Модульные монолиты

Модульные монолиты сочетают в себе модульность и четкие границы микросервисов с простотой эксплуатации монолитного развертывания. Такие организации, как Shopify, GitHub и Basecamp, успешно внедрили крупномасштабные системы, используя этот подход, добившись автономности команд и четких архитектурных границ, а также избежав сложности распределенных систем.

Что делает монолит модульным

Традиционный монолит

Единая кодовая база без четких внутренних границ. Компоненты тесно связаны и взаимодействуют напрямую через общие базы данных и методы.

- Слабое разделение ответственности
- Прямой доступ к данным
- Сложность изменений

Модульный монолит

Единое развертывание с четкими внутренними границами. Модули взаимодействуют через определенные интерфейсы и владеют своими данными.

- Четкие архитектурные границы
- Контролируемое взаимодействие
- Независимая разработка модулей

Разница между модульным монолитом и традиционным монолитом не заключается в способе развертывания — оба являются едиными целыми. Разница заключается в их внутренней организации и дисциплине, с которой поддерживаются границы.

Модули как первоклассные элементы

Управление пользователями

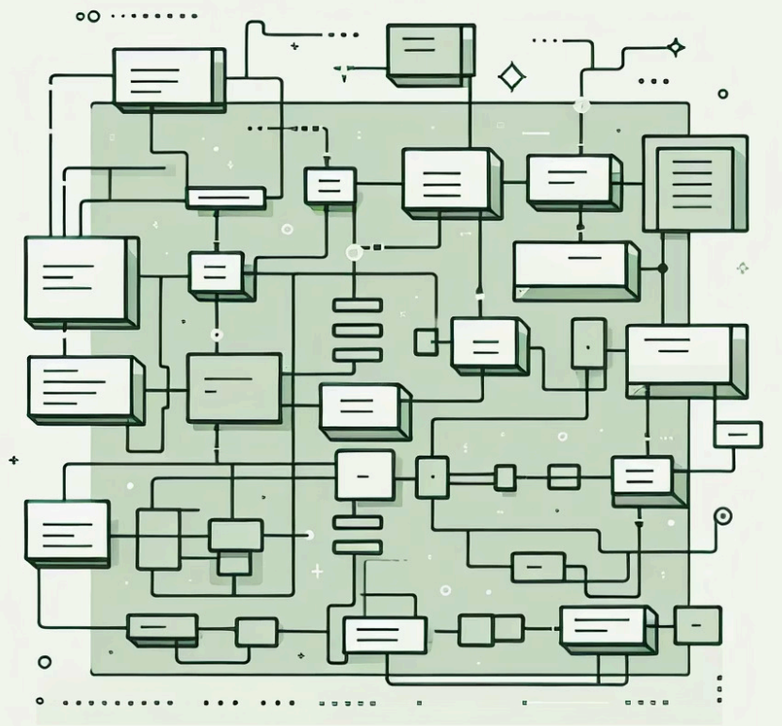
Отдельная бизнес-возможность со своей моделью данных, логикой и интерфейсом для аутентификации и профилей пользователей.

Обработка заказов

Самостоятельная единица для управления жизненным циклом заказов от создания до выполнения.

Отслеживание запасов

Независимый модуль для контроля товарных остатков и управления складскими операциями.



В модульном монолите **модули — это не просто папки или пакеты, а архитектурные границы**. Каждый модуль представляет собой отдельную бизнес-возможность со своей собственной моделью данных, бизнес-логикой и четким интерфейсом для остальной части системы.

Взаимодействие через интерфейсы

Эти модули взаимодействуют через **четко определенные интерфейсы**, а не через прямые вызовы методов или общий доступ к базе данных. Когда модуль заказов нуждается в информации о пользователе, он не запрашивает ее напрямую из базы данных пользователей, а вызывает API модуля пользователя.

> Этот интерфейс может быть реализован в виде вызовов методов внутри процесса для повышения производительности, но архитектурная граница остается четкой.

Принудительные границы

01

Дисциплина команды

Соблюдение архитектурных принципов и избежание соблазна прямого доступа к данным других модулей.

02

Инструменты проверки

Автоматическая валидация границ модулей на этапе компиляции с помощью специализированных фреймворков.

03

Код-ревью

Систематическая проверка соблюдения архитектурных границ в процессе разработки.

Критической задачей модульных монолитов является **принудительное соблюдение границ**. Без сетевых границ, которые естественным образом разделяют микросервисы, возникает соблазн пойти по пути упрощения — напрямую обратиться к базе данных другого модуля или вызвать внутренние методы.

Пример: Spring Modulith

Современные фреймворки предоставляют механизмы для обеспечения соблюдения этих границ. **Spring Modulith** предлагает проверку границ модулей на этапе компиляции, гарантируя, что модули взаимодействуют только через назначенные интерфейсы.

```
@Modulith
public class OrderModule {

    @ApplicationModuleListener
    void on(UserRegisteredEvent event) {
        // Обработка события из модуля пользователей
    }

    @Component
    class OrderService {
        // Публичный API модуля
    }
}
```

Как это работает?

1. **Обнаружение модулей**: Аннотация `@Modulith` указывает на корень модуля. По умолчанию Spring Modulith считает все подпакеты частью этого модуля.
2. **Проверка зависимостей**: На этапе выполнения тестов или сборки Spring Modulith автоматически проверяет все зависимости между модулями. Если он обнаруживает попытку доступа к непубличному компоненту (например, к классу с доступом `package-private`, как `InternalOrderProcessor`), сборка завершится с ошибкой.
3. **Событийная архитектура**: Использование `ApplicationEventPublisher` для публикации событий и `@ApplicationModuleListener` для их прослушивания является предпочтительным способом межмодульного взаимодействия. Это позволяет модулям оставаться полностью независимыми друг от друга.

Этот подход превращает архитектурные правила из рекомендаций, которые легко нарушить, в **автоматически проверяемые ограничения**. Это значительно упрощает поддержку чистоты кода в крупных проектах и снижает риск появления "спагетти-кода".

Доменно-ориентированное проектирование

Наиболее успешные модульные монолиты выравнивают границы модулей в соответствии с **принципами доменно-ориентированного проектирования**. Каждый модуль представляет собой ограниченный контекст — область, в которой последовательно применяются определенные бизнес-концепции, правила и терминология.

Почему это так эффективно?

Связка "модуль = ограниченный контекст" — это не просто теоретическое упражнение, а прагматичный подход, дающий реальные преимущества:

- **Автономность команд.** Команда, отвечающая за модуль `Payments` (Платежи), может развивать его, не вмешиваясь в работу команды модуля `Inventory` (Склад). У них свой код, свои модели и свой темп работы.
- **Единый язык (Ubiquitous Language).** Внутри модуля `Catalog` (Каталог) термин "Товар" (`Product`) означает одно (описание, фото, характеристики). А в модуле `Inventory` (Склад) тот же "Товар" — это совсем другая сущность (артикул, количество на складе, расположение). Разделение на модули позволяет избежать путаницы и двусмысленности в коде.
- **Низкая связанность и явные контракты.** Модули не должны "залезать" во внутренние дела друг друга. Их взаимодействие строится на четко определенных "контрактах": публичных API-методах или, что еще лучше, на асинхронных событиях. Например, модуль `Orders` (Заказы) не меняет остатки на складе напрямую, а публикует событие `OrderPlaced` (Заказ размещен), на которое реагирует модуль `Inventory`.

Различные представления данных

Модуль управления клиентами

- Полный профиль клиента
- История покупок
- Предпочтения и настройки
- Контактная информация
- Программы лояльности

Модуль доставки

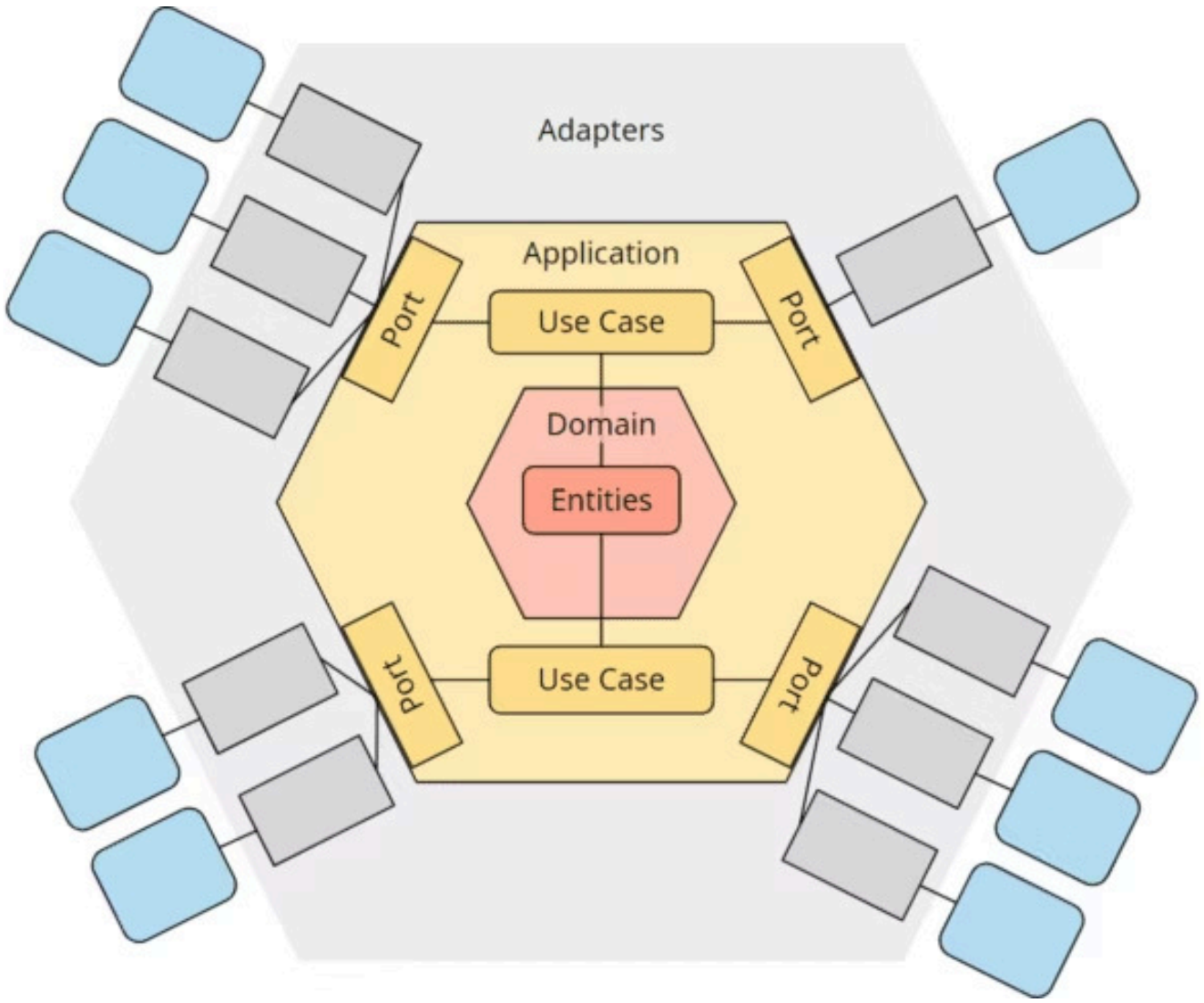
- Адрес доставки
- Предпочтения по доставке
- Временные окна
- Специальные инструкции

Каждый модуль имеет свою собственную модель основных концепций, таких как «клиент» или «продукт», оптимизированную для конкретного случая использования

Почему это так важно?

1. **Простота и фокус.** Модель в каждом модуле содержит **только то, что необходимо** для решения его задач. Разработчикам модуля доставки не нужно разбираться в сложной логике программ лояльности, что снижает когнитивную нагрузку и ускоряет разработку.
2. **Автономность и устойчивость к изменениям.** Представьте, что отдел маркетинга решил добавить в профиль клиента поле "Любимый цвет". Если бы у нас была одна общая модель клиента, это изменение потенциально затронуло бы все модули системы. В нашем случае оно коснется только модуля управления клиентами. Модуль доставки даже не заметит этих изменений, так как он работает со своей собственной, независимой моделью.
3. **Предотвращение "Божественного Объекта" (God Object).** Такой подход помогает избежать создания гигантского класса `Customer`, который знает абсолютно всё и используется повсюду. Подобные "божественные объекты" — главный источник головной боли в больших проектах, так как любое изменение в них рискует сломать что-то в самом неожиданном месте..

Hexagonal Architecture



Каждый модуль может быть структурирован с использованием **гексогональной архитектуры** (порты и адаптеры) с четким разделением бизнес-логики и внешних задач.

- **Ядро:** "Чистая" бизнес-логика, не знающая о базах данных или веб-фреймворках.
- **Порты:** Интерфейсы, определяющие, как можно взаимодействовать с ядром.
- **Адаптеры:** Конкретные реализации для технологий — REST-контроллеры, репозитории для работы с базой данных, клиенты для очередей сообщений.

Эта структура делает ядро модуля чрезвычайно легко тестируемым и позволяет менять технологический стек (например, перейти с PostgreSQL на MongoDB), не затрагивая бизнес-логику.

Бизнес-логика

Основная логика модуля находится в центре, изолированная от внешних зависимостей.

Порты

Интерфейсы для коммуникации с другими модулями, базами данных и внешними системами.

Адаптеры

Конкретные реализации портов для различных технологий и протоколов.

Событийно-ориентированная коммуникация

Модуль заказов

Публикует событие "OrderCompleted" при завершении заказа

Модуль инвентаря

Обновляет остатки товаров при получении события

Модуль уведомлений

Отправляет подтверждение клиенту

Модуль аналитики

Обновляет метрики продаж

Даже в рамках одного процесса модули могут взаимодействовать через **события**, а не прямые вызовы. Этот событийно-ориентированный подход обеспечивает те же преимущества слабой связности, что и микросервисы, сохраняя при этом простоту внутрипроцессной коммуникации.

Общее ядро и антикоррупционные слои

Shared Kernel (Общее ядро)

Общие абстракции, которые законно охватывают несколько модулей:

- Идентичность пользователя
- Общие объекты значений
- Базовые бизнес-правила

Anti-Corruption Layer

Защитный слой для взаимодействия с внешними системами:

- Перевод между моделями
- Изоляция от устаревших систем
- Защита от внешних изменений

Spring Modulith: возможности фреймворка

Проверка границ

Автоматическая валидация соблюдения архитектурных границ на этапе компиляции с генерацией ошибок при нарушениях.

Генерация документации

Автоматическое создание диаграмм и документации о взаимосвязях между модулями системы.

Изолированное тестирование

Возможность тестирования отдельных модулей в изоляции от остальной системы.

Spring Modulith привносит в экосистему Spring первоклассную поддержку модульных монолитов, позволяя разработчикам определять модули как Java-пакеты с явными публичными API.

Подробнее о Spring Modulith **тут**

.NET и модульный дизайн

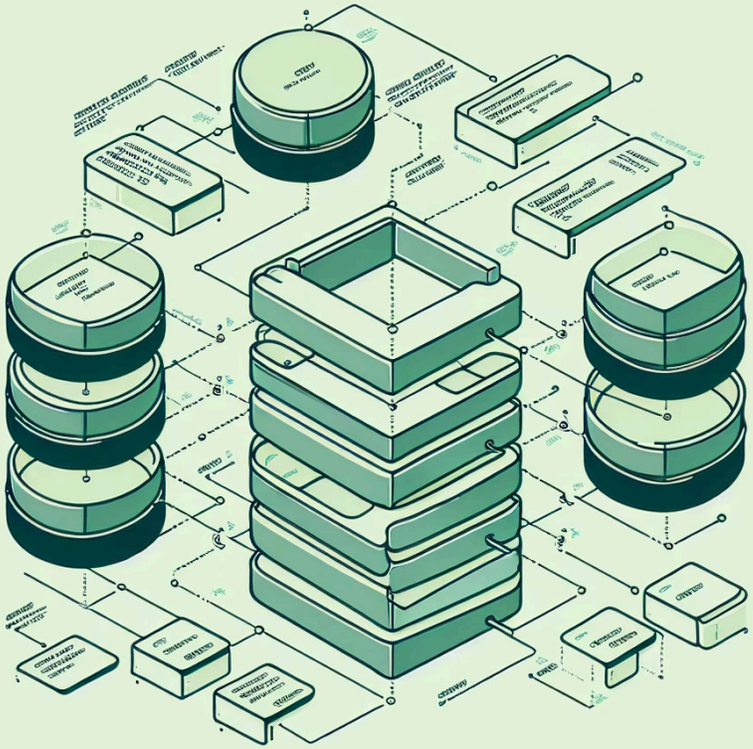
Встроенные возможности

- Сборки и пространства имен
- Модификаторы внутренней доступности
- Контроль доступа на уровне сборки

Дополнительные инструменты

- **Wolverine** - событийно-ориентированная коммуникация
- **MediatR** - паттерн медиатор для слабой связности
- Высокая производительность внутри процесса

Экосистема **.NET** уже давно поддерживает модульный дизайн с помощью сборок и пространств имен. Современный .NET предоставляет дополнительные инструменты для поддержания границ модулей.



Шаблоны модулей баз данных

Схема для каждого модуля

Каждый модуль имеет свою схему базы данных. Отношения внешних ключей существуют только в пределах модуля.

Представления для интеграции

Представления только для чтения обеспечивают контролируемый доступ к данным между модулями для отчетности.

Исходные события

События служат точками интеграции и контрольным журналом для обмена изменениями состояния.

Одним из самых сложных аспектов модульных монолитных систем является управление данными. Несколько шаблонов помогают сохранить границы данных при использовании общей инфраструктуры.

От монолита к модульному монолиту

1. **Анализ границ**

   Определение естественных швов в существующем коде и областей с минимальной связью

2. **Паттерн Strangler ("душитель")**

   Постепенное извлечение функциональности в четко определенные модули

3. **Установка интерфейсов**

   Создание четких API между модулями и устранение прямых зависимостей

4. **Рефакторинг данных**

   Разделение общих баз данных и установка границ владения данными

Путь от традиционного монолита к модульному монолиту обычно начинается с **определения границ домена** в существующем кодовом базе. Этот процесс часто показывает, что система уже имеет естественные швы.

Анализ границ

Первый и самый важный шаг — это **анализ границ** (Boundary Analysis) внутри существующего монолита. Цель — найти «естественные швы», то есть логические разделения в коде, которые соответствуют различным бизнес-возможностям или доменам.

- **Определение естественных швов**: Анализируйте код на предмет областей с минимальной связностью (**low coupling**) и высокой сплоченностью (**high cohesion**). Например, код, отвечающий за управление пользователями, скорее всего, слабо связан с кодом управления счетами. Такие части системы являются главными кандидатами на выделение в модули.
- **Инструменты и подходы**: Для выявления границ можно использовать анализ графов зависимостей между классами и компонентами, изучение потоков данных и бизнес-процессов, а также общение с экспертами в предметной области. Часто структура команд разработчиков уже отражает неявные границы в системе.

Паттерн Strangler ("Душитель")

После определения границ модулей можно приступать к их постепенному извлечению. **Паттерн "Душитель" (Strangler Fig Pattern)** — это эффективная и низкорисковая стратегия для такой миграции. Название происходит от фигового дерева-душителя, которое обвивает другое дерево и со временем полностью его замещает.

1. **Создание фасада**: Сначала создается фасад, который перехватывает вызовы к старой функциональности, предназначенной для извлечения. Весь внешний трафик теперь проходит через него.
2. **Постепенная замена**: Новая функциональность реализуется в виде отдельного, изолированного модуля. Фасад начинает перенаправлять соответствующие вызовы на новый модуль вместо старой системы.
3. **"Удушение" монолита**: Шаг за шагом, все больше функций переносится в новые модули, пока от старой реализации ничего не останется. В этот момент старый код можно безопасно удалить.

Этот подход позволяет проводить рефакторинг постепенно, без необходимости останавливать работу всей системы и рисковать «большим взрывом» при запуске.

Установка интерфейсов

Ключевое отличие модульного монолита от традиционного — это **строго определенные и принудительные границы** между модулями. Просто разложить код по папкам недостаточно.

- **Создание четких API**: Каждый модуль должен предоставлять публичный API (Application Programming Interface), через который с ним могут взаимодействовать другие модули. Любое прямое обращение к внутренним компонентам модуля (например, вызов его внутренних функций или прямой доступ к его таблицам в базе данных) должно быть запрещено.
- **Устранение прямых зависимостей**: Вместо прямых вызовов методов используются вызовы через интерфейсы. Это могут быть как простые интерфейсы на уровне языка программирования, так и более сложные механизмы, например, внутренняя шина событий (Event Bus), которая позволяет модулям общаться асинхронно и еще больше снижает их связанность.

Рефакторинг данных

Часто самым сложным этапом является работа с данными, поскольку в классическом монолите все компоненты используют одну общую базу данных.

- **Разделение общих баз данных**: Цель — прийти к состоянию, когда **каждый модуль владеет своими данными** и отвечает за их целостность. Другие модули могут получать доступ к этим данным только через API владельца.
- **Стратегии разделения**: Начать можно с логического разделения: определить, какие таблицы к какому модулю относятся. Затем можно постепенно переходить к физическому разделению. Это может потребовать дублирования данных или введения представлений (views) в базе данных на переходный период.
- **Установка границ владения данными**: Четкое определение владельца данных устраняет конфликты и упрощает внесение изменений. Если модулю А нужны данные из модуля Б, он должен запросить их через API модуля Б, а не обращаться напрямую к его таблицам.

Переход к модульному монолиту — это инвестиция в будущее проекта, которая окупается за счет упрощения поддержки, повышения надежности и ускорения разработки.

Извлечение модулей в микросервисы

Когда рост организации или технические требования оправдывают операционную сложность микросервисов, хорошо спроектированные модульные монолиты делают переход более плавным. Каждый модуль потенциально может стать независимым сервисом с установленными границами и моделями коммуникации.

После того как модуль становится отдельным сервисом, ему необходимо общаться с оставшейся частью монолита и другими сервисами. Для этого используются синхронные (например, REST API, gRPC) или асинхронные (через брокеры сообщений, такие как RabbitMQ или Kafka) паттерны коммуникации. Выбор зависит от требований к надежности и согласованности данных.

> **Модульные монолиты служат отличной площадкой для микросервисов**. Команды могут экспериментировать с границами сервисов, совершенствовать интерфейсы и наращивать операционные возможности, сохраняя при этом простоту монолитного развертывания.

Возвращение к модульным монолитам

01

Анализ сложности

Оценка операционных расходов распределенной архитектуры и выявление проблемных областей

02

Планирование консолидации

Определение модулей для объединения с сохранением четких архитектурных границ

03

Миграция данных

Объединение баз данных микросервисов в модульную структуру с сохранением целостности

04

Упрощение операций

Сокращение операционных расходов при сохранении независимости команд

Как мы видели на примере таких компаний, как Prime Video и Segment, иногда путь микросервисов ведет обратно к консолидированным архитектурам. Модульные монолиты являются отличной целью для такой консолидации.

Преимущества модульных монолитов

Простота эксплуатации

Единая единица развертывания означает упрощенные конвейеры CI/CD, более простые откаты и снижение накладных расходов на координацию.

Высокая производительность

Внутрипроцессная коммуникация устраняет сетевую задержку и накладные расходы на сериализацию.

Автономность команд

Четкие границы ответственности позволяют командам работать независимо над своими модулями.

Низкая когнитивная нагрузка

Разработчикам нужно понимать интерфейсы модулей, а не сложности распределенных систем.

Проблемы и ограничения

Дисциплина границ

Самая большая проблема — сохранение архитектурных границ без физических механизмов принудительного выполнения. Накопление технического долга может разрушить границы.

Ограниченная автономия

В отличие от микросервисов, команды ограничены общими решениями по инфраструктуре, стеку технологий и стратегиям развертывания.

Ограничения масштабирования

Все модули используют одну среду выполнения и пространство памяти. Для крупных систем эти ограничения могут стать сдерживающими факторами.

Сложность эволюции

Крупномасштабная рефакторинг между модулями может быть сложной задачей, особенно при изменении бизнес-требований.

Заключение

1	100%	0
Развертывание	Модульность	Сетевые вызовы
Единая точка развертывания упрощает операции	Четкие границы между бизнес-доменами	Внутрипроцессная коммуникация для высокой производительности

Модульные монолиты обеспечивают преимущества модульной архитектуры, сохраняя при этом простоту развертывания. Этот подход хорошо подходит для организаций, которым требуются четкие границы без операционной сложности микросервисов.

Архитектура эффективно работает, когда масштаб организации не оправдывает накладные расходы на распределенные системы, хотя она требует дисциплинированного поддержания границ и может ограничивать выбор технологий команды. Модульные монолиты могут служить либо долгосрочным архитектурным решением, либо эволюционным шагом к микросервисам при изменении требований к масштабируемости.