# Архитектурная парадигма персистентности в Enterprise-системах: Глубокий анализ теории первого дня восьмой недели

## 1. Эпистемологический сдвиг: От Stateful-систем реального времени к Stateless-архитектуре транзакционной обработки

Переход квалифицированного инженера из индустрии разработки интерактивных приложений реального времени (Unity/GameDev) в сферу корпоративных серверных систем (Enterprise Backend) сопряжен с необходимостью фундаментальной когнитивной перестройки. Восьмая неделя учебного плана, сфокусированная на персистентности данных и реляционной алгебре, представляет собой не просто изучение новой библиотеки доступа к данным, а смену базовой ментальной модели управления состоянием приложения.

В среде Unity разработчик привык оперировать так называемой «горячей памятью» (Hot RAM). Игровые объекты, менеджеры состояния и контроллеры существуют в управляемой куче (Managed Heap) на протяжении длительных периодов времени — от загрузки сцены до завершения сессии. Доступ к данным в такой архитектуре является мгновенным (порядка наносекунд), а само приложение классифицируется как **Stateful** (сохраняющее состояние). Персистентность здесь часто воспринимается как вторичный, вспомогательный механизм: периодический дамп состояния в JSON-файлы или локальную базу SQLite, происходящий синхронно или асинхронно, но редко являющийся блокирующим фактором для основного цикла рендеринга (Game Loop).1

Напротив, архитектура корпоративного сервиса, такого как проектируемая система "Календарь", базируется на **Stateless** (бессостоятельной) парадигме. Сервер приложений ASP.NET Core рассматривается как эфемерный вычислительный узел, который не владеет данными, а лишь временно заимствует их для обработки конкретной бизнес-транзакции. База данных PostgreSQL становится единственным источником истины (Single Source of Truth), и любое изменение состояния должно быть атомарно зафиксировано в ней, прежде чем система подтвердит успешное выполнение операции клиенту. Этот сдвиг требует от инженера отказа от интуитивного желания кэшировать данные в статических переменных или Singleton-сервисах, так как в распределенной системе (например, при горизонтальном масштабировании на несколько подов Kubernetes) локальная память одного экземпляра сервиса несинхронизирована с другими.2

### 1.1. Сравнительный анализ архитектурных примитивов

Для формализации различий между подходами необходимо рассмотреть, как изменяется семантика работы с данными. В то время как Unity-разработчик оптимизирует код для минимизации аллокаций памяти в каждом кадре (Zero Garbage Allocation), Backend-архитектор оптимизирует количество и сложность сетевых вызовов к базе данных (I/O Bound Operations).

| **Аспект архитектуры** | **Парадигма Unity (GameDev)** | **Парадигма.NET Enterprise (Backend)** | **Импликации для Недели 8** |
| --- | --- | --- | --- |
| **Владение данными** | Приложение владеет данными в RAM. БД — это архив. | БД владеет данными. Приложение — временный процессор. | Необходимость минимизации объема выборки (Projections) и отказ от загрузки всего графа объектов.4 |
| **Жизненный цикл** | Длительный (минуты/часы). Объекты живут между кадрами. | Краткосрочный (миллисекунды). Контекст создается и умирает с HTTP-запросом. | Строгое управление IDisposable и понимание Scoped-зависимостей.4 |
| **Конкурентность** | Низкая. Обычно один поток логики (Main Thread). | Экстремальная. Тысячи параллельных запросов к одним данным. | Внедрение механизмов оптимистичной блокировки (Optimistic Concurrency) и транзакционной изоляции.1 |
| **Сохранение** | Сериализация (SaveGame). Полный снимок состояния. | Транзакция (Commit). Изменение только дельты (разницы) данных. | Использование Change Tracker для вычисления минимально необходимого SQL UPDATE.5 |
| **Масштабируемость** | Вертикальная (мощность устройства пользователя). | Горизонтальная (количество инстансов сервиса). | Запрет на хранение сессионных данных в памяти процесса (State Leaks).3 |

Данная таблица демонстрирует, почему прямой перенос паттернов из Unity (например, создание глобального DataManager как Singleton) в среду ASP.NET Core приводит к критическим архитектурным ошибкам, таким как утечки памяти и некорректная обработка параллельных запросов.

## 2. Архитектурный паттерн Unit of Work и его реализация в Entity Framework Core

Центральным элементом теоретической базы первого дня является концепция **Unit of Work** (Единица работы), реализованная в классе DbContext. В отличие от низкоуровневых соединений (таких как NpgsqlConnection), DbContext представляет собой сложную абстракцию, объединяющую в себе паттерн Repository и Identity Map (Карту идентичности). Понимание внутреннего устройства DbContext критично для предотвращения проблем производительности, характерных для новичков в ORM.4

### 2.1. Внутренняя механика Change Tracking (Отслеживания изменений)

В экосистеме Unity состояние объекта изменяется явно разработчиком, и разработчик сам решает, когда и как это сохранить. В EF Core используется декларативный подход: разработчик изменяет свойства POCO-объектов (Plain Old CLR Objects), а фреймворк автоматически вычисляет необходимые SQL-инструкции. Этот процесс базируется на механизме **Change Tracker**.

Существует две фундаментальные стратегии отслеживания изменений, понимание которых необходимо для оптимизации 5:

1. **Snapshot Change Tracking (Снапшоты):** Это стратегия по умолчанию. Когда сущность извлекается из базы данных, EF Core сохраняет теневую копию (snapshot) всех её свойств во внутреннем словаре. При вызове метода SaveChanges() запускается алгоритм DetectChanges(). Он итерирует по всем отслеживаемым объектам, сравнивает их текущие значения со снапшотами и формирует граф изменений. Этот процесс надежен, но его вычислительная сложность линейно зависит от количества загруженных объектов ($O(N)$), что может стать узким местом при массовой обработке данных.6
2. **Notification/Proxy Tracking (Прокси-уведомления):** В этом сценарии EF Core генерирует динамические классы-наследники для сущностей, переопределяя виртуальные свойства. При изменении свойства (entity.Name = "New") прокси-объект мгновенно уведомляет трекер об изменении. Это исключает необходимость полного сканирования в DetectChanges(), однако накладывает жесткие ограничения на архитектуру классов (все свойства должны быть virtual, класс не может быть sealed), что противоречит современным тенденциям к использованию неизменяемых (immutable) структур данных и записей (Records) в C#.8

В контексте восьмой недели и перехода на.NET 8 рекомендуется использование Snapshot-стратегии как более устойчивой и предсказуемой, однако разработчик должен быть осведомлен о накладных расходах памяти на хранение дубликатов данных в снапшотах.

### 2.2. Жизненный цикл DbContext: Scoped vs Singleton

Одной из самых распространенных и разрушительных ошибок при переходе с Unity на серверную разработку является неправильное управление временем жизни зависимостей (Dependency Injection Lifetimes). В Unity синглтоны являются стандартом де-факто для менеджеров. В ASP.NET Core регистрация DbContext как Singleton является антипаттерном, приводящим к фатальным последствиям.4

#### Проблема потокобезопасности (Thread Safety)

Класс DbContext **не является потокобезопасным**. Внутренние структуры данных, такие как Dictionary<Key, EntityEntry>, не защищены блокировками. В среде веб-сервера, где каждый HTTP-запрос обрабатывается в отдельном потоке (или асинхронном контексте), использование одного экземпляра DbContext несколькими потоками одновременно неминуемо приведет к состоянию гонки (Race Condition), повреждению внутренних структур и выбросу исключения InvalidOperationException с сообщением о конкурентном доступе.4

#### Проблема Captive Dependency (Захваченная зависимость)

Даже если DbContext зарегистрирован правильно (как Scoped), риск возникает при его внедрении в сервисы с более длительным временем жизни, например, в Singleton-сервисы (фоновые задачи, кэши). Если Singleton-сервис принимает в конструктор DbContext, он "захватывает" этот экземпляр контекста и удерживает его на протяжении всего времени работы приложения. Это явление называется **Captive Dependency**.11

Последствия захвата контекста катастрофичны для производительности:

* **Утечка памяти (Memory Leak):** Поскольку Change Tracker хранит ссылки на все загруженные объекты, "захваченный" контекст будет бесконечно накапливать данные в памяти, так как он никогда не уничтожается.
* **Устаревание данных (Stale Data):** Контекст работает с данными, актуальными на момент их загрузки. Долгоживущий контекст будет возвращать устаревшие данные из своего кэша первого уровня, игнорируя изменения, сделанные другими транзакциями.13

**Архитектурное решение:** Для использования DbContext в долгоживущих сервисах (например, IHostedService) необходимо применять паттерн фабрики областей видимости (IServiceScopeFactory). Вместо прямой инъекции контекста, сервис должен создавать новый Scope, запрашивать в нем DbContext, выполнять работу и уничтожать скоуп.11

### 2.3. Оптимизация через DbContext Pooling

В высоконагруженных сценариях (High-Frequency Trading, игровые бэкенды) постоянное создание и уничтожение объектов DbContext может создавать избыточное давление на сборщик мусора (Garbage Collector), так как инициализация контекста включает в себя настройку внутренних сервисов и парсеров опций.14

.NET 8 предлагает механизм **DbContext Pooling** (AddDbContextPool). В этом режиме при вызове Dispose() контекст не уничтожается, а сбрасывает свое состояние (Reset State) и возвращается в специальный пул управляемых объектов. При следующем запросе готовый экземпляр извлекается из пула, что позволяет сэкономить на аллокациях памяти и инициализации.14

Однако использование пулинга требует строгой дисциплины:

* **State Reset Pitfalls:** Если разработчик добавляет в наследник DbContext собственные приватные поля (например, private int \_currentUserId), EF Core не сможет их автоматически очистить при возврате в пул. Это приведет к утечке данных между запросами разных пользователей — критической уязвимости безопасности.16
* **Global State:** Контекст в пуле не должен зависеть от изменяемого глобального состояния, которое фиксируется при его создании.

Для первого дня обучения рекомендуется использовать стандартный AddDbContext, переходя к AddDbContextPool только на этапе профилирования производительности, четко понимая механизмы сброса состояния.14

## 3. Технологический стек PostgreSQL: Специфика провайдера Npgsql и маппинг типов

Выбор PostgreSQL в качестве системы управления базами данных для сервиса "Календарь" не случаен. В отличие от многих реляционных СУБД, PostgreSQL предоставляет богатый набор типов данных, специфичных для временных рядов и неструктурированных документов, что делает его гибридным решением (Relational + NoSQL).1 Провайдер Npgsql в связке с EF Core 8 позволяет прозрачно транслировать эти возможности в код на C#.

### 3.1. Эволюция типов даты и времени: DateOnly и TimeOnly

До выхода.NET 6/8 и соответствующих обновлений EF Core, работа с датами в связке C# + SQL была источником постоянных ошибок. Тип System.DateTime в.NET исторически содержит как дату, так и время, а также неопределенность в отношении часового пояса (DateTimeKind). При маппинге на тип date в PostgreSQL (который не содержит времени) часто происходили неявные конвертации, приводящие к смещению дат из-за временных зон (например, 2023-10-05 00:00:00 UTC превращалось в 2023-10-04 23:00:00 в локальном времени -1).19

В.NET 8 и EF Core 8 введена нативная поддержка типов **DateOnly** и **TimeOnly**. Эти структуры идеально соответствуют типам date и time в стандарте SQL.20

* **DateOnly:** Хранит только год, месяц и день. Идеально для поля "День рождения" пользователя. Оно не подвержено смещению часовых поясов, так как день рождения — это абсолютное календарное понятие в контексте документов.
* **TimeOnly:** Хранит время без привязки к дате. Идеально для хранения расписания работы (например, "магазин открыт с 09:00 до 18:00") или длительности событий, не превышающих 24 часов.19

Провайдер Npgsql 8.0 автоматически мапит эти типы, устраняя необходимость в написании кастомных ValueConverter и предотвращая целый класс ошибок, связанных с временными зонами.22 Для проекта календаря это означает, что поле User.Birthday должно быть строго типизировано как DateOnly уже на уровне доменной модели.

### 3.2. Конфигурация NpgsqlDataSourceBuilder

Современный подход к конфигурации подключения в Npgsql (начиная с версии 7.0) базируется на абстракции NpgsqlDataSource. Это фабрика соединений, которая инкапсулирует в себе конфигурацию пула соединений, логирование, типы данных и плагины (например, NodaTime или NetTopologySuite).24

Использование NpgsqlDataSourceBuilder позволяет настроить глобальные правила маппинга один раз при старте приложения, вместо того чтобы конфигурировать каждый DbContext отдельно.

C#

var dataSourceBuilder = new NpgsqlDataSourceBuilder(connectionString);  
dataSourceBuilder.EnableDynamicJson(); // Подготовка к работе с JSONB (День 2)  
var dataSource = dataSourceBuilder.Build();  
  
builder.Services.AddDbContext<ApplicationDbContext>(options =>  
 options.UseNpgsql(dataSource));

Этот подход обеспечивает лучшую производительность и управление ресурсами, особенно в архитектурах с несколькими контекстами, использующими одну базу данных.25

### 3.3. Соглашения именования: Snake Case Rewriter

PostgreSQL чувствителен к регистру идентификаторов, если они заключены в кавычки. Исторически в мире PostgreSQL принят стандарт snake\_case (нижний регистр с подчеркиванием), тогда как в C# стандартом является PascalCase.

Если не настроить маппинг имен, EF Core сгенерирует SQL-запросы вида:

SQL

SELECT "Id", "UserName" FROM "Users"

Это требует постоянного использования кавычек при написании ручных SQL-скриптов и нарушает конвенции PostgreSQL.

Библиотека EFCore.NamingConventions и метод UseSnakeCaseNamingConvention() решают эту проблему, автоматически транслируя имена сущностей и свойств в snake\_case при построении модели.27

Результат:

SQL

SELECT id, user\_name FROM users

Это делает схему базы данных более идиоматичной для администраторов PostgreSQL и упрощает интеграцию с внешними инструментами аналитики, которые могут ожидать стандартного именования.27

## 4. Стратегии эволюции схемы данных: Миграции, идемпотентность и CI/CD

В отличие от подхода Unity, где структура сохраненных данных часто определяется структурой класса на момент сериализации, в реляционных базах данных схема (Schema) является строгим контрактом. Любое изменение этого контракта (добавление колонки, изменение типа) должно быть версионировано и применено контролируемо.

### 4.1. Анатомия миграции и ModelSnapshot

Механизм миграций EF Core опирается на три ключевых компонента, генерируемых при создании миграции 29:

1. **Файл миграции (Migration Class):** Содержит методы Up() (применение изменений) и Down() (откат). Это императивные инструкции для изменения схемы.
2. **Файл метаданных (Designer File):** Содержит техническую информацию о версии EF и целевой БД.
3. **ModelSnapshot (Снимок модели):** Это критически важный файл (DbContextModelSnapshot.cs), который содержит полное описание текущего состояния модели данных на языке Fluent API.

Когда разработчик добавляет новую миграцию (dotnet ef migrations add), EF Core не смотрит в реальную базу данных. Вместо этого он сравнивает текущую C#-модель с ModelSnapshot.29 Разница (Diff) между ними и становится новой миграцией. После генерации миграции ModelSnapshot обновляется.

Это объясняет, почему конфликты слияния в файле ModelSnapshot.cs при работе в команде являются частой проблемой: если два разработчика добавили миграции параллельно, их снапшоты разойдутся, и EF Core не сможет корректно вычислить следующую дельту.29

### 4.2. Идемпотентность в процессах CI/CD

Автоматическая доставка изменений базы данных (Database Delivery) — один из самых сложных аспектов DevOps. Наивный подход — запуск команды dotnet ef database update на продакшн-сервере — категорически не рекомендован по соображениям безопасности (требует наличия SDK на сервере, хранит исходный код) и надежности.32

Современным стандартом является генерация идемпотентных SQL-скриптов (Idempotent SQL Scripts).

Идемпотентность в контексте БД означает, что скрипт можно запустить на базе данных в любом состоянии (пустой, частично обновленной, полностью обновленной), и он корректно приведет её к целевому состоянию, не вызывая ошибок дублирования.34

Генерация такого скрипта выполняется командой:

Bash

dotnet ef migrations script --idempotent --output migration.sql

EF Core оборачивает каждую миграцию в блок проверки:

SQL

IF NOT EXISTS(SELECT \* FROM "\_\_EFMigrationsHistory" WHERE "MigrationId" = '20251205\_AddUsers')  
BEGIN  
 CREATE TABLE users (...);  
 INSERT INTO "\_\_EFMigrationsHistory"...  
END;

Этот скрипт можно безопасно передать DBA (администратору баз данных) или запустить через автоматизированный пайплайн развертывания (Azure DevOps, GitHub Actions, GitLab CI), используя утилиты типа sqlcmd или psql.35

### 4.3. Migration Bundles (Миграционные пакеты)

В.NET 8 набирает популярность альтернативный подход — **Migration Bundles**. Это самодостаточные исполняемые файлы (Single-file executables), которые содержат в себе логику миграций и необходимый рантайм EF Core.

Bash

dotnet ef migrations bundle --self-contained -r linux-x64

Бандл решает проблему зависимости от.NET SDK на целевом сервере и позволяет запускать миграции как простой шаг в Docker-контейнере (./efbundle). Это упрощает CI/CD пайплайны, устраняя необходимость в генерации и пересылке SQL-скриптов, сохраняя при этом безопасность выполнения.38

## 5. Заключение и синтез: От теории к практике первого дня

Теоретический базис первого дня восьмой недели формирует фундамент для всей последующей работы над сервисом "Календарь". Мы деконструировали процесс перехода от Unity-мышления к Enterprise-мышлению, выделив ключевые точки бифуркации:

1. **Смена парадигмы состояния:** От долгоживущих объектов в памяти к транзакционному циклу "Загрузка -> Изменение -> Сохранение" в рамках Scoped-контекста. Понимание Unit of Work и Change Tracker позволяет эффективно управлять этим циклом, избегая лишних запросов к БД.
2. **Дисциплина управления зависимостями:** Строгое разделение Scoped и Singleton сервисов предотвращает утечки памяти и коллизии данных, характерные для "наивного" переноса игровых паттернов на сервер.
3. **Использование возможностей платформы:** Выбор Npgsql и конфигурация SnakeCase, DateOnly/TimeOnly превращает PostgreSQL из "черного ящика" в мощный аналитический узел, способный эффективно выполнять операции реляционной алгебры, разгружая приложение.
4. **Культура DevOps:** Внедрение идемпотентных миграций с первого дня обучения прививает стандарты промышленной разработки, где код не существует в вакууме, а является частью непрерывного конвейера поставки.

Практическое задание первого дня — "Инициализация Инфраструктуры" — требует синтеза всех вышеописанных концепций. Создание изолированного слоя доступа к данным, настройка DbContext с правильным пулингом и маппингом типов, а также организация пайплайна миграций — это не просто "настройка базы", а проектирование нервной системы будущего приложения, способной выдерживать нагрузки и эволюционировать вместе с бизнес-требованиями. Этот фундамент позволит на следующих этапах (День 2-5) беспрепятственно внедрять сложные функции, такие как поиск по JSONB, предотвращение овербукинга через Range Types и оптимизацию конкурентного доступа.

#### Источники

1. План обучения общий неделя 8
2. Stateful vs. Stateless Web App Design - DreamFactory Blog, дата последнего обращения: декабря 6, 2025, <https://blog.dreamfactory.com/stateful-vs-stateless-web-app-design>
3. Stateful vs. Stateless Architecture - GeeksforGeeks, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/system-design/stateful-vs-stateless-architecture/>
4. DbContext Lifetime, Configuration, and Initialization - EF Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/dbcontext-configuration/>
5. difference between DetectChange and ChangeTracking in entity framework - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/36292295/difference-between-detectchange-and-changetracking-in-entity-framework>
6. Change Detection and Notifications - EF Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/change-tracking/change-detection>
7. Working With Change Tracking Proxy in Entity Framework 6.0 - C# Corner, дата последнего обращения: декабря 6, 2025, <https://www.c-sharpcorner.com/UploadFile/ff2f08/working-with-change-tracking-proxy-in-entity-framework-6-0/>
8. Should I enable or disable dynamic proxies with entity framework 4.1 and MVC3?, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/7111109/should-i-enable-or-disable-dynamic-proxies-with-entity-framework-4-1-and-mvc3>
9. Should you use Entity Framework change-tracking proxies? - 1unicorn2, дата последнего обращения: декабря 6, 2025, <https://blog.oneunicorn.com/2011/12/05/should-you-use-entity-framework-change-tracking-proxies/>
10. Can someone explain when to use Singleton, Scoped and Transient with some real life examples? : r/csharp - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/csharp/comments/1acwtar/can_someone_explain_when_to_use_singleton_scoped/>
11. Dependency injection guidelines - .NET | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines>
12. The ASP.NET Core Dependency Injection System | ABP.IO, дата последнего обращения: декабря 6, 2025, <https://abp.io/community/articles/the-asp.net-core-dependency-injection-system-3vbsdhq8>
13. How to refresh an Entity Framework Core DBContext? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/46205114/how-to-refresh-an-entity-framework-core-dbcontext>
14. Advanced Performance Topics - EF Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/performance/advanced-performance-topics>
15. Entity Framework DbContext with pooling v without pooling with EF 8 on .NET 8, дата последнего обращения: декабря 6, 2025, <https://dotnetbenchmarks.com/benchmark/1006>
16. DbContext Pooling in .NET 8: A Deep Dive into Performance Optimization - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@serhatalftkn/dbcontext-pooling-in-net-8-a-deep-dive-into-performance-optimization-9e7af6f480f0>
17. Does .AddDbContextPool increase overall performance of the data layer? : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/f3szqe/does_adddbcontextpool_increase_overall/>
18. What is called when a DbContext is reset when pooling is used? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/53385522/what-is-called-when-a-dbcontext-is-reset-when-pooling-is-used>
19. Date and Time Handling | Npgsql Documentation, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/doc/types/datetime.html>
20. How to Map DateOnly and TimeOnly Types to SQL - Code Maze, дата последнего обращения: декабря 6, 2025, <https://code-maze.com/csharp-map-dateonly-timeonly-types-to-sql/>
21. What's New in EF Core 8 | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-8.0/whatsnew>
22. Use DateOnly and TimeOnly with EF Core 6, 7 & 8 and Azure SQL / SQL Server, дата последнего обращения: декабря 6, 2025, <https://erikej.github.io/efcore/sqlserver/2023/09/03/efcore-dateonly-timeonly.html>
23. Persisting Date Only in PostgreSQL Database - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/78187437/persisting-date-only-in-postgresql-database>
24. Npgsql Entity Framework Core Provider, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/efcore/>
25. Npgsql Basic Usage, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/doc/basic-usage.html>
26. Multiple database contexts with Npgsql causing migration failure - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/77625973/multiple-database-contexts-with-npgsql-causing-migration-failure>
27. efcore/EFCore.NamingConventions: Entity Framework Core plugin to apply naming conventions to table and column names (e.g. snake\_case) - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/efcore/EFCore.NamingConventions>
28. Tables | Npgsql Documentation, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/efcore/modeling/tables.html>
29. The Model Snapshot In Entity Framework Core, дата последнего обращения: декабря 6, 2025, <https://www.learnentityframeworkcore.com/migrations/model-snapshot>
30. Managing Migrations - EF Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/managing>
31. Entity Framework Core Migrations Guide - Custom Software Development, дата последнего обращения: декабря 6, 2025, <https://blackflow.co.uk/entreprise-software-development/entity-framework-core-migrations-guide/>
32. entity framework - EF Migrations with Azure Pipeline Tasks - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/59761685/ef-migrations-with-azure-pipeline-tasks>
33. How to Use Entity Framework Core Migrations in Production - Code Maze, дата последнего обращения: декабря 6, 2025, <https://code-maze.com/efcore-how-to-use-entity-framework-core-migrations-in-production/>
34. Applying Migrations - EF Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying>
35. EF Core and migrating production databases : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/k8dodi/ef_core_and_migrating_production_databases/>
36. Entity Framework Core migrations through CI and CD in Azure DevOps - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/58027720/entity-framework-core-migrations-through-ci-and-cd-in-azure-devops>
37. EF Core migrations script file in Azure Pipelines - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/77451651/ef-core-migrations-script-file-in-azure-pipelines>
38. Introducing DevOps-friendly EF Core Migration Bundles - .NET Blog, дата последнего обращения: декабря 6, 2025, <https://devblogs.microsoft.com/dotnet/introducing-devops-friendly-ef-core-migration-bundles/>
39. How many of you use EF migrations? : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/1dgn1z7/how_many_of_you_use_ef_migrations/>