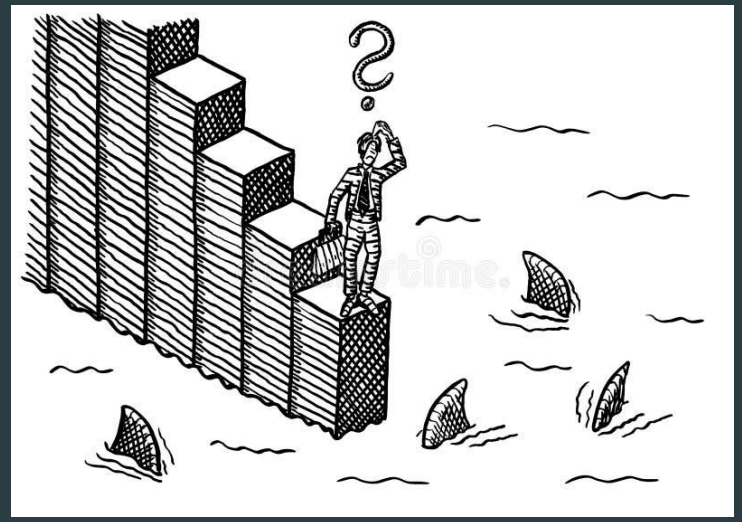
Адаптация бизнес приложений (типовых решений) с сохранением поддержки

Выполнил: Швецов Руслан

Почему это важно?

- Адаптивность вашего бизнес приложения закладывается с первого дня разработки. Это фундамент, который определяет сможет ли ваша система расти вместе с бизнесом или станет тормозом для развития
- Адаптация приложения это шаг в безопасность и экономию, без должной адаптации затраты на улучшение будут расти как со стороны бюджета на улучшение так и рабочего времени сотрудников задействованных в обновлении приложения



Рассматриваемые методы

- Всего в данной презентации будет рассмотрено 6 методов адаптации среди которых:
- 1. **Configuration Over Customization -** конфигурирование вместо кастомизации.
- 2. **Extension Points -** точка расширения
- 3. **Separation of Concerns(SoC) -** разделение ответственности
- 4. **API-First Integration**
- 5. **Database Abstraction -** детали внутренней реализации СУБД скрываются от разработчика
- 6. **Event-Driven Architecture (EDA)** – событийно-ориентированная архитектура

1. Configuration Over Customization

- В данном методе, предпочтение отдаётся параметризации и конфигурирования встроенными средствами системы вместо разработки пользовательских модификаций.

Аспект	Конфигурирование (Configuration)	Кастомизация (Customization)
Определение	Настройка системы через штатные инструменты без изменения кода	Изменение исходного кода системы для добавления функциональности
Подход	Использование встроенных возможностей и параметров	Программирование новых функций и модулей
Навыки	Бизнес-аналитики, администраторы, power-users	Разработчики, программисты, software engineers
Стоимость владения	Низкая, предсказуемая	Высокая, часто непредсказуемая
Влияние на обновления	Нулевое или минимальное	Критическое, требует переработки

Плюсы и минусы:

- **Плюсы Configuration:**

1.Мгновенное применение – изменения применяютсябез разработки и деплоя

2.Гарантированная поддержка – разработчик несет ответственность за работу функций

3.Низкая стоимость владения – не требует программистов для поддержки

Минусы Configuration:

1.Ограниченная гибкость – можно использовать только то, что предусмотрел разработчик

2.Зависимость от разработчика – ждем нужный функционал в следующих версиях

3.Стандартизированные процессы – сложно реализовать уникальные бизнес-процессы

2. Extension Points

- Использование предусмотренных разработчиком интерфейсов для подключения дополнительной функциональности без модификации исходного кода системы.

Аспект	Точки расширения (Extension Points)	Прямая модификация (Direct Modification)
Определение	Использование официальных интерфейсов для подключения доп. функциональности	Прямое изменение исходного кода ядра системы
Подход	Подключение через плагины, хуки, API	Переписывание существующих методов и классов
Изоляция	Полная изоляция кастомного кода	Полное смешение кастомного и стандартного кода
Совместимость	Сохраняется при обновлениях	Нарушается при каждом обновлении

Плюсы и минусы:

- **Плюсы Extension Points:**
- **Сохранение обновляемости** – можно безопасно обновлять ядро системы
- **Изоляция рисков** – ошибки в расширениях не ломают основную систему
- **Экосистема решений** – возможность использовать готовые плагины сообщества

- **Минусы Extension Points:**
- **Сложность разработки** – требуется глубокое понимание архитектуры
- **Ограничения фреймворка** – можно делать только то, что позволяет SDK
- **Производительность** – дополнительные слои абстракции могут замедлять работу

3. Separation of Concerns

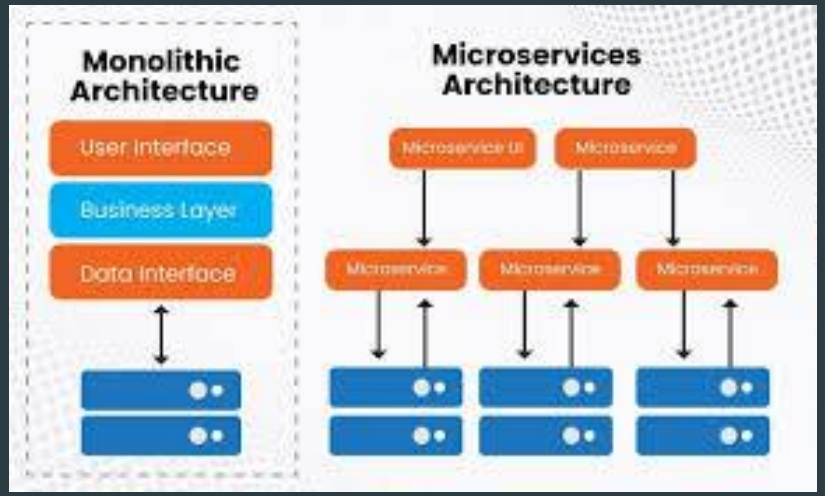
- Четкое разделение системы на независимые компоненты с определенными зонами ответственности.

Аспект	Разделение ответственности (SoC)	Монолитный подход (Monolithic)
Архитектура	Модульная, слоистая, микросервисы	Единая, tightly-coupled система
Изменения	Локализованы в отдельных модулях	Распространяются на всю систему
Тестирование	Независимое тестирование компонентов	Сложное интеграционное тестирование
Разработка	Параллельная работа разных команд	Последовательная разработка
Масштабируемость	Горизонтальное и вертикальное масштабирование	Ограниченное вертикальное масштабирование

Плюсы и минусы:

- **Плюсы Separation of Concerns:**
- **Независимая разработка** – команды могут работать параллельно
- **Упрощенное тестирование** – каждый компонент тестируется изолированно
- **Гибкость технологий** – можно использовать разные технологии для разных модулей
- **Легкий рефакторинг** – изменения в одном модуле не затрагивают другие

- **Минусы Separation of Concerns:**
- **Сложность координации** – нужно синхронизировать работу между командами
- **Нагрузка на инфраструктуру** – больше серверов, сетевых взаимодействий
- **Сложность отладки** – трассировка запросов через множество компонентов



4. API-First Integration

- Организация взаимодействия между системами исключительно через четко определенные программные интерфейсы.

Аспект	API-First интеграция	Прямой доступ к БД
Связность	Слабая связность, интерфейсное взаимодействие	Сильная связность, прямое зависимость
Безопасность	Контролируемый доступ, авторизация	Прямой доступ к данным, риски утечек
Эволюция	Независимое развитие систем	Синхронные изменения всех систем
Производительность	Оптимизированная нагрузка через кэширование	Прямая нагрузка на БД
Мониторинг	Централизованный мониторинг вызовов	Распределенный мониторинг запросов

Плюсы и минусы:

- **Плюсы API-First Integration:**
- **Безопасность** – контролируемый доступ вместо прямых манипуляций с данными
- **Независимость эволюции** – системы могут развиваться независимо
- **Кэширование и оптимизация** – возможность кэшировать частые запросы
- **Мониторинг и аналитика** – четкое отслеживание всех взаимодействий

- **Минусы API-First Integration:**
- **Задержки производительности** – дополнительные сетевые вызовы
- **Сложность реализации** – нужно проектировать и поддерживать API
- **Единая точка отказа** – проблемы с API затрагивают все интегрированные системы

5. Database Abstraction

- Изоляция бизнес-логики от конкретной реализации хранения данных через абстракции.

Аспект	Абстракция БД (Database Abstraction)	Прямые SQL-запросы
Подход	Работа через ORM, представления, хранимые процедуры	Написание нативных SQL-запросов
Переносимость	Независимость от конкретной СУБД	Привязка к конкретной СУБД
Безопасность	Защита от SQL-инъекций, параметризованные запросы	Риски SQL-инъекций, ручное экранирование
Рефакторинг	Легкий рефакторинг через миграции	Сложный рефакторинг при изменении схемы
Производительность	Оптимизация через кэширование и индексы	Прямое управление производительностью

Плюсы и минусы:

- **Плюсы Database Abstraction:**
- **Независимость от БД** – возможность сменить СУБД без переписывания кода
- **Безопасность** – защита от SQL-инъекций "из коробки"
- **Производительность** – автоматическое кэширование и оптимизация запросов
- **Упрощенная разработка** – работа с объектами вместо SQL

- **Минусы Database Abstraction:**
- **Потеря контроля** – ограниченные возможности тонкой оптимизации
- **Накладные расходы** – дополнительные преобразования данных
- **Сложные запросы** – некоторые сложные SQL сложно выразить через ORM

Пример использования

```
public DataTable GetSalesReport(DateTime startDate, DateTime endDate)
{
    var sql = @"
        SELECT u.name, u.email, SUM(o.total) as total, COUNT(o.id) as order_count
        FROM users u
        JOIN orders o ON u.id = o.user_id
        WHERE o.created_at BETWEEN @start AND @end
        GROUP BY u.id, u.name, u.email
        HAVING COUNT(o.id) > 0";

    using var connection = new MySqlConnection(_connectionString);
    return connection.ExecuteReader(sql, new { start = startDate, end = endDate });
}
```

- Было решено рассмотреть метод адаптации **Database Abstraction поскольку он вкладывается с самого начала разработки БД** и используется на протяжении всей разработки.

```
{
    return await _context.Users
        .Join(_context.Orders,
              u => u.Id,
              o => o.UserId,
              (u, o) => new { User = u, Order = o })
        .Where(x => x.Order.CreatedAt >= startDate && x.Order.CreatedAt <= endDate)
        .GroupBy(x => new { x.User.Id, x.User.Name, x.User.Email })
        .Select(g => new SalesReportItem
        {
            UserName = g.Key.Name,
            Email = g.Key.Email,
            TotalAmount = g.Sum(x => x.Order.Total),
            OrderCount = g.Count()
        })
        .Where(x => x.OrderCount > 0)
        .ToListAsync();
}
```

6. Event-Driven Architecture

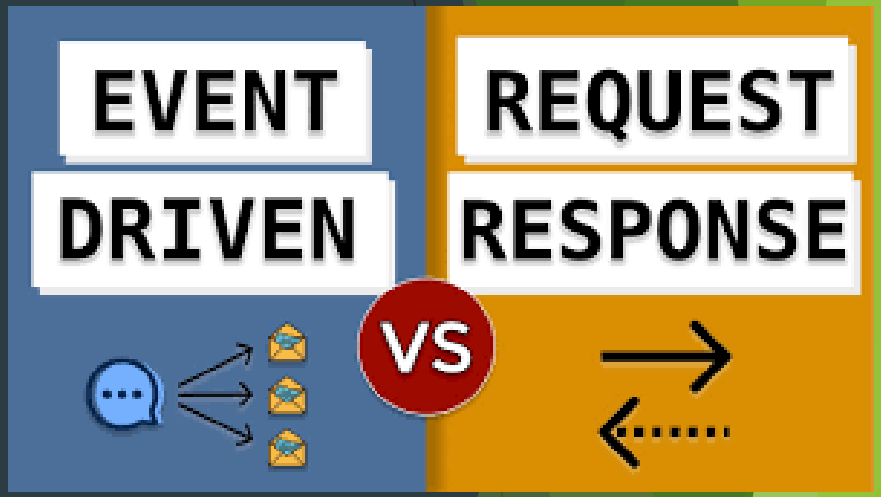
- Организация системы как набора независимых компонентов, реагирующих на события, вместо монолитных процессов.

Аспект	Событийная архитектура (EDA)	Запрос-ответ (Request-Response)
Коммуникация	Асинхронная, через события	Синхронная, через прямые вызовы
Связность	Полная декомпозиция систем	Сильная связность между системами
Масштабируемость	Высокая, горизонтальное масштабирование	Ограниченная, вертикальное масштабирование
Отказоустойчивость	Высокая, graceful degradation	Низкая, single point of failure
Сложность	Высокая начальная сложность	Низкая начальная сложность

Плюсы и минусы:

- **Плюсы Event-Driven Architecture:**
- **Высокая масштабируемость** – легко добавлять обработчики событий
- **Отказоустойчивость** – падение одного обработчика не ломает всю систему
- **Асинхронность** – неблокирующая обработка длительных операций
- **Гибкость** – легко добавлять новые реакции на события

- **Минусы Event-Driven Architecture:**
- **Сложность отладки** – трудно отслеживать цепочки событий
- **Потенциальная потеря данных** – если событие не обработано
- **Сложность обеспечения порядка** – гарантия последовательной обработки событий
- **Избыточность для простых систем** – unnecessary complexity для малых проектов



Итог:

Предусмотреть используемые для программного решения методы адаптации следует до начала разработки поскольку внедрение каждого из них в готовый проект приведёт к изменению кода основного функционала что негативно отразится на времени и стоимости разработки

Каждый из описанных методов не является взаимоисключающим чем больше методов используется в разработке приложения тем проще будет его обновлять и дополнять

Спасибо за внимание