от абстракций, а не от конкретных реализаций. Можно без труда менять его поведение, внедряя нужную зависимость в момент создания экземпляра OrderManager.

2005 год: шестиугольная, порты и адаптеры

Алистер Кокберн

2005 год: шестиугольная

Основные характеристики:

a. Разработано Алистером Кокберном, также известно как шаблон «Порты и адаптеры».
b. Фокусируется на создании ядра приложения (логики домена), которое взаимодействует с внешним миром через четко определенные порты (интерфейсы).
c. Адаптеры реализуют эти порты для обработки определенных задач, таких как взаимодействие с базой данных или внешним API.
d. Поток коммуникации:
e. Логика ядра (внутри) взаимодействует с внешними системами (UI, базы данных, API) через порты.
f. Внешние системы взаимодействуют с ядром через адаптеры.

Плюсы:

Гибкость и адаптируемость к изменениям.

Поощряет тестируемость, поскольку ядро отделено от внешних зависимостей.

Хорошо подходит для микросервисов и распределенных систем.

Минусы:

Может добавить сложности из-за строгого разделения задач.

Требует тщательного проектирования портов и адаптеров.

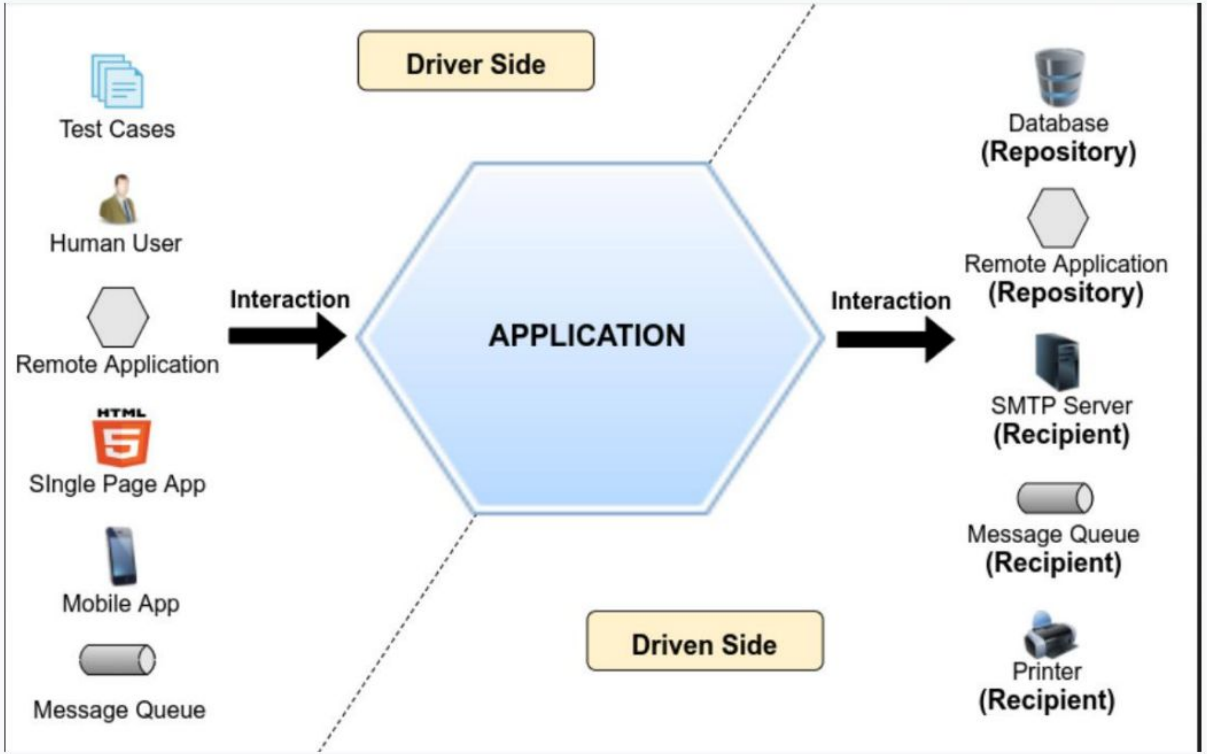
Гексагональная

Alister Cockburn

Цель – сделать ядро приложения независимым от технологий и платформ

Primary Actors (Drivers) – системы, взаимодействующие с приложением по своей инициативе (люди, приложения)

Secondary Actors (Driven) – системы, вызываемые приложением (базы данных, получатели сообщений)



Гексагональная

Порты

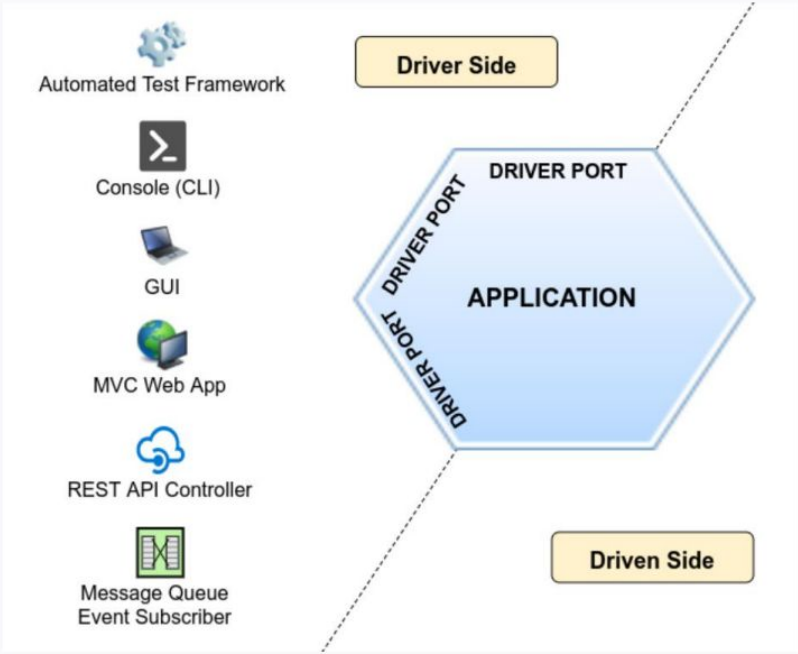
Порты

Порт – интерфейс для взаимодействий

- Диалоговый интерфейс взаимодействия с пользователем
- Интерфейс взаимодействия с БД
- и т.д.

Driver ports – интерфейсы для Primary Actors

Driven ports – интерфейсы для Secondary Actors



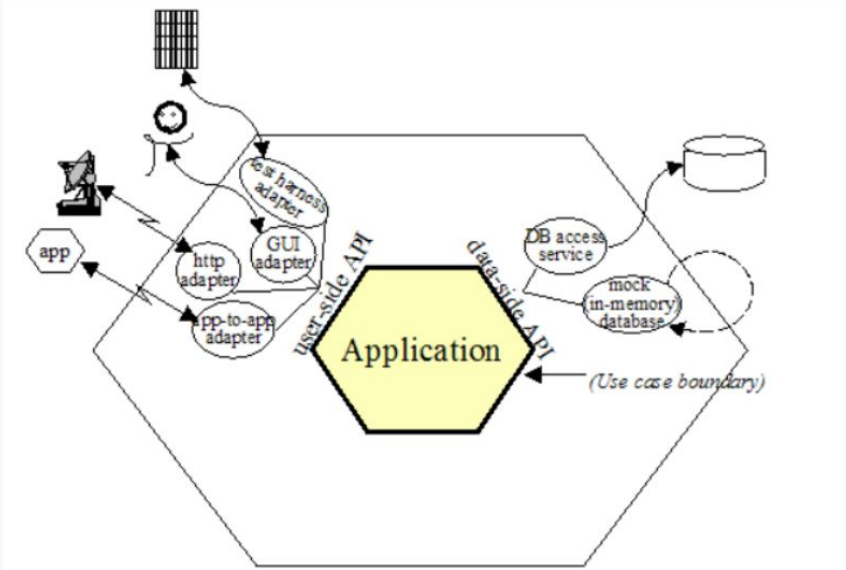
Гексагональная

Адаптеры

Адаптеры

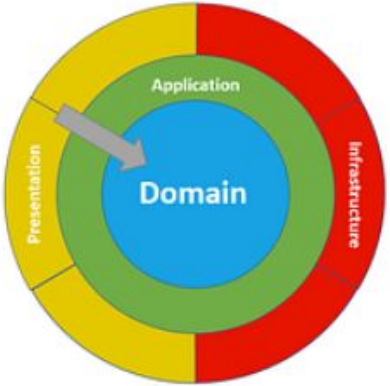
Адаптер - программный компонент, позволяющий некоторой технологии взаимодействовать с портом

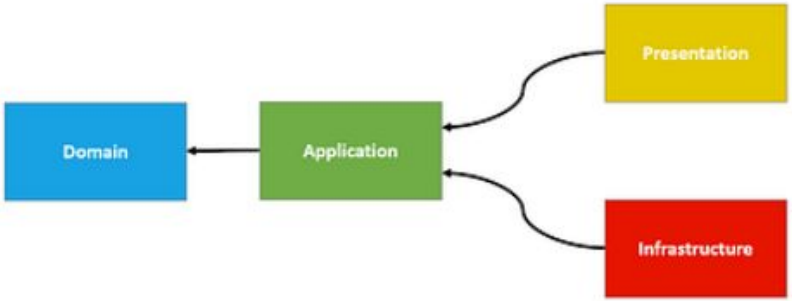
Находятся вне ядра приложения



2008 год: луковая

Джеффри Палермо





2008 год: луковая

Основные характеристики:

a. Сосредоточение вокруг доменной модели, размещение ее в ядре приложения.
b. Слои структурированы в концентрические круги:
c. Ядро: сущности домена и бизнес-логика (полностью независимы от внешних зависимостей).
d. Внутренние слои: службы приложений, использующие домен.
e. Внешние слои: инфраструктура и пользовательский интерфейс.
f. Зависимости текут внутрь — внешние слои зависят от внутренних, но не наоборот.

Плюсы:

Сильный акцент на доменно-ориентированном проектировании.

Базовая доменная логика полностью независима от фреймворков и инфраструктуры.

Высокая тестируемость, поскольку бизнес-логику можно тестировать, не полагаясь на внешние системы.

Минусы:

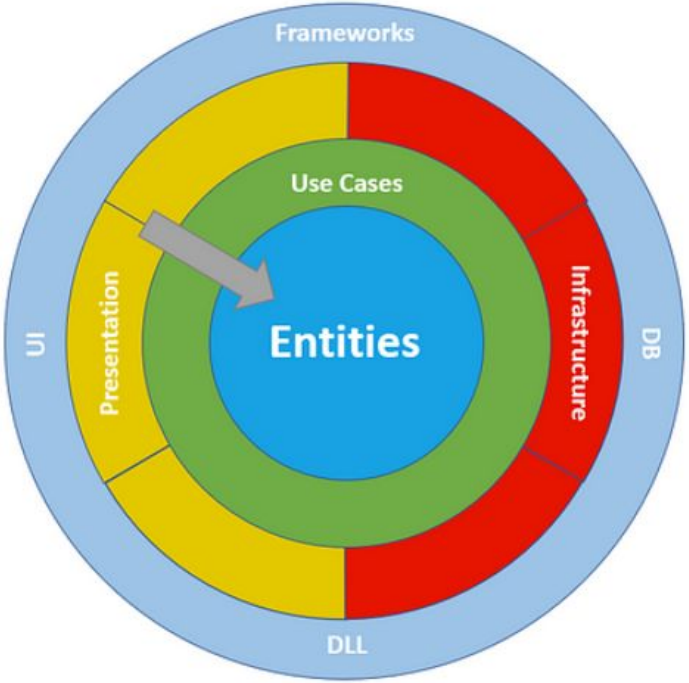
Более крутая кривая обучения по сравнению с многоуровневой архитектурой.

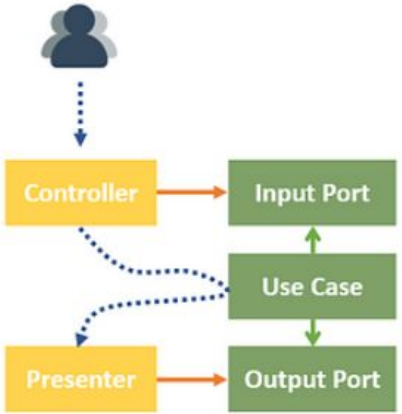
Первоначальная настройка может быть сложной.

Может показаться излишней разработкой для более простых проектов.

2012 год: чистая архитектура

Роберт С. Мартин





2012 год: чистая архитектура

Аналогично архитектуре лука, но с более четкими указаниями:

a. Сущности: представляют основные бизнес-правила.
b. Случаи использования (UseCase): бизнес-правила, специфичные для приложения.
c. Интерфейсы: шлюзы и адаптеры для внешних систем.
d. Фреймворки/драйверы: внешние зависимости, такие как пользовательский интерфейс, базы данных или API.
e. Зависимости указывают внутрь, а код в ядре ничего не знает о внешних слоях.

Плюсы:

Высокая модульность и удобство обслуживания.

Поощряет разделение задач.

Легко адаптируется к будущим изменениям (фреймворки или базы данных можно менять местами, не затрагивая основную логику).

Минусы:

Концептуально абстрактный и может показаться незнакомым традиционным разработчикам.

Требует дисциплинированного следования своим принципам.

Настройка и первоначальные усилия по разработке выше.

Слои и ссылки

Обязанности слоев

Уровни в архитектуре программного обеспечения представляют собой различные функциональные части приложения, каждый из которых несет определенную ответственность. Определение четких границ для этих слоев необходимо для обеспечения поддерживаемой, масштабируемой и тестируемой системы.

1. Почему важны четкие границы
   - Разделение задач: каждый слой должен быть сосредоточен на одном аспекте приложения, чтобы снизить сложность.
   - Поддерживаемость: изменения в одном слое (например, пользовательский интерфейс) не должны требовать изменений в другом (например, доступ к данным).
   - Масштабируемость: изолированные слои позволяют легче обновлять или заменять отдельные компоненты, не нарушая работу всей системы.
   - Тестируемость: изолированные слои упрощают модульное тестирование, поскольку каждый слой можно тестировать независимо от других.

Обязанности слоев

2. Типичные обязанности для общих слоев

Уровень представления:

- Обрабатывает логику пользовательского интерфейса (UI).
- Отображает данные и фиксирует вводимые пользователем данные.
- Не должен содержать бизнес-логику или логику базы данных.
- Пример: фронтенд HTML/JS, компоненты Angular или представления React.

Уровень бизнес-логики:

- Реализует основные бизнес-правила и рабочие процессы.
- Организует операции между несколькими сущностями и уровнями.
- Должен оставаться независимым от внешних систем или фреймворков.
- Пример: классы служб, доменная логика в чистой или луковой архитектуре.

Уровень доступа к данным:

- Отвечает за сохранение и извлечение данных.
- Должен изолировать специфику базы данных или механизма хранения.
- Пример: репозитории, контексты Entity Framework или запросы SQL.

Обязанности слоев

3. Минимизируйте зависимости

Сокращение ненужных зависимостей между уровнями помогает:

- Избегать тесной связи, которая может привести к волновым эффектам при изменении одного уровня.
- Улучшать возможность повторного использования, поскольку слабосвязанные компоненты легче интегрировать в другие проекты.

Связи между слоями

Слои не существуют изолированно; им необходимо взаимодействовать. Правильное структурирование этих связей обеспечивает чистый и эффективный поток данных между слоями, минимизируя при этом связанность.

1. **Внедрение зависимостей**
Определение: шаблон проектирования, в котором зависимости (например, службы, репозитории) предоставляются компоненту, а не создаются им.

Почему это важно:

- Способствует слабой связанности, позволяя компонентам зависеть от абстракций, а не от конкретных реализаций.

- Упрощает замену, расширение или тестирование компонентов.

Связи между слоями

2. **Разделение через интерфейсы**
Определение: Интерфейсы определяют контракты, которым должны следовать слои, что снижает прямую зависимость от конкретных реализаций.
Преимущества:

- Включает полиморфизм, позволяя слоям взаимодействовать с абстракциями, а не с конкретными классами.
- Облегчает имитацию и тестирование, заменяя реальные реализации тестовыми двойниками.

Best Practice

- Используйте контейнеры внедрения зависимостей:
- Соблюдайте принципы SOLID: Например, придерживайтесь принципа инверсии зависимостей (DIP), который гласит, что слои должны зависеть от абстракций, а не от конкретных классов.
- Избегайте циклических зависимостей: Слои должны иметь четкий направленный поток без циклов.
- Централизуйте логику связи: Используйте классы служб для управления связью, а не вызывайте методы напрямую между слоями.

Вопросы для проверки

Каковы основные обязанности уровня бизнес-логики в многоуровневой архитектуре?

Обрабатывает основную логику приложения, обрабатывает входные данные из уровня представления и взаимодействует с уровнем доступа к данным

Как внедрение зависимостей помогает минимизировать зависимости между уровнями?

оно разделяет уровни, внедряя зависимости во время выполнения, что повышает гибкость и упрощает тестирование.

Каковы потенциальные риски неправильного определения границ между уровнями?

повышенная связанность, снижение поддерживаемости, сложность масштабирования и более сложная отладка.

Интерфейсы и абстракции

Роль интерфейсов в архитектуре

- Интерфейс в архитектуре программного обеспечения — это контракт, который определяет, какие методы или свойства должен реализовать класс. Интерфейсы играют ключевую роль в разделении слоев и обеспечении гибкости и тестируемости в приложении
- Инкапсуляция деталей реализации: Слоям не нужно знать конкретные детали реализации классов, с которыми они взаимодействуют. Вместо этого они полагаются на контракт интерфейса. (*Пример: Уровню бизнес-логики (BLL) не нужно знать, хранятся ли данные в SQL Server, MongoDB или файле. Он взаимодействует только с интерфейсом репозитория.*)
- Гибкость: Реализация может меняться, не влияя на зависимые слои, пока контракт интерфейса остается согласованным.
- Тестируемость: Интерфейсы позволяют использовать фиктивные реализации во время модульного тестирования, изолируя тестируемый слой от его зависимостей.

Проектирование интерфейсов

- Единая ответственность: Сосредоточьте интерфейсы на одной задаче, чтобы следовать принципу единой ответственности (SRP). (*Пример: избегайте объединения не связанных между собой методов в одном интерфейсе (например, IUserRepository не должен включать методы, связанные с Order*).
- Определяйте интерфейсы для абстракций, а не конкретных деталей: Интерфейсы должны описывать поведение, а не то, как оно достигается.
- Избегайте больших, раздутых интерфейсов: Придерживайтесь принципа разделения интерфейсов (ISP): клиенты не должны зависеть от методов, которые они не используют.
- Используйте описательные имена: Интерфейсы должны четко указывать свое назначение (например, IRepository<T> или ILogger).
- Префикс «I» (необязательно, но широко распространенное соглашение):

Пример

Вопросы для проверки

Почему важно, чтобы уровень бизнес-логики зависел от интерфейса, а не от конкретной реализации уровня доступа к данным?

он обеспечивает гибкость при изменении реализаций без влияния на бизнес-логику и повышает тестируемость

Как интерфейс работает как контракт между уровнями?

он определяет методы, которые должен реализовать уровень, обеспечивая согласованную связь и поведение

Приведите пример интерфейса, который вы бы использовали в шаблоне репозитория, и объясните его назначение.

например, IBlogRepository с методами для операций CRUD; он абстрагирует доступ к базе данных

Интеграция API

Внутренние и внешние API

Внутренние API:

- Используются в приложении для обеспечения связи между частями.

Внешние API:

- Предоставлять функциональность внешним системам или использовать сторонние сервисы.
- Пример: Приложение погоды вызывает API внешнего сервиса погоды для получения данных.
- Цель: Обеспечить взаимодействие с другими системами, интеграциями или внешними клиентами.

REST (Representational State Transfer)

REST — это архитектурный стиль, используемый для разработки сетевых приложений, особенно веб-сервисов. Он опирается на связь без сохранения состояния и стандартные методы HTTP для взаимодействия с ресурсами, что делает его простым, масштабируемым и широко распространенным.

Основные принципы REST

1. Ресурсо-ориентированный: https://example.com/api/users/123 (представляет пользователя с идентификатором 123).
2. Связь без сохранения состояния: Каждый клиентский запрос содержит всю информацию, необходимую для его обработки. Сервер не сохраняет состояние клиента между запросами.
3. Методы HTTP для действий
4. Представление ресурсов (обычно JSON или XML).
5. Кэширование

Http vs gRPC

Функция	REST	gRPC
Протокол	HTTP/1.1	HTTP/2
Формат данных	JSON или XML	Protocol Buffers (Protobuf)
Скорость	Медленнее из-за разбора JSON	Быстрее благодаря бинарной сериализации
Подходящий сценарий	Простые операции CRUD	Взаимодействие в реальном времени, высокоскоростной обмен данными
Инструменты	Широкая поддержка в браузерах и HTTP-инструментах	Требуется поддержка Protobuf и gRPC
Стриминг	Ограниченный	Встроенный двунаправленный стриминг
Примеры	Публичные API, веб-сервисы	Внутреннее взаимодействие в микросервисах

Определения конечных точек

Проектирование на основе ресурсов:

1. Определяйте конечные точки на основе ресурсов, а не действий.
    - Пример:
    - Хорошо: GET /users/{id}
    - Плохо: GET /getUserById
2. Последовательность:
    - Используйте согласованные соглашения об именовании и методы HTTP:
    - GET для извлечения данных.
    - POST для создания ресурсов.
    - PUT для обновления ресурсов.
    - DELETE для удаления ресурсов.
3. Обработка ошибок:
    - Используйте соответствующие коды состояния HTTP (например, 200 OK, 404 Not Found, 500 Internal Server Error).
    - Предоставляйте осмысленные сообщения об ошибках в теле ответа.

Определения конечных точек

4. Безопасность:
    - Внедряйте механизмы аутентификации и авторизации (например, OAuth, JWT).
    - Используйте HTTPS для шифрования связи API.
5. Пагинация и фильтрация:
    - Поддержка пагинации для больших наборов данных.
    - Пример: GET /users?page=1&pageSize=10

Управление версиями API

API со временем развиваются. Управление версиями обеспечивает обратную совместимость для старых клиентов, одновременно внедряя новые функции.

Стратегии управления версиями:

- Управление версиями URI: GET /v1/users
- Управление версиями параметров запроса: GET /users?version=1
- Управление версиями заголовков: Accept: application/vnd.myapp.v1+json

Рекомендации:

- Всегда планируйте управление версиями с самого начала.
- Ясно сообщайте пользователям об устаревших версиях.

Пример

Вопросы для проверки

В чем основные различия между REST и gRPC с точки зрения использования и производительности?

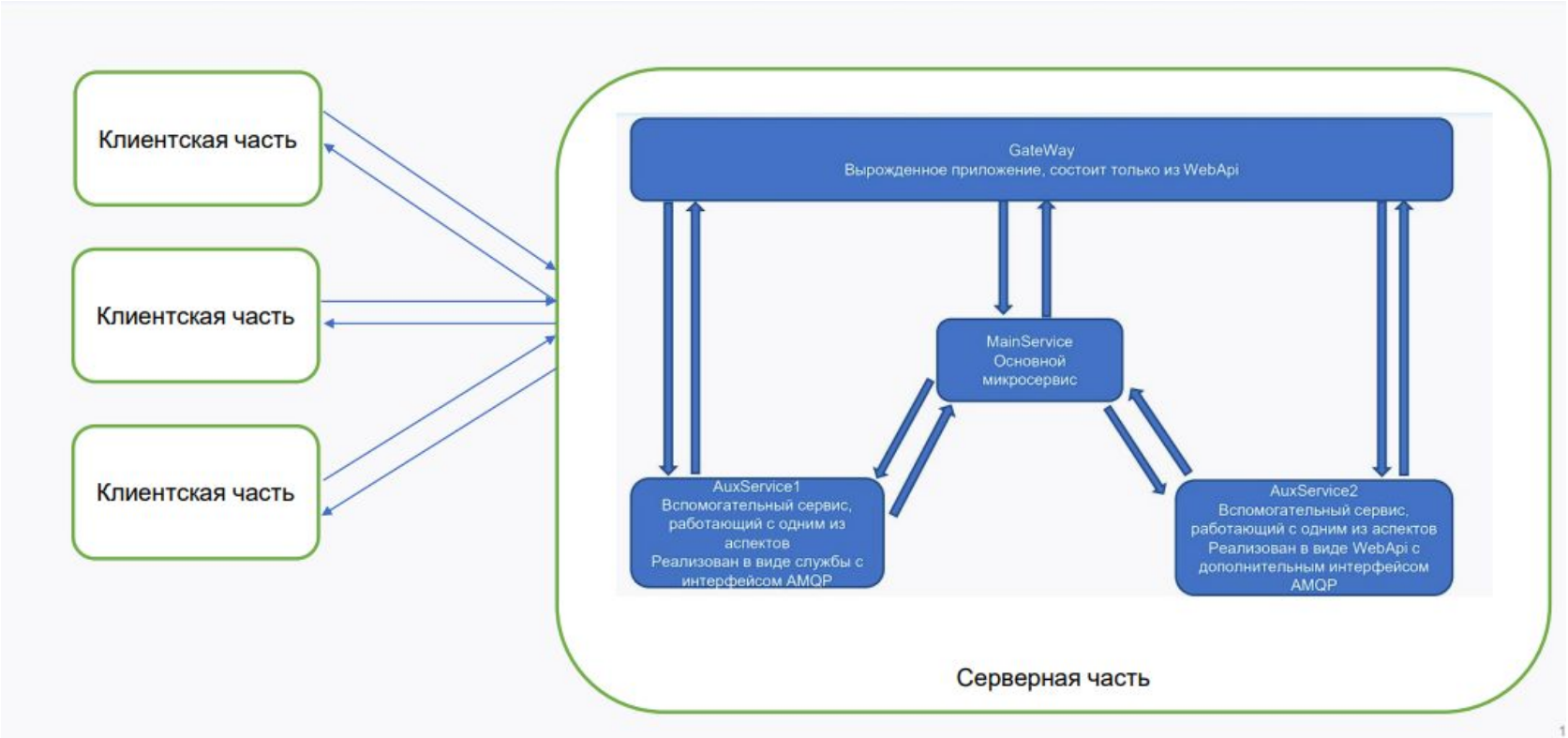
REST использует текстовую связь по HTTP и не зависит от языка, в то время как gRPC использует двоичную связь по HTTP/2, предлагая более высокую производительность для тесно связанных систем.

Какие рекомендации следует соблюдать при проектировании конечных точек API?

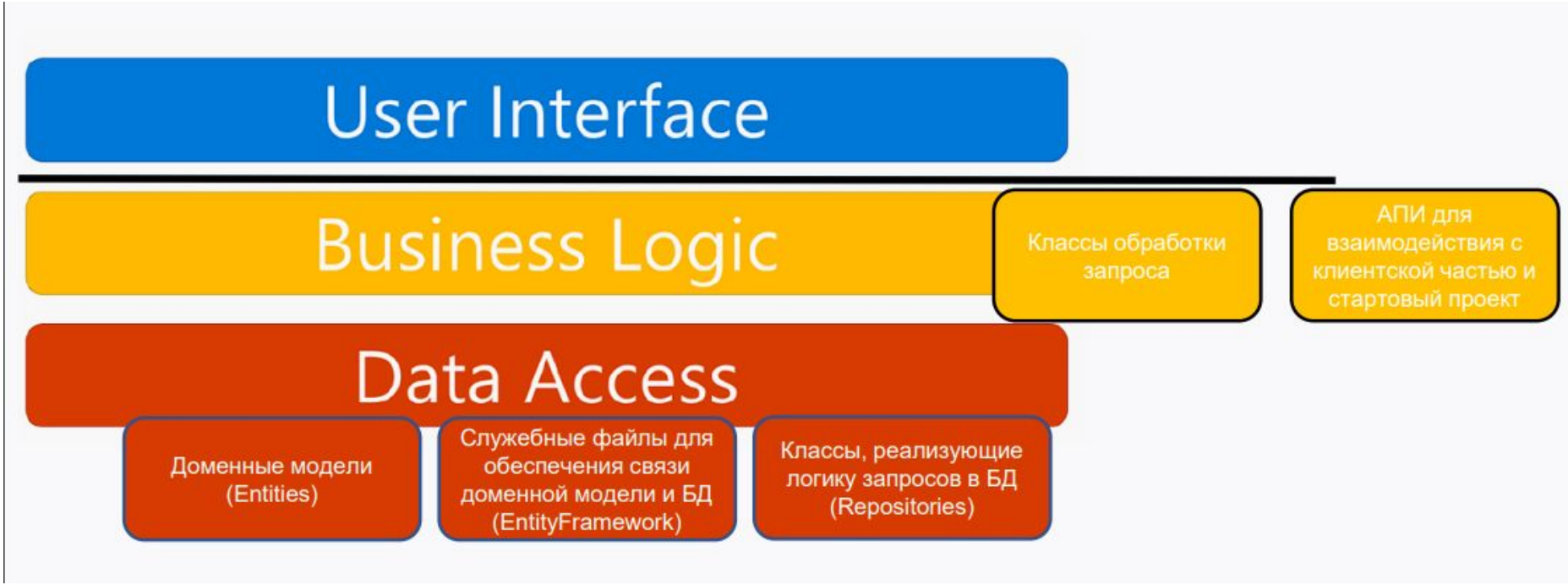
Используйте соглашения RESTful, понятные и описательные конечные точки, правильное управление версиями и единообразное именование.

Выбор структуры проекта

Пример частой реализации



Пример частой реализации



Задание

Разработайте приложение для блога, в котором пользователи смогут создавать, редактировать и просматривать сообщения в блоге. Приложение должно иметь аутентификацию пользователя и общедоступный API для получения сообщений.

Список материалов для изучения

1.Мартин Фаулер Архитектура корпоративных программных приложений;
2. Мартин Фаулер Рефакторинг. Улучшение существующего кода;
3. Роберт Мартин Чистая архитектура. Искусство разработки программного обеспечения;
4. https://web.archive.org/web/20180822100852/ http://alistair.cockburn.us/Hexagonal+architecture
5. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
6. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
7. https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/commonweb-application-architectures

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Рефлексия

Рефлексия

С какими впечатлениями уходите с вебинара?

Как будете применять на практике то,
что узнали на вебинаре?