# Эволюция Архитектурных Паттернов и Стратегии Обеспечения Качества в Экосистеме Enterprise.NET: Теоретический Базис Пятого Дня

Версия документа: 1.0

Контекст: Программа трансформации Unity Developer ->.NET Backend Architect

Тема: Неделя 8, День 5 — Продвинутые паттерны, гибридный доступ к данным (Dapper) и интеграционное тестирование

Уровень допуска: Senior Engineering / Architect

## 1. Введение: Архитектурный Рубикон Восьмой Недели

Завершение восьмой недели обучения знаменует собой не просто окончание знакомства с синтаксисом или библиотеками, но фундаментальный сдвиг в инженерном мышлении. Для специалиста, чей опыт сформирован в среде GameDev (Unity), где архитектура часто диктуется жизненным циклом кадра (Frame) и прямым доступом к состоянию в оперативной памяти (Stateful), переход к Enterprise-разработке требует радикальной перестройки ментальных моделей. Пятый день этой недели посвящен кристаллизации этих знаний в устойчивые архитектурные формы. Мы отходим от вопроса «Как написать запрос?» к вопросам «Как организовать систему, чтобы она пережила смену требований?» и «Как гарантировать, что база данных ведет себя предсказуемо в условиях высокой конкурентности?».

В данном аналитическом отчете мы проведем глубокую деконструкцию трех столпов современного бэкенда: спора об уровне абстракции над данными (Repository vs Direct DbContext), симбиоза ORM для производительности (EF Core + Dapper) и культуры автоматизированного тестирования инфраструктуры (Testcontainers). Этот документ служит теоретическим фундаментом для выполнения финальных заданий недели и призван сформировать у разработчика интуитивное понимание компромиссов (trade-offs), неизбежных при проектировании распределенных систем.1

## 2. Кризис Абстракций: Репозитории, Unit of Work и Вертикальные Слайсы

Центральной дилеммой при проектировании слоя доступа к данным в.NET остается выбор правильного уровня абстракции. Исторически сложившиеся практики, пришедшие из эры до-DDD (Domain-Driven Design) и ранних версий Entity Framework, часто вступают в конфликт с современными возможностями фреймворков и методологиями разработки.

### 2.1. Анатомия Паттерна Repository: Исторический Контекст и Современная Критика

Классический паттерн Repository, популяризированный Мартином Фаулером, был призван создать иллюзию того, что хранилище данных — это коллекция в памяти (in-memory collection). Это позволяло скрыть от бизнес-логики детали реализации SQL, ADO.NET и управления соединениями. Однако, с появлением мощных ORM, таких как Entity Framework Core, роль этого паттерна стала предметом ожесточенных дискуссий.2

#### 2.1.1. Тавтология Абстракции: DbContext как Unit of Work

Критический аргумент против создания собственных репозиториев (например, IUserRepository) поверх EF Core заключается в том, что EF Core сам по себе уже реализует паттерны Repository и Unit of Work.

* **DbSet как Repository:** Класс DbSet<T> предоставляет методы Add, Remove, Find и, что наиболее важно, реализует IQueryable<T>, позволяя строить запросы. Он ведет себя как коллекция, но транслирует операции в SQL.
* **DbContext как Unit of Work:** Класс DbContext отслеживает изменения во всех загруженных сущностях через механизм Change Tracker и атомарно персистирует их в транзакции при вызове SaveChangesAsync().

Создание дополнительного слоя абстракции вида:

C#

public interface IGenericRepository<T> {  
 Task<T> GetByIdAsync(int id);  
 void Add(T entity);  
}

часто приводит к анти-паттерну «Обертка ради обертки». Это лишает разработчика доступа к специфическим возможностям EF Core, таким как TagWith (маркировка SQL-запросов для отладки), SplitQuery (оптимизация декартовых произведений при JOIN) или глобальным фильтрам запросов, если только эти методы не дублируются в интерфейсе репозитория, что нарушает принцип DRY (Don't Repeat Yourself).2

#### 2.1.2. Проблема IQueryable и «Дырявые» Абстракции

Сторонники репозиториев часто аргументируют их использование необходимостью инкапсуляции логики запросов. Однако, если репозиторий возвращает IQueryable<T>, абстракция немедленно «протекает». Слой бизнес-логики, получающий IQueryable, может добавить вызовы .Where(), .OrderBy() или .Select(). Это означает, что бизнес-слой должен знать о структуре базы данных и именах колонок, что нивелирует саму цель разделения ответственности. Если же репозиторий возвращает материализованные списки (List<T> или IEnumerable<T>), мы теряем возможность отложенного выполнения (Deferred Execution) и фильтрации на стороне сервера базы данных, что ведет к катастрофическому падению производительности при выборке больших объемов данных.4

### 2.2. Vertical Slice Architecture (VSA): Отказ от Горизонтальных Слоев

В противовес традиционной «луковой» (Onion) или слоистой (N-Tier) архитектуре, где приложение делится на технические слои (Controller -> Service -> Repository -> DAL), набирает популярность подход Vertical Slice Architecture (VSA). Этот подход особенно актуален для Unity-разработчиков, привыкших мыслить категориями игровых механик (Features), а не технических слоев.

#### 2.2.1. Фича как Единица Изменения

В VSA приложение структурируется вокруг бизнес-сценариев (Use Cases), или «слайсов». Каждый слайс представляет собой вертикальный срез системы, содержащий всё необходимое для выполнения одной конкретной задачи: от приема HTTP-запроса до сохранения в БД. Это устраняет необходимость прыгать между четырьмя разными проектами и папками (DTO, Interfaces, Services, Repositories) для изменения одной фичи, например, «Добавить поле 'Отчество' при регистрации».6

**Сравнительный анализ структур:**

| **Характеристика** | **N-Tier / Clean Architecture** | **Vertical Slice Architecture** |
| --- | --- | --- |
| **Группировка** | По техническому назначению (Controllers, Services) | По бизнес-фичам (Features/Bookings/Create) |
| **Связность (Cohesion)** | Низкая (логика фичи размазана) | Высокая (всё, что меняется вместе, лежит вместе) |
| **Доступ к данным** | Жестко через Repositories/Services | Гибкий: EF Core, Dapper или Raw SQL внутри слайса |
| **Дублирование** | Стремление к переиспользованию (Generic Repositories) | Принятие дублирования ради изоляции (Decoupling) |

#### 2.2.2. Структура Папок и «Кричащая Архитектура»

Структура проекта в VSA должна «кричать» о том, что делает приложение, а не о том, какой фреймворк используется. В контексте нашего сервиса Календаря это выглядит следующим образом 6:

src/

├── Features/

│ ├── Bookings/

│ │ ├── CreateBooking/ <-- Слайс "Создание брони"

│ │ │ ├── CreateBookingEndpoint.cs (API Controller / Minimal API)

│ │ │ ├── CreateBookingRequest.cs (DTO)

│ │ │ ├── CreateBookingValidator.cs (FluentValidation)

│ │ │ ├── CreateBookingHandler.cs (Бизнес-логика + DB)

│ │ │ └── CreateBookingResponse.cs

│ │ ├── GetAvailability/ <-- Слайс "Проверка доступности"

│ │ │ ├── GetAvailabilityQuery.cs

│ │ │ └──...

│ └── Reports/

│ ├── GetMonthlyStats/ <-- Слайс "Отчеты"

│ └── GetMonthlyStatsHandler.cs (Здесь может быть Dapper!)

Такой подход позволяет выбирать инструмент доступа к данным для каждого конкретного случая. В слайсе CreateBooking мы можем использовать EF Core для валидации доменной модели и сохранения графа объектов. В соседнем слайсе GetMonthlyStats, который требует сложной агрегации миллионов записей, мы можем использовать чистый SQL через Dapper, не нарушая никакой «общей архитектуры», так как слайсы изолированы друг от друга. Это решает проблему «общего знаменателя», когда весь проект страдает от ограничений одной выбранной абстракции.6

#### 2.2.3. Интеграция с CQRS и MediatR

VSA естественно ложится на паттерн CQRS (Command Query Responsibility Segregation). Запросы на изменение состояния (Commands) и запросы на чтение (Queries) обрабатываются разными хендлерами. Библиотека MediatR часто используется как «клей» для связывания запросов и обработчиков, позволяя контроллерам быть максимально тонкими. Однако, даже без MediatR, прямая инъекция хендлеров в эндпоинты является валидным и часто более производительным подходом, избавляющим от лишней рефлексии.7

## 3. Гибридная Модель Доступа: Симбиоз EF Core и Dapper

Несмотря на колоссальный прогресс в производительности EF Core 8 (компилируемые запросы, улучшенный SQL-транслятор), существуют сценарии, где накладные расходы ORM становятся узким местом. В высоконагруженных системах Enterprise-уровня, таких как сервис глобального календаря, часто применяется гибридный подход: EF Core используется для команд (Commands) — вставок, обновлений, где важна бизнес-логика и трекинг изменений, а Dapper используется для запросов (Queries) — быстрых выборок данных для чтения (Read-Models).5

### 3.1. Dapper как Микро-ORM: Преимущества и Ограничения

Dapper — это набор методов расширения для интерфейса IDbConnection. Его ключевое преимущество — работа с «сырым» SQL и маппинг результатов напрямую в POCO (Plain Old CLR Objects) с минимальными накладными расходами памяти. В отличие от EF Core, Dapper не создает снимков состояния (snapshots), не отслеживает изменения и не строит сложные графы объектов, если это не требуется явно. Это делает его идеальным для формирования DTO, которые отправляются клиенту.

Однако, Dapper — это «глупый» маппер. Он не знает о специфических типах PostgreSQL, таких как jsonb, массивы или новые типы дат.NET 8 (DateOnly, TimeOnly), если его этому не научить. Это создает разрыв между богатой функциональностью провайдера Npgsql и примитивным маппингом Dapper.12

### 3.2. Архитектура Общей Транзакции (Shared Transaction)

Одной из самых сложных задач при гибридном подходе является обеспечение атомарности операций, затрагивающих оба инструмента. Представим сценарий: мы создаем событие через EF Core (чтобы сработала доменная логика валидации) и одновременно хотим записать сложную аналитическую запись через Dapper в ту же транзакцию. Если Dapper упадет, вставка EF Core тоже должна откатиться.

#### 3.2.1. Механика IDbContextTransaction

EF Core инкапсулирует управление транзакциями через IDbContextTransaction. Чтобы Dapper мог участвовать в этой транзакции, он должен использовать то же самое физическое соединение с базой данных и тот же объект транзакции ADO.NET (System.Data.Common.DbTransaction).

Алгоритм реализации общей транзакции следующий 11:

1. **Инициализация:** Транзакция всегда начинается через DbContext, так как он является «владельцем» Unit of Work. Метод \_context.Database.BeginTransactionAsync() возвращает обертку транзакции.
2. **Извлечение зависимостей:** Из контекста извлекается текущее соединение (\_context.Database.GetDbConnection()) и, из объекта транзакции EF, извлекается вложенная транзакция ADO.NET (efTransaction.GetDbTransaction()).
3. **Выполнение EF:** Выполняются стандартные операции Add, Update и SaveChangesAsync. Важно понимать, что SaveChangesAsync не закроет транзакцию автоматически, если она была открыта вручную.
4. **Выполнение Dapper:** Вызываются методы ExecuteAsync или QueryAsync. Критически важно передать параметр transaction, полученный на шаге 2. Если этого не сделать, Dapper попытается выполнить команду вне транзакции (или упадет, если соединение занято транзакцией), что приведет к нарушению целостности или дедлокам.
5. **Фиксация (Commit):** Транзакция фиксируется через EF-обертку.

#### 3.2.2. Реализация Best Practices

Ниже приведен эталонный пример реализации метода в сервисе (или VSA хендлере), демонстрирующий этот паттерн 11:

C#

public async Task<int> CreateEventWithAuditLog(Event newEvent, string auditPayload)  
{  
 // 1. Старт транзакции через EF Core (Unit of Work)  
 using var efTransaction = await \_context.Database.BeginTransactionAsync();  
  
 try  
 {  
 // 2. Операция EF Core: Вставка доменной сущности  
 // EF Core использует ChangeTracker и сгенерирует INSERT при SaveChanges  
 \_context.Events.Add(newEvent);  
 await \_context.SaveChangesAsync();   
  
 // 3. Подготовка Dapper: Получение доступа к "кишкам" соединения  
 // Важно: Не используем 'using' для connection, так как он управляется EF Context!  
 var connection = \_context.Database.GetDbConnection();   
   
 // Получаем низкоуровневую транзакцию для передачи в Dapper  
 var dbTransaction = efTransaction.GetDbTransaction();  
  
 // 4. Операция Dapper: Высокопроизводительная вставка лога  
 // Используем специфичный синтаксис Postgres (::jsonb)  
 var sql = @"  
 INSERT INTO event\_audit\_logs (event\_id, payload, created\_at)  
 VALUES (@EventId, @Payload::jsonb, @CreatedAt)";  
   
 await connection.ExecuteAsync(sql, new {   
 EventId = newEvent.Id, // Id уже доступен после SaveChangesAsync  
 Payload = auditPayload,   
 CreatedAt = DateTime.UtcNow   
 }, transaction: dbTransaction); // <-- Обязательная привязка транзакции  
  
 // 5. Фиксация единой транзакции  
 await efTransaction.CommitAsync();  
   
 return newEvent.Id;  
 }  
 catch (Exception)  
 {  
 // Откат всех изменений (и EF, и Dapper) при любой ошибке  
 await efTransaction.RollbackAsync();  
 throw;  
 }  
}

### 3.3. Расширение Типовой Системы Dapper (Type Handlers)

Dapper «из коробки» не умеет работать со сложными типами PostgreSQL, которые мы активно используем в проекте. Это требует регистрации кастомных TypeHandler.

#### 3.3.1. JSONB и Проблема IDataRecord

Когда Npgsql читает колонку типа jsonb, он возвращает её как строку (string) или символьный поток, если не сконфигурирован иначе. Dapper, пытаясь смапить это в свойство объекта (например, UserSettings Settings), не знает, как превратить строку в объект. В EF Core 8 это делается автоматически через ToJson(), но в Dapper мы должны написать код вручную.12

Реализация JsonbTypeHandler использует System.Text.Json для сериализации/десериализации. Важным нюансом является явное указание NpgsqlDbType.Jsonb при записи. Если этого не сделать, Npgsql отправит параметр как text, и PostgreSQL может отвергнуть запрос или выполнить неявное приведение типов, что лишит нас преимуществ бинарного JSONB (например, индексации).16

#### 3.3.2. DateOnly и TimeOnly в.NET 8

Типы DateOnly и TimeOnly являются структурами, введенными для решения проблемы «лишнего времени» в датах (дни рождения, календарные даты). Npgsql 8.0 полностью поддерживает их маппинг на типы date и time в Postgres. Однако Dapper может по инерции пытаться мапить их как DateTime, что приведет к исключениям или потере данных.18

Регистрация хендлера обязательна для корректной работы:

C#

public class DateOnlyTypeHandler : SqlMapper.TypeHandler<DateOnly>  
{  
 public override void SetValue(IDbDataParameter parameter, DateOnly value)  
 {  
 parameter.Value = value;  
 parameter.DbType = DbType.Date; // Явное указание типа ADO.NET  
 }  
  
 public override DateOnly Parse(object value)  
 {  
 // Обработка полиморфизма возвращаемых значений Npgsql  
 return value switch  
 {  
 DateOnly d => d,  
 DateTime dt => DateOnly.FromDateTime(dt),  
 string s => DateOnly.Parse(s),  
 \_ => throw new DataException($"Cannot convert {value.GetType()} to DateOnly")  
 };  
 }  
}

Этот код должен выполняться один раз при старте приложения (например, в Program.cs), чтобы Dapper «узнал» о новых типах.18

## 4. Инженерная Культура Качества: Интеграционное Тестирование

В мире Unity тестирование часто сводится к ручной проверке в редакторе (Play Mode) или написанию простых юнит-тестов для алгоритмов. В Enterprise-системах, где логика распределена между кодом приложения и базой данных (Constraints, Triggers, Default Values), юнит-тесты с использованием моков (Mock<DbContext>) становятся не только бесполезными, но и вредными. Они создают ложное чувство безопасности, проверяя поведение мока, а не реальной базы данных.21

### 4.1. Смена Парадигмы: От Mocks к Testcontainers

Единственный способ гарантировать, что ваш SQL-запрос (LINQ или Dapper) работает корректно, особенно при использовании специфических фич Postgres (tsrange, jsonb, xmin), — это выполнить его на реальном инстансе PostgreSQL.

Библиотека **Testcontainers** революционизировала этот процесс в.NET. Она позволяет программно управлять Docker-контейнерами прямо из тестового кода. Это устраняет проблему «на моей машине работает», так как среда тестирования (версия Postgres, конфигурация) становится частью кодовой базы и идентична для всех разработчиков и CI/CD пайплайнов.22

### 4.2. Управление Жизненным Циклом Контейнеров в xUnit

Наивный подход к использованию Testcontainers заключается в создании нового контейнера для каждого теста (IAsyncLifetime в каждом классе). Однако запуск контейнера Postgres занимает 2-5 секунд. Если у вас 100 тестов, прогон займет 8 минут только на ожидание запуска БД. Это убивает быстрый цикл обратной связи (Feedback Loop).22

#### 4.2.1. Паттерн Singleton Container (ICollectionFixture)

Для решения проблемы производительности в xUnit используется механизм ICollectionFixture. Он позволяет создать единственный экземпляр тестового контекста (содержащего контейнер БД) и разделить его между множеством тестовых классов.

Архитектура тестового решения:

1. **PostgresContainerFixture:** Класс, ответственный за конфигурацию и запуск контейнера через PostgreSqlBuilder. Он реализует IAsyncLifetime для асинхронного старта (InitializeAsync) и остановки (DisposeAsync) контейнера. Важно использовать образ, максимально близкий к продакшену (например, postgres:16-alpine).
2. **Collection Definition:** Класс-маркер с атрибутом ``, который связывает фикстуру с коллекцией тестов.
3. **BaseIntegrationTest:** Абстрактный класс, от которого наследуются все интеграционные тесты. Он получает фикстуру через конструктор и предоставляет доступ к AppDbContext и ConnectionString.25

### 4.3. Проблема Изоляции Данных и Respawn

Использование одного контейнера для всех тестов (Shared Database) порождает новую проблему: **загрязнение состояния**. Тест А создает пользователя, Тест Б рассчитывает, что таблица пользователей пуста, но находит там пользователя из Теста А и падает.

Традиционное решение — оборачивать каждый тест в транзакцию (TransactionScope) и откатывать её в конце. Однако этот метод имеет недостатки:

* Не работает, если тестируемый код сам управляет транзакциями (как в нашем примере с Dapper и гибридной транзакцией).
* Может маскировать проблемы с дедлоками и уровнями изоляции.

#### 4.3.1. Respawn: Интеллектуальная Очистка

Библиотека **Respawn**, разработанная Джимми Богардом, предлагает более надежный подход. Вместо пересоздания БД или отката транзакции, она выполняет интеллектуальную очистку таблиц. Respawn анализирует схему базы данных, строит граф зависимостей внешних ключей и удаляет данные в правильном порядке, чтобы не нарушить целостность (Referential Integrity).

Внедрение Respawn 27:

1. Инициализация Respawner происходит один раз при старте фикстуры. Здесь мы конфигурируем таблицы, которые **нельзя** удалять (например, таблицу миграций \_\_EFMigrationsHistory и справочники).
2. Метод ResetAsync вызывается в методе DisposeAsync (или TearDown) каждого теста. Это гарантирует, что каждый тест получает чистую базу, но без накладных расходов на перезапуск контейнера.

C#

// Конфигурация Respawner  
\_respawner = await Respawner.CreateAsync(\_connection, new RespawnerOptions  
{  
 DbAdapter = DbAdapter.Postgres,  
 SchemasToInclude = new { "public" },  
 TablesToIgnore = new Table { "\_\_EFMigrationsHistory" }  
});

### 4.4. Тестирование Специфики Postgres: Ограничения Исключения (Exclusion Constraints)

Одной из ключевых задач недели является реализация защиты от овербукинга (Double Booking) на уровне базы данных. Это нельзя проверить моками. Тест должен реально попытаться вставить пересекающийся интервал и убедиться, что база данных отвергла транзакцию.

#### 4.4.1. Перехват и Анализ PostgresException

При нарушении EXCLUDE ограничения, Npgsql выбрасывает исключение PostgresException (часто обернутое в DbUpdateException при использовании EF Core). Ключевым маркером ошибки является свойство SqlState. Согласно стандарту PostgreSQL, код ошибки нарушения ограничения исключения — **23P01** (exclusion\_violation).29

Пример стратегии ассерта в xUnit тесте 32:

C#

[Fact]  
public async Task CreateBooking\_OverlappingInterval\_ThrowsExclusionViolation()  
{  
 // Arrange: Создаем конфликтную ситуацию  
 var timeRange = new NpgsqlRange<DateTime>(DateTime.UtcNow, DateTime.UtcNow.AddHours(1));  
 await CreateBookingAsync(roomId, timeRange); // Успешное создание  
  
 // Act & Assert  
 var exception = await Assert.ThrowsAsync<DbUpdateException>(async () =>   
 await CreateBookingAsync(roomId, timeRange) // Попытка дублирования  
 );  
  
 // Deep Assert: Проверяем, что причина именно в констрейнте Postgres, а не в валидации C#  
 var postgresEx = exception.InnerException as PostgresException;  
 Assert.NotNull(postgresEx);  
 Assert.Equal("23P01", postgresEx.SqlState);  
}

Этот тест является золотым стандартом проверки надежности системы. Он подтверждает, что даже если валидация на уровне приложения (C#) будет пропущена или обойдена (например, при гонке потоков), база данных гарантирует целостность данных.

## 5. Синтез: Рекомендации по Итоговым Проектам (Capstone Projects)

В завершение теоретического блока, сформулируем архитектурные директивы для реализации финальных проектов недели, основанные на рассмотренном материале.

### 5.1. Проект А: «Система Бронирования Переговорных» (Room Booker)

* **Архитектурный стиль:** Строго **Vertical Slice Architecture**. Создайте отдельную папку Features/Bookings и поместите туда всю логику. Не создавайте BookingService.
* **Уровень данных:** Используйте **EF Core** для основных операций CRUD. Реализуйте защиту от овербукинга через миграцию с Raw SQL (ALTER TABLE... ADD CONSTRAINT... EXCLUDE...), так как Fluent API это не поддерживает.
* **Тестирование:** Напишите интеграционный тест с **Testcontainers**, который пытается создать два пересекающихся бронирования параллельно (используя Task.WhenAll), чтобы проверить работу блокировок БД.

### 5.2. Проект Б: «Аналитический Дашборд» (Analytics Dashboard)

* **Архитектурный стиль:** **CQRS**. Разделите операции записи (события календаря) и операции чтения (статистика).
* **Уровень данных:** Используйте **Гибридный подход**.
  + Запись событий: EF Core (для гарантии инвариантов).
  + Чтение дашборда: **Dapper** + Raw SQL. Напишите SQL-запрос с агрегацией JSONB полей (jsonb\_each, count).
* **Оптимизация:** Примените **AsNoTracking** для всех read-only запросов EF Core.
* **Типы:** Реализуйте JsonbTypeHandler для Dapper, чтобы читать метаданные событий в типизированные C# объекты.

### 5.3. Заключение

Переход от Unity к Enterprise.NET требует от вас переосмысления того, где «живет» истина. В играх истина часто в памяти (RAM). В Enterprise истина всегда в базе данных. Паттерны, изученные сегодня — VSA, Dapper, Testcontainers — это инструменты, позволяющие строить системы, которые не просто «работают», а являются надежными, масштабируемыми и, что самое важное, предсказуемыми в своем поведении.

#### Источники

1. План обучения общий неделя 8
2. Repository Pattern vs Direct EF Core: The Great Debate - DEV ..., дата последнего обращения: декабря 6, 2025, <https://dev.to/adrianbailador/repository-pattern-vs-direct-ef-core-the-great-debate-1fon>
3. Repository pattern (the real deal) - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@bananicabananica/repository-pattern-the-real-deal-94218155c0ba>
4. Those of you who are using EF Core and skipping the "Repository" layer by directly injecting dbcontext into you service classes. Are you just rawdogging linq data access code in your service methods, or is linq code abstracted away somehow? : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/1h5ub4x/those_of_you_who_are_using_ef_core_and_skipping/>
5. Building Scalable APIs with Vertical Slice Architecture in .NET | by Amit Naik - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/expertminds/building-scalable-apis-with-vertical-slice-architecture-in-net-23f9d8c8a84e>
6. Vertical Slice Architecture in .NET — From N‑Tier Layers to Feature Slices - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/cristiansifuentes/vertical-slice-architecture-in-net-from-n-tier-layers-to-feature-slices-4iha>
7. Vertical Slice Architecture: The Best Ways to Structure Your Project : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/1eo7uhk/vertical_slice_architecture_the_best_ways_to/>
8. Develop Vertical Slice Feature Folder with CQRS and MediatR in .NET 8 Microservices, дата последнего обращения: декабря 6, 2025, <https://mehmetozkaya.medium.com/develop-vertical-slice-feature-folder-with-cqrs-and-mediatr-in-net-8-microservices-32efb99e78f8>
9. poorna-soysa/vertical-slice-architecture-template - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/poorna-soysa/vertical-slice-architecture-template>
10. The Simplest Vertical Slice Architecture With .NET 8 - YouTube, дата последнего обращения: декабря 6, 2025, <https://www.youtube.com/watch?v=dQdXHRkePr8>
11. Dapper + Entity Framework Core (EF Core) together against PostgreSQL in .NET 8, дата последнего обращения: декабря 6, 2025, <https://www.c-sharpcorner.com/article/dapper-entity-framework-core-ef-core-together-against-postgresql-in-net-8/>
12. PostgreSQL JSONB in .NET - Marek Sirkovský - Medium, дата последнего обращения: декабря 6, 2025, <https://mareks-082.medium.com/postgresql-jsonb-in-net-25fbcc7b64b2>
13. Using Entity Framework Core and Dapper in ASP.NET Core – Safe Transactions, дата последнего обращения: декабря 6, 2025, <https://sd.blackball.lv/en/articles/read/18761-using-entity-framework-core-and-dapper-in-aspnet-core-safe-transactions>
14. Dapper and EF Core transaction fails with error SqlConnection does not support parallel transactions - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/59405072/dapper-and-ef-core-transaction-fails-with-error-sqlconnection-does-not-support-p>
15. JSON Mapping | Npgsql Documentation, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/efcore/mapping/json.html>
16. Dapper Query Postgresql Jsonb Column To String · Issue #719 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/StackExchange/Dapper/issues/719>
17. Bug/Regression doing a query with a jsonb column as string with 2.1.\* · Issue #2049 · DapperLib/Dapper - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/DapperLib/Dapper/issues/2049>
18. Dapper DateOnly/TimeOnly - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/karenpayneoregon/dapper-dateonlytimeonly-1ii9>
19. "Date" default binding to "DateTime", and not "DateOnly" - causing issues with dynamic binding with Dapper #5792 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/npgsql/npgsql/issues/5792>
20. NotSupportedException: The member IssueDate of type System.DateOnly cannot be used as a parameter value - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/78369971/notsupportedexception-the-member-issuedate-of-type-system-dateonly-cannot-be-us>
21. How to use Testcontainers with .NET Unit Tests - The JetBrains Blog, дата последнего обращения: декабря 6, 2025, <https://blog.jetbrains.com/dotnet/2023/10/24/how-to-use-testcontainers-with-dotnet-unit-tests/>
22. Testing with xUnit.net - Testcontainers for .NET, дата последнего обращения: декабря 6, 2025, <https://dotnet.testcontainers.org/test_frameworks/xunit_net/>
23. Getting started with Testcontainers for .NET, дата последнего обращения: декабря 6, 2025, <https://testcontainers.com/guides/getting-started-with-testcontainers-for-dotnet/>
24. IAsyncLifetime and IClassFixture with Testcontainers · xunit xunit · Discussion #2834, дата последнего обращения: декабря 6, 2025, <https://github.com/xunit/xunit/discussions/2834>
25. Testcontainers Best Practices for .NET Integration Testing - Milan Jovanović, дата последнего обращения: декабря 6, 2025, <https://www.milanjovanovic.tech/blog/testcontainers-best-practices-dotnet-integration-testing>
26. Supercharge Your dotnet Integration Tests: Using Testcontainers, xUnit, and AssemblyFixture for Real SQL Server Testing - Md hasanuzzzaman, дата последнего обращения: декабря 6, 2025, <https://md-hasanuzzaman.medium.com/supercharge-your-net-36924d78104e>
27. Practical Approaches to Database Testing with Testcontainers, дата последнего обращения: декабря 6, 2025, <https://roosma.dev/p/practical-approaches-database-testing-testcontainers/>
28. jbogard/Respawn: Intelligent database cleaner for integration tests - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/jbogard/Respawn>
29. PostgresDB 23P01: Exclusion Violation - Doctor Droid, дата последнего обращения: декабря 6, 2025, <https://drdroid.io/stack-diagnosis/postgresdb-23p01-exclusion-violation>
30. Class PostgresException | Npgsql Documentation, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/doc/api/Npgsql.PostgresException.html>
31. Documentation: 18: Appendix A. PostgreSQL Error Codes, дата последнего обращения: декабря 6, 2025, <https://www.postgresql.org/docs/current/errcodes-appendix.html>
32. Assert an Exception using XUnit - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/45017295/assert-an-exception-using-xunit>