# Архитектурная теория гибридной персистентности: Глубокое погружение в объектно-реляционное отображение и неструктурированные типы данных в экосистеме.NET 8 и PostgreSQL

## 1. Введение: Эпистемологический сдвиг в управлении состоянием

### 1.1. От эфемерности кадра к вечности транзакции

Переход от разработки интерактивных приложений реального времени (GameDev/Unity) к проектированию корпоративных серверных систем (Enterprise Backend) требует не просто изучения нового синтаксиса или библиотек, но фундаментальной перестройки ментальной модели работы с данными. В контексте восьмой недели обучения, посвященной персистентности, второй день является поворотным моментом, где происходит столкновение двух философий: императивного управления состоянием в оперативной памяти и декларативного управления данными в постоянном хранилище.1

Для старшего разработчика Unity привычной средой обитания является оперативная память (RAM). Объекты в сцене (GameObject), их компоненты и связи существуют в едином адресном пространстве процесса. Сохранение (persistence) воспринимается как дискретное событие — "чекпоинт" или завершение сессии, когда граф объектов сериализуется в JSON, XML или бинарный поток и сбрасывается на диск. В этой парадигме база данных часто рассматривается как пассивное хранилище файлов, "большой жесткий диск", чья задача — просто принять байты и вернуть их в неизменном виде. Состояние приложения в основном *Stateful* — оно живет в памяти между кадрами.

В архитектуре высоконагруженных систем, таких как проектируемый сервис "Календарь", действует парадигма *Stateless*. Сервер приложений — это эфемерный обработчик, который не должен хранить состояние между HTTP-запросами. Истинным владельцем данных становится Система Управления Базами Данных (СУБД) PostgreSQL. Здесь база данных выступает не как склад, а как сложный вычислительный узел, гарантирующий целостность, конкурентный доступ и долговечность данных (ACID).1

Теория второго дня восьмой недели фокусируется на том, как преодолеть "разрыв несоответствия" (Impedance Mismatch) между объектно-ориентированной природой C# и реляционной (а теперь и гибридной) природой PostgreSQL. Мы переходим от примитивных скалярных типов к сложным структурам данных: JSONB-документам, массивам и диапазонам, используя всю мощь Entity Framework Core 8 (EF Core 8) не как простого маппера, а как инструмента моделирования предметной области.1

### 1.2. Контекст задачи: Гибридная природа современных данных

Разрабатываемый сервис "Календарь" представляет собой классический пример системы, требующей гибридного подхода к хранению данных. С одной стороны, существуют жесткие реляционные структуры: пользователи, события, комнаты. Эти данные требуют строгой схемы, ссылочной целостности (Foreign Keys) и транзакционных гарантий. С другой стороны, современные требования к персонализации и интеграции диктуют необходимость хранения слабоструктурированных данных: пользовательских настроек интерфейса (UserSettings), метаданных событий, приходящих из внешних систем (Jira, Zoom, Slack), и динамических наборов тегов.

Традиционный SQL-подход (до появления PostgreSQL JSONB) предлагал для решения этих задач паттерн EAV (Entity-Attribute-Value), который известен своей крайне низкой производительностью и сложностью поддержки. Альтернативой была полная нормализация, приводящая к созданию десятков таблиц для хранения простых настроек. Современный стек.NET 8 и PostgreSQL предлагает третий путь: использование jsonb и массивов как первоклассных типов данных. Это позволяет внедрять документы внутри реляционных таблиц, объединяя гибкость NoSQL (как в MongoDB) с надежностью SQL.1

В данном документе мы проведем исчерпывающий анализ теоретических основ и практических механик реализации этого гибридного подхода, опираясь на план обучения 1 и техническую документацию Npgsql и Microsoft.1

## 2. PostgreSQL как объектно-реляционная платформа: Взгляд изнутри

Понимание того, как EF Core мапит объекты, невозможно без глубокого понимания того, *куда* именно эти объекты попадают. PostgreSQL не является классической реляционной СУБД (RDBMS); это объектно-реляционная система (ORDBMS). Это различие критично для понимания теории второго дня.

### 2.1. Анатомия JSONB: Почему это не просто текст

Одной из самых распространенных ошибок разработчиков, приходящих из мира Unity, является восприятие JSON в базе данных как обычной строки (String), содержащей скобки и кавычки. В PostgreSQL существуют два типа данных для JSON: json и jsonb. Понимание различий между ними является фундаментом для принятия архитектурных решений на этой неделе.

Тип json — это действительно текстовое хранилище. Он сохраняет форматирование, пробелы и порядок ключей. При каждом запросе к такому полю база данных вынуждена производить парсинг текста, что создает колоссальную нагрузку на CPU. Это аналог TextAsset в Unity — просто блоб текста.

Тип jsonb (binary JSON), который мы используем в нашем плане обучения 1, — это декомпозированная бинарная структура. При вставке данных (INSERT или UPDATE) PostgreSQL выполняет следующие операции:

1. **Парсинг и валидация:** Проверяется синтаксическая корректность JSON.
2. **Нормализация:** Удаляются незначимые пробелы.
3. **Дедупликация:** Если в JSON есть дублирующиеся ключи (например, {"a": 1, "a": 2}), сохраняется только последнее значение ("a": 2).
4. **Сортировка:** Ключи объектов сортируются (обычно по длине и лексикографически). Это критически важно для бинарного поиска.
5. **Структурирование:** Данные преобразуются в дерево JEntry, где заголовки содержат информацию о типе и длине значений.

**Архитектурное следствие:** Использование jsonb немного замедляет операции записи (из-за накладных расходов на конвертацию), но радикально ускоряет операции чтения и позволяет производить глубокую индексацию. Для сервиса календаря, где настройки пользователя читаются при каждом запросе, а меняются редко, jsonb является безальтернативным выбором.1

### 2.2. Механизм TOAST и физическое хранение

Unity-разработчики привыкли заботиться о фрагментации кучи (Heap Fragmentation). В PostgreSQL существует схожая проблема управления страницами памяти. Размер страницы данных в Postgres фиксирован и составляет обычно 8 КБ. Что происходит, если мы пытаемся сохранить JSON-документ размером 50 КБ в одну строку?

Здесь вступает в игру механизм TOAST (The Oversized-Attribute Storage Technique). Если строка превышает пороговое значение (обычно 2 КБ), PostgreSQL пытается сжать большие атрибуты (включая JSONB). Если сжатия недостаточно, данные нарезаются на чанки (обычно по 2 КБ) и сохраняются в отдельной системной таблице TOAST, а в основной таблице остается лишь указатель.

Влияние на проектирование схемы (Schema Design):

Понимание TOAST критично для выбора стратегии маппинга в EF Core. Если мы решаем хранить огромный граф объектов (например, полную историю изменений события) в одной колонке jsonb, каждый запрос SELECT \* FROM Events будет вынужден обращаться к TOAST-таблице, генерируя дополнительные операции I/O (чтения диска). Это объясняет, почему на третий день обучения мы будем рассматривать проекции (.Select()). Однако уже на этапе проектирования (День 2) мы должны следовать правилу: в jsonb колонках следует хранить только те данные, которые логически являются частью агрегата и часто запрашиваются вместе с ним. Хранение редко используемых тяжелых данных в JSONB внутри часто запрашиваемой таблицы — антипаттерн.1

### 2.3. Массивы (Arrays) как нативная структура

В отличие от многих других SQL-баз, PostgreSQL поддерживает многомерные массивы для любого типа данных. В контексте плана обучения 1 это используется для хранения тегов событий.

В C# массив — это ссылочный тип, занимающий непрерывный блок памяти. В PostgreSQL массив — это также специальная структура данных, которая содержит заголовок (размерность, типы, OID элемента) и данные.

Сравним архитектурные подходы:

| **Характеристика** | **Нормализованная модель (Таблица связей)** | **Модель массивов PostgreSQL (text)** |
| --- | --- | --- |
| **Физическое хранение** | Данные разбросаны по трем таблицам (События, Теги, Связи). Требуется поиск по B-Tree индексам для сборки. | Данные локальны (Data Locality). Массив хранится внутри строки события (или в TOAST рядом). |
| **Операция вставки** | 3 транзакционных записи в WAL (Write Ahead Log). Обновление индексов трех таблиц. | 1 запись в WAL. Обновление индекса одной таблицы. |
| **Операция чтения** | Сложный JOIN, потребляющий CPU и память для Hash Join или Nested Loop. | Простая выборка колонки. |
| **Целостность (FK)** | Гарантируется базой данных. Нельзя удалить используемый тег. | Не гарантируется автоматически. Требует триггеров или логики приложения. |

**Вывод:** Для сущностей типа "Теги", которые семантически являются простыми значениями (Value Objects) и не имеют собственного сложного поведения, использование массивов text является предпочтительным паттерном оптимизации в высоконагруженных системах на PostgreSQL.1

## 3. Entity Framework Core 8: Революция маппинга JSON

До выхода.NET 8 работа с JSON в EF Core требовала использования ValueConverters. Это был "костыль": EF Core просто сериализовывал объект в строку перед отправкой и десериализовывал обратно при чтении. Это делало невозможным выполнение SQL-запросов, фильтрующих данные *внутри* JSON. Разработчик был вынужден загружать все данные в память и фильтровать их через LINQ-to-Objects, что катастрофически сказывалось на производительности (проблема N+1 и Memory Leaks).

### 3.1. Механика ToJson() и Owned Entity Types

EF Core 8 ввел нативную поддержку маппинга JSON, которая интегрирована в конвейер трансляции запросов LINQ. Это реализуется через концепцию "Владеемых типов" (Owned Types).

В парадигме Domain-Driven Design (DDD) существуют Сущности (Entities), имеющие уникальный идентификатор, и Объекты-Значения (Value Objects), определяемые своими атрибутами. UserSettings в нашем календаре — это классический Value Object. Он не имеет смысла в отрыве от пользователя.

C#

// Конфигурация в DbContext  
protected override void OnModelCreating(ModelBuilder modelBuilder)  
{  
 modelBuilder.Entity<User>()  
 .OwnsOne(u => u.Settings, builder =>  
 {  
 builder.ToJson(); // Магический метод EF Core 8  
 });  
}

Что происходит "под капотом" при вызове builder.ToJson()?

1. **Изменение стратегии маппинга:** По умолчанию OwnsOne использует стратегию "Table Splitting" (все поля объекта сохраняются как отдельные колонки в таблице родителя). ToJson() переключает это на маппинг в единственную колонку типа jsonb (для Npgsql).
2. **Генерация прокси-доступа:** EF Core создает внутренние структуры для навигации по свойствам JSON-объекта при построении деревьев выражений (Expression Trees).
3. **Трансляция LINQ:** Когда разработчик пишет запрос:  
   C#  
   var users = context.Users.Where(u => u.Settings.Theme == "Dark").ToList();  
     
   EF Core 8 распознает, что Settings — это JSON-колонка, а Theme — это ключ внутри нее. Транслятор генерирует SQL, специфичный для PostgreSQL:  
   SQL  
   SELECT id, user\_name, email, settings  
   FROM users  
   WHERE settings ->> 'Theme' = 'Dark'

Это фундаментальное изменение. Мы получаем возможность работать с денормализованными данными, используя строгую типизацию C# и мощь SQL-движка для фильтрации.1

### 3.2. Вложенные коллекции и проблема "Cartesian Explosion"

EF Core 8 также поддерживает OwnsMany внутри JSON. Например, если у пользователя есть список адресов доставки, их можно сохранить в той же JSONB-колонке.

C#

modelBuilder.Entity<User>()  
 .OwnsOne(u => u.Settings, settings =>  
 {  
 settings.ToJson();  
 settings.OwnsMany(s => s.CustomShortcuts); // Список горячих клавиш  
 });

Это решает одну из главных проблем реляционных баз данных — проблему "декартова взрыва" (Cartesian Explosion). Когда мы делаем JOIN пользователя с таблицей адресов, количество строк в результирующем наборе умножается. Если у пользователя 5 адресов и 10 телефонов, JOIN вернет 50 строк для одного пользователя. Передача этих данных по сети избыточна.

Хранение коллекций в JSONB устраняет эту проблему: возвращается *одна* строка, содержащая упакованный массив данных. Для Unity-разработчика это интуитивно понятно — это похоже на загрузку префаба, содержащего дочерние объекты, за один вызов, вместо инстанцирования каждого дочернего объекта отдельным вызовом.1

### 3.3. Теневые свойства (Shadow Properties) и автоматический аудит

Еще одной мощной концепцией EF Core, рассматриваемой на второй день, являются теневые свойства. В Unity компоненты часто имеют публичные поля для настройки в редакторе. В Enterprise-системах мы стремимся к инкапсуляции и чистоте доменной модели.

Shadow Properties — это свойства, которые существуют в схеме базы данных, но отсутствуют в классе C# сущности. Это идеальный механизм для служебных данных, таких как:

* CreatedAt (Дата создания)
* LastModifiedAt (Дата последнего изменения)
* xmin (Системная колонка версионирования Postgres)

**Конфигурация:**

C#

modelBuilder.Entity<User>()  
 .Property<DateTime>("LastModifiedAt");

Доступ и использование:

Поскольку свойства нет в классе, к нему нельзя обратиться через точку (user.LastModifiedAt). Доступ осуществляется через API отслеживания изменений:

C#

context.Entry(user).Property("LastModifiedAt").CurrentValue = DateTime.UtcNow;

В рамках плана обучения 1 это используется для реализации автоматического аудита. Переопределяя метод SaveChanges в DbContext, мы можем автоматически находить все измененные сущности и обновлять их теневые свойства. Это избавляет бизнес-логику от рутинного кода обновления дат и гарантирует, что эти поля никогда не будут забыты.

**Связь с JSONB:** Теневые свойства не могут быть частью JSONB-документа, так как они требуют отдельных колонок для эффективной работы индексов и триггеров БД. Это подчеркивает гибридный характер архитектуры: метаданные аудита живут в реляционных колонках, а бизнес-конфигурация — в JSONB.

## 4. Продвинутые типы данных и Npgsql

Провайдер Npgsql — это мост между миром.NET и миром PostgreSQL. На второй день обучения мы детально разбираем, как этот мост работает для специфических типов.

### 4.1. DateOnly и TimeOnly: Конец "Birthday Paradox"

В версиях.NET до 6.0 (и EF Core до 8.0 в полной мере) для хранения даты без времени использовался тип DateTime. Это порождало бесконечные проблемы с часовыми поясами.

* *Сценарий:* Пользователь родился 15 апреля.
* *Проблема:* Мы сохраняем 2000-04-15 00:00:00 (UTC). Если пользователь открывает календарь в часовом поясе UTC-5 (Нью-Йорк), он видит 2000-04-14 19:00:00. День рождения сместился на день назад.

Для решения этой проблемы приходилось писать сложные конвертеры. В.NET 8 типы DateOnly и TimeOnly являются первоклассными гражданами. Npgsql мапит их напрямую на типы PostgreSQL date и time без каких-либо преобразований часовых поясов.

SQL

-- PostgreSQL  
birthday date NOT NULL

C#

// C#  
public DateOnly Birthday { get; set; }

Это архитектурное решение полностью устраняет целый класс ошибок, связанных с временными зонами, что критично для календаря.1

### 4.2. Маппинг перечислений (Enums)

В Unity enum часто сериализуется как int. Это эффективно, но делает данные в базе нечитаемыми (Role: 1 — это админ или модератор?). Сериализация в строку (Role: "Admin") читаема, но занимает много места и медленна при поиске.

PostgreSQL поддерживает нативные перечисления (CREATE TYPE mood AS ENUM (...)). Npgsql позволяет мапить C# enum на Postgres enum.

Однако, в плане обучения 1 часто рекомендуется подход "Smart Enum" или хранение строкового представления в базе для упрощения миграций (изменение Postgres ENUM требует блокировки таблицы).

Для гибридного подхода, если Enum хранится внутри JSONB, он автоматически сериализуется как строка или число (в зависимости от настроек JsonSerializerOptions). Это создает интересный нюанс: внутри JSONB мы теряем строгую типизацию базы данных, которую имели бы с нативным ENUM колонкой.

## 5. Индексация гибридных данных: Теория GIN

Создание гибкой схемы данных с помощью JSONB бессмысленно, если поиск по ней занимает секунды. На второй день мы знакомимся с концепцией обобщенного обратного индекса (Generalized Inverted Index — GIN).

### 5.1. B-Tree vs GIN

Классический индекс B-Tree (сбалансированное дерево), используемый для стандартных колонок (ID, Email), эффективен для операторов равенства (=), диапазона (<, >) и сортировки. Однако он бесполезен для запросов вида "содержит ли этот JSON документ ключ 'Theme'?".

Индекс GIN работает иначе. Он спроектирован для композитных значений. Для JSONB документа GIN индекс декомпозирует документ на ключи и значения и сохраняет их в структуру, напоминающую оглавление книги:

* Ключ "Theme" ->
* Значение "Dark" ->

### 5.2. Операторы включения и их поддержка в EF Core

Для использования GIN индекса SQL-запрос должен использовать специфические операторы. Самый важный из них — оператор включения @> (contains).

SQL

-- Находит строки, где JSON содержит фрагмент  
SELECT \* FROM "Users" WHERE "Settings" @> '{"Theme": "Dark"}'

EF Core 8 умеет транслировать метод .Contains для массивов и некоторые паттерны для JSON в этот оператор. Однако, простые сравнения (u.Settings.Theme == "Dark") транслируются в оператор извлечения ->>, который **не может** использовать стандартный GIN индекс по всему JSONB-полю (хотя может использовать специализированный B-Tree индекс по выражению).

**Критический инсайт для обучения:** Разработчик должен понимать, что "гибкость" JSONB не дается бесплатно. Индекс GIN занимает больше места на диске и медленнее обновляется, чем B-Tree. Каждое обновление JSONB-документа приводит к обновлению индекса. Поэтому индексировать нужно только те поля JSON, по которым реально происходит поиск.1

## 6. Архитектурные паттерны: Инкапсуляция и DDD

Второй день обучения также затрагивает вопросы чистоты кода и защиты инвариантов.

### 6.1. Backing Fields (Скрытые поля)

В Unity распространена практика public полей. В Enterprise это недопустимо. Свойства должны быть защищены. EF Core позволяет мапить приватные поля.

C#

public class User  
{  
 private string \_email; // Backing field  
  
 public string Email  
 {  
 get => \_email;  
 private set // Закрытый сеттер  
 {  
 if (!value.Contains("@")) throw new ValidationException();  
 \_email = value;  
 }  
 }  
}

Конфигурация .HasField("\_email") говорит EF Core писать напрямую в поле, минуя сеттер при загрузке из БД. Это важно, так как данные в БД считаются "доверенными" и валидация при чтении (Read Path) избыточна и может вызвать ошибки при изменении правил валидации в будущем.1

### 6.2. Разделение моделей (CQRS Lite)

Теория второго дня подводит к мысли, что одна модель данных не может идеально подходить и для записи, и для чтения. Использование JSONB — это шаг в сторону CQRS (Command Query Responsibility Segregation).

* **Write Model:** Доменная сущность с методами поведения, использующая богатые типы.
* **Read Model:** При чтении через Dapper (тема 5-го дня) или проекции EF Core мы можем выбирать данные из JSONB напрямую в плоские DTO (Data Transfer Objects) для отправки клиенту, минуя сложную логику домена. JSONB облегчает создание таких "Read Models", так как данные уже хранятся в формате, близком к тому, что ожидает фронтенд.

## 7. Практическое применение в проекте "Календарь"

Все вышеописанные теоретические концепции кристаллизуются в конкретных задачах второго дня плана обучения.

### 7.1. Реализация UserSettings

Создание свойства UserSettings в классе User и его конфигурация через OwnsOne + ToJson — это "Hello World" в мире гибридной персистентности. Это учит разработчика мыслить агрегатами. Вместо создания таблицы UserThemes, UserNotifications, UserPrivacy, мы создаем один документ настроек.

* *Преимущество:* Атомарность чтения и записи. Мы загружаем пользователя вместе с его настройками за один IO запрос.
* *Риск:* Если настройки станут слишком большими (например, пользователь начнет хранить там логи), это замедлит загрузку основной сущности пользователя.

### 7.2. События и Naming Conventions

Настройка UseSnakeCaseNamingConvention — это урок дисциплины. В C# принят PascalCase, в Postgres — snake\_case. Автоматическое преобразование имен позволяет C# коду выглядеть идиоматично (UserId), а базе данных — следовать своим стандартам (user\_id). Это снимает когнитивную нагрузку с разработчика, которому не нужно помнить точные имена колонок в базе при написании LINQ-запросов.1

### 7.3. Инфраструктурный слой

Выделение конфигурации EF Core в отдельный проект Calendar.Infrastructure учит разделению ответственности. Calendar.Domain содержит чистые классы (POCO), а Calendar.Infrastructure знает о том, что Settings — это JSONB, а Tags — это массив Postgres. Это инверсия зависимостей в действии: домен не зависит от базы данных, но инфраструктура зависит от домена и настраивает базу под него.

## 8. Глобальный анализ: Влияние на производительность и надежность

Завершая теоретический обзор второго дня, необходимо синтезировать полученные знания в контексте метрик качества программного обеспечения.

### 8.1. Влияние на I/O и пропускную способность

Переход на JSONB и массивы снижает количество необходимых JOIN операций. В традиционной реляционной модели выборка события с его тегами и настройками автора могла потребовать 3-4 соединения таблиц. В гибридной модели это часто один SELECT из одной таблицы.

* **Снижение Latency:** Меньше времени на планирование запроса и его выполнение.
* **Увеличение Throughput:** База данных может обработать больше запросов, так как каждый из них потребляет меньше ресурсов CPU.

### 8.2. Влияние на надежность и целостность

Здесь гибридный подход требует компромиссов. Отказ от внешних ключей (Foreign Keys) для данных внутри JSON означает, что приложение берет на себя ответственность за целостность данных.

* *Пример:* Если мы храним {"PreferredRoomId": 123} внутри JSON настроек, база данных не помешает нам удалить комнату 123.
* *Решение:* Использовать JSONB только для данных, которые действительно являются "Частными" для родительского объекта и не имеют сильных связей с другими агрегатами. Для сильных связей всегда использовать реляционные колонки и внешние ключи.

### 8.3. Итог

Второй день восьмой недели — это момент взросления разработчика. Отказ от "серебряной пули" (будь то полная нормализация или полная сериализация в BLOB) в пользу прагматичного гибридного подхода характеризует переход на уровень Senior Backend Engineer. Понимание внутренней механики jsonb, text и конвейера трансляции EF Core 8 позволяет создавать системы, которые не просто работают, а масштабируются и поддаются поддержке на протяжении лет.1

Данный теоретический базис служит фундаментом для последующих дней недели, где эти структуры данных будут подвергнуты нагрузочному тестированию и оптимизации конкурентного доступа.

#### Источники

1. План обучения общий неделя 8