# Архитектурная Эпистемология: Фундаментальная теория и механика интеграционного тестирования в экосистеме ASP.NET Core

## 1. Введение: Онтологический сдвиг в парадигме верификации качества

Переход от разработки интерактивных приложений реального времени (Unity/GameDev) к проектированию распределенных корпоративных систем (Enterprise Backend) требует не просто смены инструментария, но и радикальной перестройки когнитивных моделей, касающихся природы программного обеспечения и критериев его качества. В контексте третьего дня пятнадцатой недели учебного плана, сфокусированного на интеграционном тестировании, мы сталкиваемся с необходимостью деконструкции привычных паттернов проверки работоспособности кода и внедрения строгих инженерных практик, обеспечивающих надежность на уровне архитектуры.

Если в среде Unity состояние системы часто воспринимается как глобальный, мутабельный феномен, привязанный к кадровому циклу (Game Loop), то в мире бэкенд-разработки на платформе.NET состояние является транзакционным, распределенным и конкурентным. Проверка корректности работы такой системы не может полагаться на визуальное наблюдение (аналог Play Mode) или ручное тестирование. Здесь вступают в силу законы автоматизированной верификации, где интеграционное тестирование занимает центральное место, выступая мостом между абстрактной логикой кода и физической реальностью инфраструктуры.

### 1.1. От детерминизма локальных вычислений к хаосу распределенных систем

Модульное тестирование (Unit Testing), с которым разработчик знакомится на ранних этапах, оперирует в искусственно созданном вакууме. Оно постулирует, что если отдельные компоненты системы функционируют корректно в изоляции, то и вся система будет работать правильно. Однако в энтерпрайз-разработке этот постулат часто оказывается ложным. Сложность современных микросервисов заключается не столько в алгоритмах внутри методов (бизнес-логика часто сводится к CRUD-операциям), сколько во взаимодействии между компонентами: контроллерами, мидлварами (middleware), базами данных, брокерами сообщений и внешними API.1

Интеграционный тест (Integration Test) в контексте ASP.NET Core — это эксперимент, проводимый в контролируемой, но максимально приближенной к реальности среде. Он проверяет не то, как компонент *должен* работать согласно спецификации, а то, как он *действительно* работает в условиях, имитирующих боевые. Это проверка контрактов, сериализации, работы Dependency Injection (DI) контейнера, корректности SQL-запросов и обработки HTTP-заголовков. Для Unity-разработчика это можно сравнить с переходом от проверки скрипта движения персонажа в пустой сцене к проверке того же скрипта в загруженном уровне с физикой, коллайдерами и асинхронной подгрузкой ассетов.2

### 1.2. Эволюция пирамиды тестирования: От Пирамиды к Трофею

Традиционная "Пирамида тестирования" (Testing Pyramid), популяризированная Майком Коном, предписывала иметь массивное основание из дешевых и быстрых Unit-тестов, средний слой интеграционных тестов и тонкую верхушку UI-тестов. Однако в современной облачной (Cloud-Native) разработке эта модель трансформируется в "Трофей тестирования" (Testing Trophy) или "Соту" (Testing Honeycomb).3

Причина этого сдвига кроется в архитектуре микросервисов. Сервис "Календарь", который разрабатывается в рамках курса, по своей сути является интеграционным узлом. Он принимает HTTP-запросы, валидирует их, преобразует в SQL-команды и возвращает результат. Если мы покроем Unit-тестами каждый класс в отдельности, используя моки (mocks) для всех зависимостей, мы получим тесты, которые проверяют лишь то, что "метод А вызвал метод Б". Такие тесты хрупки: они ломаются при любом рефакторинге внутренней структуры кода, даже если внешнее поведение системы не изменилось. Это явление известно как "тестирование реализации" (Implementation Details Testing) вместо "тестирования поведения" (Behavior Testing).5

Интеграционные тесты, напротив, рассматривают сервис как "черный ящик". Им безразлично, используется ли внутри паттерн Repository, CQRS или Mediator. Важно лишь, что на вход подается валидный JSON, а на выходе получается код ответа 200 OK и запись в базе данных. Это обеспечивает высокий уровень доверия (Confidence) к коду и устойчивость к рефакторингу, что критически важно для долгосрочной поддержки проекта.7

Таблица 1. Сравнительный анализ стратегий тестирования в контексте перехода с Unity на.NET Backend

| **Характеристика** | **Unit Testing (xUnit + Moq)** | **Integration Testing (WebApplicationFactory)** | **E2E Testing (Selenium/Playwright)** |
| --- | --- | --- | --- |
| **Аналогия Unity** | Edit Mode Tests (логика без сцены) | Play Mode Tests (Headless, без рендера) | Ручной плейтест билда (QA) |
| **Область видимости** | Класс / Метод в изоляции | Приложение в сборе (In-Process TestServer) | Вся система + UI (Out-of-Process) |
| **Работа с зависимостями** | Полная изоляция (Test Doubles/Mocks) | Реальные компоненты или Docker-контейнеры | Реальное окружение (Staging/Prod) |
| **Скорость выполнения** | < 1 мс (Микросекунды) | 100 - 500 мс (Миллисекунды) | Секунды / Минуты |
| **Устойчивость к рефакторингу** | Низкая (связаны с реализацией) | Высокая (связаны с публичным API контрактом) | Высокая, но хрупки к изменениям UI |
| **Стоимость поддержки** | Низкая | Средняя | Очень высокая |
| **Глубина верификации** | Проверяет алгоритм | Проверяет систему и конфигурацию | Проверяет бизнес-сценарий пользователя |

## 2. Анатомия WebApplicationFactory: Технический дип-дайв

Ключевым инструментом, обеспечивающим возможность интеграционного тестирования в ASP.NET Core, является класс WebApplicationFactory<TEntryPoint> из пространства имен Microsoft.AspNetCore.Mvc.Testing. Понимание внутренней механики этого класса необходимо для эффективного использования и отладки тестов. Это не просто утилита для запуска, это сложный механизм виртуализации среды исполнения.8

### 2.1. Механизм загрузки хоста (Bootstrapping)

В.NET 6 и более поздних версиях, использующих Minimal API и Top-Level Statements, точка входа в приложение (класс Program) часто является неявной. WebApplicationFactory использует механизмы рефлексии для обнаружения сборки, содержащей точку входа, и воспроизводит процесс запуска приложения в памяти тестового процесса.

Процесс инициализации WebApplicationFactory включает следующие этапы:

1. **Discovery (Обнаружение):** Фабрика анализирует generic-параметр TEntryPoint (обычно это Program или Startup), чтобы определить корневую сборку приложения и найти конфигурационные файлы (appsettings.json).
2. **Host Creation (Создание Хоста):** Фабрика создает экземпляр IHostBuilder (или IWebHostBuilder), фактически выполняя тот же код, который выполняется при реальном старте приложения (WebApplication.CreateBuilder). Это означает, что все регистрации в DI-контейнере, настройки конфигурации и логирования выполняются точно так же, как в продакшене.
3. **Pipeline Construction (Построение Конвейера):** Формируется конвейер обработки запросов (Middleware Pipeline). Все мидлвары — от обработки исключений до аутентификации — включаются в цепь.9

### 2.2. TestServer vs Kestrel: Иллюзия сетевого взаимодействия

Одним из самых неочевидных и важных аспектов работы WebApplicationFactory является то, что она **не запускает реальный веб-сервер Kestrel** и **не открывает сетевой порт**. Вместо этого используется TestServer — специализированная реализация интерфейса IServer.

**Kestrel (Реальный сервер):**

* Работает на уровне транспортного слоя (L4 OSI).
* Открывает TCP-сокет и слушает входящие соединения.
* Парсит сырые байты HTTP-запроса.
* Управляет пулом соединений, SSL-терминацией и буферами памяти.
* Подвержен ограничениям операционной системы (лимиты файловых дескрипторов, порты).

**TestServer (Виртуальный сервер):**

* Работает полностью в памяти процесса (In-Memory).
* Игнорирует сетевой стек операционной системы (нет TCP-рукопожатий, нет Loopback-интерфейса).
* Взаимодействует с приложением напрямую через абстракцию HttpContext.

Когда в интеграционном тесте выполняется вызов client.GetAsync("/api/events"):

1. HttpClient, созданный фабрикой, не выполняет сетевой запрос.
2. Он использует специальный HttpMessageHandler (внутренний класс ClientHandler в TestServer), который перехватывает запрос.
3. ClientHandler преобразует объект HttpRequestMessage непосредственно в HttpContext ASP.NET Core.
4. Сформированный HttpContext передается напрямую в начало пайплайна Middleware (RequestDelegate).11

**Архитектурные следствия и ограничения:**

* **Производительность:** Исключение накладных расходов на сетевой стек (сериализация/десериализация в байты, системные вызовы) делает тесты экстремально быстрыми.
* **Изоляция:** Отсутствие реальных портов позволяет запускать тесты параллельно в сотнях потоков без конфликтов (Port Exhaustion).
* **Слепые зоны:** TestServer не может проверить проблемы, специфичные для транспортного уровня. Вы не сможете протестировать таймауты TCP-соединений, поведение при разрыве связи, корректность настройки SSL-сертификатов, особенности работы HTTP/2 или HTTP/3, а также ограничения размера тела запроса (MaxRequestBodySize), которые накладывает Kestrel. Для проверки этих аспектов требуются полноценные E2E-тесты или нагрузочное тестирование.14

### 2.3. Жизненный цикл WebApplicationFactory и управление ресурсами

Класс WebApplicationFactory реализует интерфейс IAsyncDisposable. Это критически важно для предотвращения утечек ресурсов. Каждый экземпляр фабрики поднимает свой собственный DI-контейнер и набор сервисов.

В среде Unity разработчики привыкли к тому, что при остановке игры (выходе из Play Mode) память очищается. В контексте тестового прогона (Test Runner) процесс может жить долго. Если WebApplicationFactory создается заново для каждого теста (без использования IClassFixture), это приводит к огромным накладным расходам на инициализацию приложения (Cold Start) и значительному потреблению памяти.

Кроме того, если приложение использует статические поля (static fields) для хранения состояния (что является анти-паттерном, но встречается), это состояние будет сохраняться между тестами, так как все они выполняются в одном процессе. Это приводит к так называемым "мерцающим" (flaky) тестам, которые то проходят, то падают в зависимости от порядка выполнения.17

**Рекомендация:** Для оптимизации производительности тестов рекомендуется использовать IClassFixture в xUnit, чтобы создавать один экземпляр WebApplicationFactory на весь класс тестов, сбрасывая только состояние базы данных между тестами.

## 3. Проблема состояния данных: In-Memory Database как анти-паттерн

В плане обучения на третий день содержится строгое предостережение против использования провайдера Microsoft.EntityFrameworkCore.InMemory. Для разработчика, пришедшего из Unity, концепция базы данных в памяти кажется привлекательной своей простотой и скоростью. Однако в контексте Enterprise Backend использование In-Memory DB считается серьезным архитектурным риском и анти-паттерном для интеграционного тестирования.19

### 3.1. Иллюзия реляционности

Провайдер In-Memory не является реляционной базой данных. Это, по сути, сложный wrapper над коллекциями объектов в памяти. Он не эмулирует поведение настоящей СУБД, такой как PostgreSQL или SQL Server.

**Ключевые отличия, ведущие к ложноположительным тестам:**

1. **Ссылочная целостность (Referential Integrity):** In-Memory база не обеспечивает проверку внешних ключей (Foreign Keys). Вы можете успешно сохранить в таблицу Orders запись с CustomerId = 999, даже если пользователя с таким ID не существует. Тест пройдет успешно, но в продакшене код упадет с PostgresException: violate foreign key constraint.21
2. **Типизация и ограничения:** In-Memory база может позволить сохранить строку длиной 500 символов в поле, ограниченном VARCHAR(50), или сохранить NULL в поле NOT NULL, если валидация на уровне EF Core не сработает. Реальная база данных такие операции отклонит.
3. **Чувствительность к регистру (Case Sensitivity):** PostgreSQL по умолчанию чувствителен к регистру при сравнении строк (если не используется специальный collation). In-Memory база использует стандартное сравнение.NET строк, которое может зависеть от настроек StringComparison. Это приводит к тому, что поиск "User" найдет "user" в тесте, но не найдет в продакшене.23

### 3.2. Игнорирование транзакций

Один из самых опасных аспектов In-Memory провайдера — его поведение в отношении транзакций. По умолчанию он игнорирует методы управления транзакциями (BeginTransaction, Commit, Rollback).

Если ваша бизнес-логика полагается на транзакционность (например, списание средств со счета и начисление бонусов должны происходить атомарно), тест с In-Memory базой может пройти успешно, даже если код содержит ошибку, из-за которой транзакция не коммитится или не откатывается при исключении. В памяти изменения применяются мгновенно. Это создает ложное чувство безопасности.19

### 3.3. Несовместимость специфичных функций (Npgsql)

PostgreSQL обладает богатым набором функций, которые не имеют аналогов в стандартном SQL или LINQ. Использование таких типов данных, как JSONB (для хранения неструктурированных данных), Array (массивы примитивов) или полнотекстового поиска (tsvector), делает использование In-Memory базы невозможным.

При попытке выполнить LINQ-запрос, использующий специфичные для провайдера Npgsql функции (например, EF.Functions.JsonContains или операции с массивами), In-Memory провайдер выбросит исключение InvalidOperationException, так как он не знает, как транслировать эти вызовы. Это вынуждает разработчиков либо отказываться от мощных возможностей PostgreSQL, либо писать абстракции поверх EF Core только ради тестов, что усложняет архитектуру без бизнес-ценности.26

**Вывод:** Использование In-Memory DB переводит тест из категории "Интеграционный" в категорию "Сложный Unit-тест с состоянием". Для полноценной верификации системы необходимо использовать реальную СУБД. В рамках третьего дня обучения, до внедрения Docker и Testcontainers, допустимым компромиссом может быть использование SQLite в режиме in-memory, так как это *реляционная* база данных, но даже она имеет отличия от PostgreSQL. Золотым стандартом является тестирование на том же движке БД, который используется в продакшене.25

## 4. Конфигурация тестовой среды: Искусство подмены зависимостей

Главная мощь WebApplicationFactory заключается в возможности вмешиваться в процесс построения приложения и хирургически точно заменять отдельные компоненты системы перед запуском. Это делается с помощью метода ConfigureTestServices, который предоставляет доступ к IServiceCollection после того, как отработал основной код Startup/Program.

### 4.1. ConfigureTestServices против ConfigureServices

Существует важное различие между ConfigureServices (метод в IWebHostBuilder) и ConfigureTestServices (расширение из Microsoft.AspNetCore.TestHost).

ConfigureTestServices выполняется **после** стандартного метода ConfigureServices вашего приложения. Это гарантирует, что регистрации сервисов, сделанные в тесте, перепишут регистрации, сделанные в основном коде приложения. Это поведение основано на принципе работы DI-контейнера в.NET: если сервис зарегистрирован дважды, при разрешении зависимости используется последняя регистрация.30

Пример паттерна замены внешней зависимости:

C#

protected override void ConfigureWebHost(IWebHostBuilder builder)  
{  
 builder.ConfigureTestServices(services =>  
 {  
 // 1. Находим дескриптор реального сервиса  
 var descriptor = services.SingleOrDefault(  
 d => d.ServiceType == typeof(IExternalPaymentGateway));  
  
 // 2. Удаляем его, если он существует  
 if (descriptor!= null)  
 {  
 services.Remove(descriptor);  
 }  
  
 // 3. Регистрируем фейковую реализацию (Stub/Mock)  
 // Используем Singleton, чтобы иметь доступ к состоянию мока внутри теста  
 services.AddSingleton<IExternalPaymentGateway, FakePaymentGateway>();  
 });  
}

Этот механизм позволяет изолировать тестируемую систему от внешних ненадежных зависимостей (платежные шлюзы, отправка email, сторонние API), сохраняя при этом интеграционную природу теста для всех остальных компонентов (контроллеры, валидаторы, база данных).9

### 4.2. Стратегия мокирования в интеграционных тестах

Важно соблюдать баланс. В интеграционных тестах следует избегать мокирования внутренних компонентов системы (например, репозиториев или сервисов бизнес-логики). Если вы замокируете репозиторий, вы перестанете тестировать взаимодействие с базой данных, превратив интеграционный тест в медленный Unit-тест.

**Что нужно мокировать:**

* Внешние HTTP API (сервисы погоды, валют, аутентификации).
* Системные часы (IDateTimeProvider), чтобы тесты были детерминированными и не зависели от времени суток.
* Файловую систему (если это не является целью теста).
* Асинхронные процессы (очереди сообщений), если они выходят за рамки проверяемого сценария.

**Что нельзя мокировать:**

* Базу данных (используйте реальную БД или Testcontainers).
* Внутреннюю логику (сервисы, медиаторы).
* Систему логирования (лучше перенаправить логи в вывод теста для отладки).33

### 4.3. Аутентификация и авторизация: Паттерн TestAuthHandler

Большинство эндпоинтов в Enterprise-системах защищены атрибутом [Authorize]. Для выполнения запроса требуется валидный JWT-токен. В интеграционных тестах получение реального токена через внешний Identity Provider (например, Keycloak или Auth0) является медленной, ненадежной и сложной операцией.

Элегантным решением является использование кастомного AuthenticationHandler. Мы создаем обработчик, который игнорирует проверку подписи токена и всегда возвращает успешный результат аутентификации с заранее определенным набором утверждений (Claims).

C#

public class TestAuthHandler : AuthenticationHandler<AuthenticationSchemeOptions>  
{  
 public TestAuthHandler(IOptionsMonitor<AuthenticationSchemeOptions> options,   
 ILoggerFactory logger, UrlEncoder encoder, ISystemClock clock)   
 : base(options, logger, encoder, clock) { }  
  
 protected override Task<AuthenticateResult> HandleAuthenticateAsync()  
 {  
 var claims = new { new Claim(ClaimTypes.Name, "TestUser"), new Claim(ClaimTypes.Role, "Admin") };  
 var identity = new ClaimsIdentity(claims, "Test");  
 var principal = new ClaimsPrincipal(identity);  
 var ticket = new AuthenticationTicket(principal, "TestScheme");  
  
 return Task.FromResult(AuthenticateResult.Success(ticket));  
 }  
}

Регистрация этого хендлера в ConfigureTestServices позволяет "обмануть" систему авторизации, сосредоточившись на тестировании бизнес-логики, доступной авторизованному пользователю.9

## 5. Стратегия организации тестов и параллелизм

### 5.1. Управление параллелизмом в xUnit

Фреймворк xUnit по умолчанию запускает тесты параллельно для разных тестовых классов, но последовательно внутри одного класса. Это поведение отлично подходит для Unit-тестов, но создает проблемы для интеграционных тестов, использующих общие ресурсы, такие как база данных.

Если два теста одновременно пытаются записать данные в одну и ту же таблицу или очистить её, возникает состояние гонки (Race Condition), приводящее к падению тестов.

Для решения этой проблемы используется механизм коллекций (Collections).

1. Определяется класс-маркер коллекции с атрибутом ``.
2. Этот класс реализует интерфейс ICollectionFixture<WebApplicationFactory<Program>>.
3. Все тестовые классы помечаются атрибутом ``.

Это инструктирует xUnit запускать все тесты, входящие в данную коллекцию, последовательно. Это снижает общую скорость выполнения тестов, но гарантирует их стабильность и детерминизм при работе с одной базой данных.35

### 5.2. Паттерн "Arrange-Act-Assert" в контексте HTTP

Структура интеграционного теста должна строго следовать каноническому паттерну, адаптированному под специфику HTTP-взаимодействия:

1. **Arrange (Подготовка):**
   * Инициализация HTTP-клиента.
   * Подготовка данных в базе (Seed Data). **Критически важно:** данные должны вставляться напрямую через DbContext, минуя API. Если использовать API для подготовки данных (например, вызвать POST /create перед тестом GET /list), тест становится зависимым от работоспособности метода создания, что усложняет локализацию ошибки.
2. **Act (Действие):**
   * Выполнение HTTP-запроса к тестируемому эндпоинту (client.PostAsJsonAsync(...)).
3. **Assert (Проверка):**
   * Проверка HTTP-статуса (response.EnsureSuccessStatusCode()).
   * Десериализация и проверка тела ответа.
   * **Side-effect Assert (Проверка побочных эффектов):** Проверка состояния базы данных после выполнения запроса. Действительно ли запись появилась в таблице? Соответствуют ли данные сохраненным значениям? Этот этап отличает интеграционный тест от простого функционального теста API.9

## 6. Второй порядок инсайтов: Влияние на процесс разработки

Внедрение практики интеграционного тестирования через WebApplicationFactory имеет глубокие последствия, выходящие за рамки простого обеспечения качества.

### 6.1. Design for Testability (Проектирование для тестируемости)

Необходимость писать интеграционные тесты заставляет разработчиков менять стиль написания основного кода.

* Код становится чище: разработчики начинают избегать использования статических вызовов (например, DateTime.Now), так как их невозможно замокировать или предсказать в тесте. Вместо этого внедряются абстракции типа IClock.
* Уменьшение связанности: Жесткие зависимости от конкретных реализаций (например, прямая отправка email через SMTP-клиент) заменяются на интерфейсы, чтобы их можно было подменить в тестах.
* Отказ от "Магии" в конструкторах: Логика инициализации выносится из конструкторов, так как сложная инициализация мешает созданию тестовых экземпляров.

### 6.2. Живая документация

Интеграционные тесты служат отличной формой документации. В отличие от Swagger, который показывает схему данных, тест показывает *сценарий* использования. Глядя на тест, новый разработчик сразу понимает: "Ага, чтобы создать событие, мне нужно отправить такой-то JSON, и я получу 201 Created, а если я не укажу заголовок, то получу 400 Bad Request". Эта документация никогда не устаревает, так как устаревший тест просто не пройдет CI/CD пайплайн.

### 6.3. Shift-Left Testing

Интеграционные тесты позволяют обнаруживать проблемы конфигурации (например, забытая регистрация сервиса в DI или ошибка в appsettings.json) на этапе разработки, прямо на машине разработчика. Без них такие ошибки часто обнаруживаются только после деплоя на тестовый стенд (Staging), что значительно удлиняет цикл обратной связи (Feedback Loop).

## 7. Заключение

Третий день пятнадцатой недели обучения знаменует собой важную веху в профессиональном развитии разработчика. Переход к использованию WebApplicationFactory и отказ от упрощенных моделей (In-Memory DB) в пользу реалистичных сценариев тестирования — это шаг от написания кода к инженерии систем.

Интеграционное тестирование — это не просто поиск багов, это способ доказательства работоспособности системы как единого целого. Понимание того, что TestServer работает в памяти, обходя сетевой стек, и осознание ограничений баз данных в памяти позволяет писать тесты, которые действительно гарантируют качество, а не создают иллюзию безопасности.

Эти знания закладывают фундамент для следующих этапов обучения: работы с контейнеризацией (Docker), оркестрацией (Kubernetes) и построения сложных распределенных систем, где надежность и предсказуемость поведения являются главными валютами.

**Список ключевых концепций для углубленного изучения:**

* **Пространство имен:** Microsoft.AspNetCore.Mvc.Testing.
* **Интерфейсы:** IClassFixture<T>, ICollectionFixture<T>, IAsyncLifetime.
* **Методы:** ConfigureTestServices, EnsureSuccessStatusCode.
* **Сетевые нюансы:** Различия между SocketsHttpHandler и TestServer ClientHandler.
* **Анти-паттерны:** Sync-over-Async в тестах (.Result, .Wait()), использование статического состояния (static).

#### Источники

1. План обучения: Docker и Тестирование неделя 11
2. The Practical Test Pyramid - Martin Fowler, дата последнего обращения: декабря 6, 2025, <https://martinfowler.com/articles/practical-test-pyramid.html>
3. Understanding the Testing Pyramid and Testing Trophy: Tools, Strategies, and Challenges, дата последнего обращения: декабря 6, 2025, <https://dev.to/craftedwithintent/understanding-the-testing-pyramid-and-testing-trophy-tools-strategies-and-challenges-k1j>
4. Pyramid or Crab? Find a testing strategy that fits | Articles - web.dev, дата последнего обращения: декабря 6, 2025, <https://web.dev/articles/ta-strategies>
5. Your Most Comprehensive Guide for Modern Test Pyramid in 2025 - Full Scale, дата последнего обращения: декабря 6, 2025, <https://fullscale.io/blog/modern-test-pyramid-guide/>
6. The Testing Trophy and Testing Classifications - Kent C. Dodds, дата последнего обращения: декабря 6, 2025, <https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications>
7. What is your definition of a Unit, Integration, and End-to-End Test? : r/programming - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/programming/comments/1cbdmg5/what_is_your_definition_of_a_unit_integration_and/>
8. WebApplicationFactory
9. Integration tests in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-10.0>
10. Different kind of testing strategies with ASP.NET Core: The basics, дата последнего обращения: декабря 6, 2025, <https://dotnetfromthemountain.com/aspnet-core-testing-strategies-the-basics/>
11. Web Application Factory VS Real HTTP Calls - Take Two - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/maxarshinov/web-application-factory-vs-real-http-calls-take-two-2a1k>
12. Difference in speed for http and tcp. Why? : r/AskProgramming - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/AskProgramming/comments/1noq5v1/difference_in_speed_for_http_and_tcp_why/>
13. TestServer Class (Microsoft.AspNetCore.TestHost), дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.testhost.testserver?view=aspnetcore-10.0>
14. TestServer exception handling differs from Kestrel/regular hosted exception handling · Issue #27208 · dotnet/aspnetcore - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/aspnetcore/issues/27208>
15. Using full Kestrel server instead of the test server? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/76595582/using-full-kestrel-server-instead-of-the-test-server>
16. Attention when working with the Asp.Net Core TestHost and its HttpMessageHandler, дата последнего обращения: декабря 6, 2025, <https://blog.dangl.me/archive/attention-when-working-with-the-aspnet-core-testhost-and-its-httpmessagehandler/>
17. Avoid shared memory between ASP.NET Core integration tests - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/78421394/avoid-shared-memory-between-asp-net-core-integration-tests>
18. Memory management and garbage collection (GC) in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/performance/memory?view=aspnetcore-10.0>
19. Avoid In-Memory Databases for Tests - Jimmy Bogard, дата последнего обращения: декабря 6, 2025, <https://www.jimmybogard.com/avoid-in-memory-databases-for-tests/>
20. EF Core In-Memory Database Provider - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/providers/in-memory/>
21. Unit Tests vs Integration tests: Entity Framework Core In memory - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/47056392/unit-tests-vs-integration-tests-entity-framework-core-in-memory>
22. TDD: Does EF Core In Memory Provider validates referential constraints? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/50999975/tdd-does-ef-core-in-memory-provider-validates-referential-constraints>
23. Entity Framework core - Contains is case sensitive or case insensitive? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/43277868/entity-framework-core-contains-is-case-sensitive-or-case-insensitive>
24. Collations and Case Sensitivity | Npgsql Documentation, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/efcore/misc/collations-and-case-sensitivity.html>
25. Testing without your Production Database System - EF Core | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/testing/testing-without-the-database>
26. Any ILIKE linq expression could not be translated in EF Core 3.1 using InMemoryDatabase · Issue #1159 · npgsql/efcore.pg - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/npgsql/efcore.pg/issues/1159>
27. Using InMemoryDb with Npgsql · Issue #774 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/npgsql/efcore.pg/issues/774>
28. NpgsqlTsVector doesn't work using EF In Memory Database - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/77806506/npgsqltsvector-doesnt-work-using-ef-in-memory-database>
29. Testing concurrency tokens with Microsoft.EntityFrameworkCore.InMemory - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/46997462/testing-concurrency-tokens-with-microsoft-entityframeworkcore-inmemory>
30. ASP.NET CORE Integration tests: API - Float In Code, дата последнего обращения: декабря 6, 2025, <https://floatincode.net/2021/03/29/asp-net-core-integration-tests-api/>
31. Overriding configuration in ASP.NET Core integration tests - Mark Vincze, дата последнего обращения: декабря 6, 2025, <https://blog.markvincze.com/overriding-configuration-in-asp-net-core-integration-tests/>
32. Net Core: Override WebApplicationFactory Services DbContext with InMemory Database, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/57385165/net-core-override-webapplicationfactory-services-dbcontext-with-inmemory-databa>
33. Mocking in integration tests. : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/16pthnq/mocking_in_integration_tests/>
34. Mock authentication with WebApplicationFactory not working - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/69315988/mock-authentication-with-webapplicationfactory-not-working>
35. xUnit - Using WebApplicationFactory with CollectionFixture for .Net Core Integration Tests, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/78318084/xunit-using-webapplicationfactory-with-collectionfixture-for-net-core-integra>
36. Running Tests in Parallel - xUnit.net, дата последнего обращения: декабря 6, 2025, <https://xunit.net/docs/running-tests-in-parallel>
37. Integration Testing with xUnit - Jimmy Bogard, дата последнего обращения: декабря 6, 2025, <https://www.jimmybogard.com/integration-testing-with-xunit/>