

# Тестируем код, взаимодействующий с базой данных



Гурий  
Самарин  
Росатом

DOTNEXT

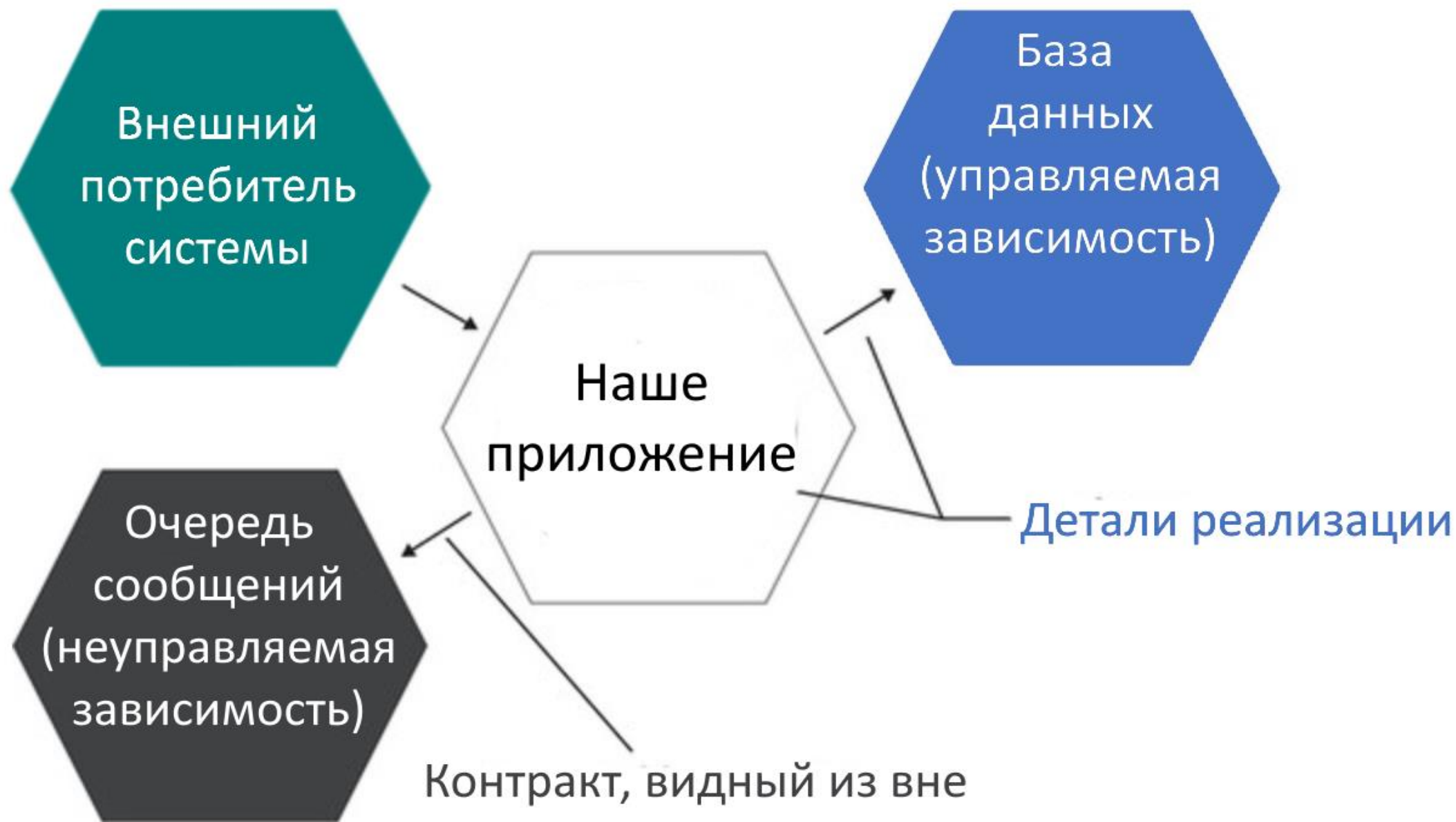
# План доклада

- Введение или о типах зависимостей
- Хранение схемы и доставка изменений
- Изоляция тестов
- Библиотеки
  - Respawn
  - EfCore.TestSupport
  - Testcontainers-dotnet
- Рекомендации

# Какие зависимости мы тестируем?

**Управляемые зависимости** - внепроцессные зависимости, над которыми мы имеем полный контроль.

**Неуправляемые зависимости** - внепроцессные зависимости, взаимодействие с которыми можно наблюдать извне.



# Если база неуправляемая

- Таблицы, которые видны другим – неуправляемые.
- Остальные – управляемые.

# Тестирование на production базе

## ***Минусы:***

- Сложно сделать идентично production.
- Медленнее.

## ***Плюсы:***

- Поведение как на бою.
- Тестируем ровно то, что надо.

# Если невозможно использовать продбазу

Возможно ли такое?

Будем использовать моки?

# Нельзя тестировать базу as is – не тестируй

Сосредоточимся на ***unit*** тестировании предметной области.



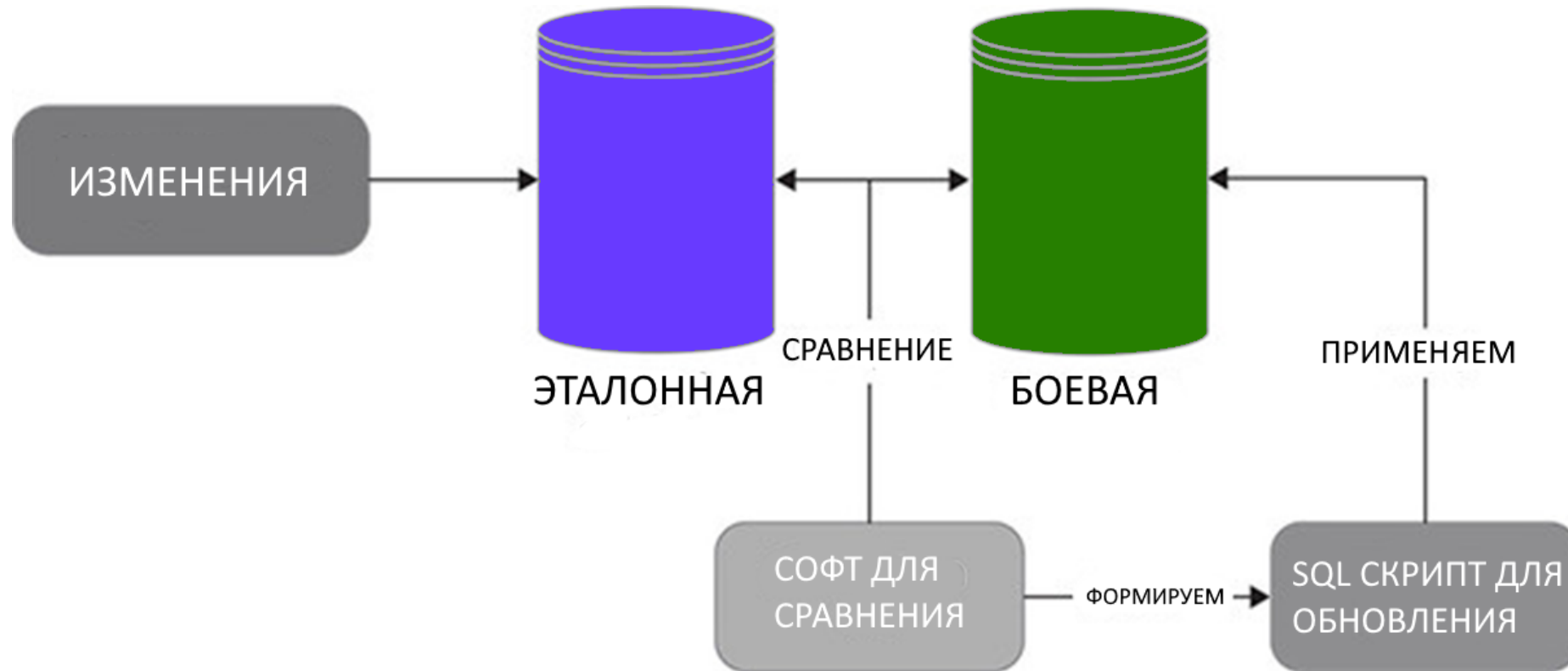
# Что мы уже узнали?

- ✓ Введение или о типах зависимостей
  - База данных – управляемая зависимость, а значит деталь реализации.
  - Необходимо выделять управляемую часть из неуправляемого взаимодействия с базой.
  - Тестируем в продбазе (**на другом экземпляре!**).

# Как хранить схему бд?

- Отдельно, например в виде эталонной базы.
- В системе управления версиями.

# Антипаттерн: эталонная база



# Недостатки подхода

- Нет истории изменений.
- Нет единого источника истины.

# Что такое схема базы данных?

Не только DDL, но и данные, необходимые для правильной работы приложения.

# Референс-данные и мастер-данные

**Референс-данные** – это относительно редко меняющиеся данные, которые определяют значения конкретных сущностей.

**Мастер-данные** – это базовые данные, которые определяют бизнес-сущности, с которыми имеет дело предприятие.

# Как их выделить?

Существует простой способ отличить **референс-мастер данные** от обычных данных.

Если ваше приложение может изменять данные, то это обычные данные; если нет, то это справочные данные.

# Референс-данные – пример в коде

```
class MeasureUnitConfiguration : IEntityTypeConfiguration<MeasureUnitRecord>
{
    public void Configure(EntityTypeBuilder<MeasureUnitRecord> builder)
    {
        builder.ToTable("UnitOfMeasurement", Constants.Schema);

        builder.HasKey(b => b.Id);

        ... ..

        builder.HasData(
            new { Id = 1, Name = "ч/ч", Key = "mh" },
            new { Id = 2, Name = "шт", Key = "pcs" }
        );
    }
}
```



# Референс-данные – пример в миграции

```
public partial class SeedMeasureUnits : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.InsertData(
            schema: "dbo",
            table: "UnitOfMeasurement",
            columns: new[] { "Id", "Key", "Name" },
            values: new object[,]
            {
                { 1, "mh", "ч/ч" },
                { 2, "pcs", "шт" }
            }
        );
    }
}
```

# Как их хранить?

В форме инструкций SQL INSERT

```
DO $EF$
```

```
BEGIN
```

```
    IF NOT EXISTS(SELECT 1 FROM "__EFMigrationsHistory" WHERE "MigrationId" =  
'20210914125325_SeedMeasureUnits') THEN
```

```
        INSERT INTO dbo."UnitOfMeasurement" ("Id", "Key", "Name")
```

```
        VALUES (1, 'mh', 'ч/ч');
```

```
        INSERT INTO dbo."UnitOfMeasurement" ("Id", "Key", "Name")
```

```
        VALUES (2, 'pcs', 'шт');
```

```
    END IF;
```

```
END $EF$;
```

# Хранение схемы или изменений схемы

- Состояние (snapshot).
- Миграции.

## ***Правило:***

Не редактируйте и не удаляйте миграции.


# Merge конфликты VS трансформации данных

***Трансформация данных*** - это изменение формы данных, чтобы они соответствовали новой схеме.

# Классический пример


При разделении столбца **Name** на **FirstName** и **LastName** вам нужно не только удалить столбец **Name** и создать новые столбцы **FirstName** и **LastName**, но также написать скрипт для разделения всех существующих имен на две части.


# DBeaver

 Schema Compare

**Preview results of migration/compare**  
Here you can review, include or exclude changes in generated diff change sets

Diff type: Changes tree

▼  public

>  attendee Modify Table

Report type: DDL

```
ALTER TABLE "public"."attendee" ADD "first_name" TEXT;  
  
ALTER TABLE "public"."attendee" ADD "last_name" TEXT;  
  
ALTER TABLE "public"."attendee" DROP COLUMN "name";
```

AllNoneRefresh ReportShow log

SaveCopyOpen in EditorMigrate

# DataGrip

The screenshot displays the DataGrip Migration interface. At the top, the 'Origin' is set to '[postgres@localhost] development' and the 'Target' is '[postgres@localhost] production'. The 'Show identical' checkbox is unchecked. The interface shows a comparison between two database schemas. The 'Origin' schema (left) has a table 'attendee' with columns 'id' (integer, auto increment), 'first\_name' (text), and 'last\_name' (text). The 'Target' schema (right) has a table 'attendee' with columns 'id' (integer, auto increment), 'first\_name' (text), 'last\_name' (text), and 'name' (text). A large arrow points from the 'Origin' to the 'Target' schema. Below the schema comparison, the 'Script Preview' tab is active, showing the following SQL script:

```
alter table attendee
add first_name text;

alter table attendee
add last_name text;

alter table attendee
drop column name;
```

At the bottom of the window, there are buttons for '?', 'Execute', 'Cancel', and 'Open in New Query Console'.

# Devart

```
-- Drop column "name" from table "public"."attendee"
```

```
ALTER TABLE public.attendee  
    DROP COLUMN name;
```

```
-- Create column "first_name" on table "public"."attendee"
```

```
ALTER TABLE public.attendee  
    ADD first_name text;
```

```
-- Create column "last_name" on table "public"."attendee"
```

```
ALTER TABLE public.attendee  
    ADD last_name text;
```



# Отложим миграции до production

Потеря тестовых данных не является проблемой - можно создавать их заново каждый раз.

# Хранение и автоматизация миграций

- SQL скрипты.
- EF миграции.
- Flyway.
- Liquibase.

# Liquibase минимально

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:pro="http://www.liquibase.org/xml/ns/pro"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-4.1.xsd
http://www.liquibase.org/xml/ns/pro
http://www.liquibase.org/xml/ns/pro/liquibase-pro-4.1.xsd">
  <includeAll path="Migrations"/>
</databaseChangeLog>
```

# Liquibase побольше

```
<?xml version="1.0" encoding="UTF-8"?>
  <databaseChangeLog ...>
    <changeSet author="lb-generated" id="1185214997195-1">
      <createTable name="BONUS">
        <column name="NAME" type="VARCHAR2(15)" />
        <column name="JOB" type="VARCHAR2(255)" />
        <column name="SAL" type="NUMBER(255)" />
      </createTable>
    </changeSet>
  </databaseChangeLog>
```

# Что мы уже узнали?

## ✓ Введение или о типах зависимостей

- База данных – управляемая зависимость, а значит деталь реализации.
- Необходимо выделять управляемую часть из неуправляемого взаимодействия с базой.
- Тестируем в продбазе (**на другом экземпляре!**).

## ✓ Хранение схемы и доставка изменений

- Используйте миграции.
- Не изменяйте миграции. Создавайте новую.
- Накатывайте миграции специальными инструментами.

# Отдельный экземпляр для каждого

- Тесты, выполняемые разными разработчиками, мешают друг другу.
- Обрато несовместимые изменения могут блокировать работу других разработчиков.

# Контроль за состоянием базы в тестах

- Изолируем тестовые инстансы базы.
- Обеспечиваем их предсказуемое наполнение.

# Управление жизненным циклом данных

- Выполнение интеграционных тестов последовательно.
- Удаление оставшихся данных между тестовыми запусками.
  - В ходе выполнения теста, а не сбоку.



# Параллельное vs последовательное

Параллельное выполнение интеграционных тестов требует значительных усилий.

xUnit и NUnit – позволяют создать отдельные тестовые коллекции и отключать в них распараллеливание.

# Коллекции в xUnit

```
[CollectionDefinition(nameof(NotThreadSafe), DisableParallelization = true)]  
public class NotThreadSafe { }
```

```
[Collection(nameof(NotThreadSafe))]  
public class TestClass1  
{  
    [Fact]  
    public void Test1() => ...;  
}
```

# Очистка между тестовыми запусками

Четыре варианта очистки оставшихся данных между тестовыми запусками:

- Восстановление резервной копии базы данных перед каждым тестированием.
- Оборачивание каждого теста в транзакцию.
- Очистка данных в конце теста.
- Очистка данных в начале теста.

# Восстановление из резервной копии

- Если база данных меняется не часто.
- Если можно брать базу из production.

# Запускаем тест в транзакции и Rollback

Атрибуты BeforeAfterTest (xUnit, а лучше ctor и Dispose) или SetUp и TearDown (NUnit) для открытия транзакции и ее отката.

## ***Недостаток:***

- Дополнительная транзакция задает отличное от production поведение.

# AutoRollback

```
[Fact]
[AutoRollback]
public void AutoRollback()
{
    using SqlConnection connection = new(connectionString);
    connection.Open();

    SqlCommand command = new("DELETE FROM Customers", connection);
    command.ExecuteNonQuery();
}
```

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
```

```
public sealed class AutoRollbackAttribute : BeforeAfterTestAttribute
{
    TransactionScope scope;

    public TransactionScopeAsyncFlowOption AsyncFlowOption {get; set;} = Enabled;
    public IsolationLevel IsolationLevel {get; set;} = Unspecified;
    public TransactionScopeOption ScopeOption {get; set;} = Required;
    public long TimeoutInMS {get; set;} = -1;

    public override void After(MethodInfo methodUnderTest) => scope.Dispose();

    public override void Before(MethodInfo methodUnderTest)
    {
        TransactionOptions options = new (){ IsolationLevel = IsolationLevel };
        if (TimeoutInMS > 0) options.Timeout = TimeSpan.FromMilliseconds(TimeoutInMS);
        scope = new TransactionScope(ScopeOption, options, AsyncFlowOption);
    }
}
```

# Очистка в начале или в конце

Очистка данных в начале теста — это лучший вариант.



# Что мы уже узнали?

- ✓ Введение или о типах зависимостей
  - База данных – управляемая зависимость, а значит деталь реализации.
  - Необходимо выделять управляемую часть из неуправляемого взаимодействия с базой.
  - Тестируем в продбазе (**на другом экземпляре!**).
- ✓ Хранение схемы и доставка изменений
  - Используйте миграции.
  - Не изменяйте миграции. Создавайте новую.
  - Накатывайте миграции специальными инструментами.
- ✓ Изоляция тестов
  - У каждого разработчика должна быть своя база.
  - Очищайте данные перед каждым тестом.

# Как же очищать данные?

В базовый класс для интеграционных тестов помещаем сценарий удаления. И тогда

- Либо отключаем все внешние ключи, очищаем каждую таблицу и восстанавливаем внешние ключи.
- Либо находим “правильный” порядок удаления данных на основе взаимосвязей и удаляем данные из каждой таблицы в этом порядке.

# “Правильный” порядок удаления данных

- Наиболее эффективное решение.
- Наиболее сложное в реализации.

# Respawn by J. Bogard

- Построение ориентированного графа по внешним ключам.
- Обход ориентированного графа в порядке, в котором мы удаляем таблицы.
- В случае цикла отключаем ограничения только в нем с последующим удалением.

# Пример использования

```
var checkpoint = await Respawner.CreateAsync(_connection,  
new RespawnerOptions  
    {  
        DbAdapter = DbAdapter.Postgres,  
        TablesToIgnore = new Table[] { "foo" }  
    });  
await checkpoint.ResetAsync(_connection);
```

# EfCore.TestSupport by J.P.Smith

Рассмотрим 2 сценария

- Та же база данных, что в production.
- SQLite в памяти.

	как в production	SQLite in-memory
+	<ul style="list-style-type: none"> <li>· все как в production</li> <li>· полная поддержка SQL</li> </ul>	<ul style="list-style-type: none"> <li>· быстрый запуск</li> <li>· актуальная схема</li> <li>· стартует пустой</li> </ul>
-	<ul style="list-style-type: none"> <li>· нужны уникальные экземпляры db на тест</li> <li>· медленно создается/очищается</li> </ul>	<ul style="list-style-type: none"> <li>· частичная поддержка SQL</li> <li>· нюансы относительно production</li> </ul>
когда нужно	используется SQL	используется только LINQ

# SQLite – шустрый, но ограниченный

## **Плюсы:**

- Схема базы данных всегда актуальна.
- Не требует очистки.
- Параллельное выполнение.
- Успешно выполняться в любом pipeline'е.

## **Минусы:**

- Ограниченная поддержка типов.
- Идемпотентный скрипт миграции не создать.



# SQLite in-memory

- Строка подключения "Filename=:memory:"
- Статический метод `SqliteInMemory.CreateOptions<TContext>` из `EFCore.TestSupport`

```
[Fact]
public void TestSqliteInMemoryOk()
{
    //SETUP
    var options = SqliteInMemory.CreateOptions<BookContext>();
    using var context = new BookContext(options);

    context.Database.EnsureCreated();

    //...
}
```

# Kak production

```
[Fact]
public void TestEnsureDeletedEnsureCreatedOk()
{
    //SETUP
    var options = this.CreateUniqueClassOptions<BookContext>();
    using var context = new BookContext(options);

    context.Database.EnsureDeleted();
    context.Database.EnsureCreated();

    //...
}
```

# EnsureDeleted + EnsureCreated. А быстрее?

Для SQL Server и PostgreSQL есть метод EnsureClean

[Fact]

```
public void TestSqlDatabaseEnsureCleanOk()
{
    //SETUP
    var options = this.CreateUniqueClassOptions<BookContext>();
    using var context = new BookContext(options);

    context.Database.EnsureClean();

    //...
}
```

# EnsureClean

```
public static void EnsureClean(this DatabaseFacade databaseFacade, bool
setUpSchema = true)
{
    if (databaseFacade.IsSqlServer())//SQL Server
        databaseFacade.CreateExecutionStrategy()
            .Execute(databaseFacade, database => new
SqlServerDatabaseCleaner(databaseFacade).Clean(database, setUpSchema));
    else if (databaseFacade.IsNpgsql())//PostgreSQL
        databaseFacade.FasterPostgreSqlEnsureClean(setUpSchema);
    else
        throw new InvalidOperationException("The EnsureClean method only
works with SQL Server or PostgreSQL databases.");
}
```

# Почему в тесте не как в production?

Например, из-за Identity Resolution в EF.

# Неправильный тест

```
[Fact]
public void ExampleIdentityResolutionBad()
{
    //ARRANGE
    var options = SqliteInMemory.CreateOptions<EfCoreContext>();
    using var context = new EfCoreContext(options);
    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();
    //ACT
    var book = context.Books.First();
    book.Price = 123;
    // Забыли вызвать context.SaveChanges()
    //ASSERT
    context.Books.First().Price.ShouldEqual(123); //В базе другая цена
}
```

# Правильный тест

```
[Fact]
public void UsingThreeInstancesOfTheDbcontext()
{
    var options = SqliteInMemory.CreateOptions<EfCoreContext>();
    options.StopNextDispose();
    using (var context = new EfCoreContext(options))
    {    //ARRANGE instance    }
    options.StopNextDispose();
    using (var context = new EfCoreContext(options))
    {    //ACT instance    }
    using (var context = new EfCoreContext(options))
    {    //ASSERT instance    }
}
```

# ChangeTracker.Clear()

[Fact]

```
public void UsingChangeTrackerClear()
{
    //ARRANGE
    using var context = new EfCoreContext(SqliteInMemory.CreateOptions<EfCoreContext>());
    context.Database.EnsureCreated();
    var setupBooks = context.SeedDatabaseFourBooks();
    //ACT
    context.ChangeTracker.Clear();
    var book = context.Books.Include(b => b.Reviews)
        .Single(b => b.BookId = setupBooks.Last().BookId);
    book.Reviews.Add(new Review { NumStars = 5 });
    context.SaveChanges();
    //ASSERT
    context.ChangeTracker.Clear();
    context.Books.Include(b => b.Reviews).Single(b => b.BookId = setupBooks.Last().BookId)
        .Reviews.Count.ShouldEqual(3);
}
```



# Где взять лучшие тестовые данные?

Сериализуем данные из production базы данных и сохраняем в JSON.

# Seed from Production

- Делает snapshot базы в JSON.
- Воссоздает их в тестовой базе.
- Деперсонифицирует данные.

# Создание базы в контейнере

- Помещаем базу данных в образ Docker или накатываем схему при старте.
- Создаем новый экземпляр контейнера из этого образа для каждого интеграционного теста.

# Правила запуска в Docker

- Для каждого теста запускаем в отдельный контейнер.
- Запускаем тесты параллельно, но группами.
- Останавливаем и удаляем использованные контейнеры.

# Testcontainers-dotnet

- Легкие, временные экземпляры баз в Docker'е.
- API для автоматизации настройки окружения.

```
PostgreSqlTestcontainerConfiguration postgresConfiguration = new ()
{
    Username = "postgres",
    Password = Guid.NewGuid().ToString("D"),
    Database = Guid.NewGuid().ToString("D"),
    Port = 5432
};
```

```
PostgreSqlTestcontainer postgresContainer = new
TestcontainersBuilder<PostgreSqlTestcontainer>()
    .WithDatabase(postgresConfiguration)
    .Build();
```

```
await postgresContainer.StartAsync();
```

# Docker.DotNet: тестируем в докере

## ***Минус:***

- Множество нюансов уже реализованных в готовых инструментах.

## ***Плюс:***

- Полное управление.

# Получаем список контейнеров

```
// получаем список контейнеров  
var contList = await dockerClient  
    .Containers.ListContainersAsync(new  
        ContainersListParameters { All = true });
```



# Создаем контейнер

```
var postgresContainer = await dockerClient.Containers
    .CreateContainerAsync(new CreateContainerParameters
    {
        Name = _dbContainerName,
        Image = DbImage,
        Env = Env,
        HostConfig = new HostConfig
        {
            PortBindings = new Dictionary<string,
                IList<PortBinding>>{{ PortInContainer, new[] { new PortBinding {
                    HostPort = freePort } } } }
        }
    });
```

# Запускаем контейнер и ждем доступности

```
await dockerClient.Containers
    .StartContainerAsync(
        sqlContainer.ID,
        new ContainerStartParameters());
```

...

```
await WaitUntilDatabaseAvailableAsync(connection);
```

# Что мы уже узнали?

- ✓ Введение или о типах зависимостей
  - База данных – управляемая зависимость, а значит деталь реализации.
  - Необходимо выделять управляемую часть из неуправляемого взаимодействия с базой.
  - Тестируем в продбазе (**на другом экземпляре!**).
- ✓ Хранение схемы и доставка изменений.
  - Используйте миграции.
  - Не изменяйте миграции. Создавайте новую.
  - Накатывайте миграции специальными инструментами.
- ✓ Изоляция тестов
  - У каждого разработчика должна быть своя база.
  - Очищайте данные перед каждым тестом.
- ✓ Библиотеки
  - Respawn.
  - EfCore.TestSupport.
  - Testcontainers-dotnet.

# Следует ли тестировать чтения?

Тестируем только самые сложные или важные операции чтения.

# Тестирование чтения

- Unit тесты бесполезны.
- В случае необходимости используйте интеграционные.

# Следует ли тестировать репозитории?

Тестировать ли репозиторий дополнительно к интеграционным тестам?

# Тестирование репозитория

- Высокие затраты на поддержку.
- Нет преимуществ перед обычными интеграционными тестами.
- Лучший способ - извлечь алгоритм и тестировать его.

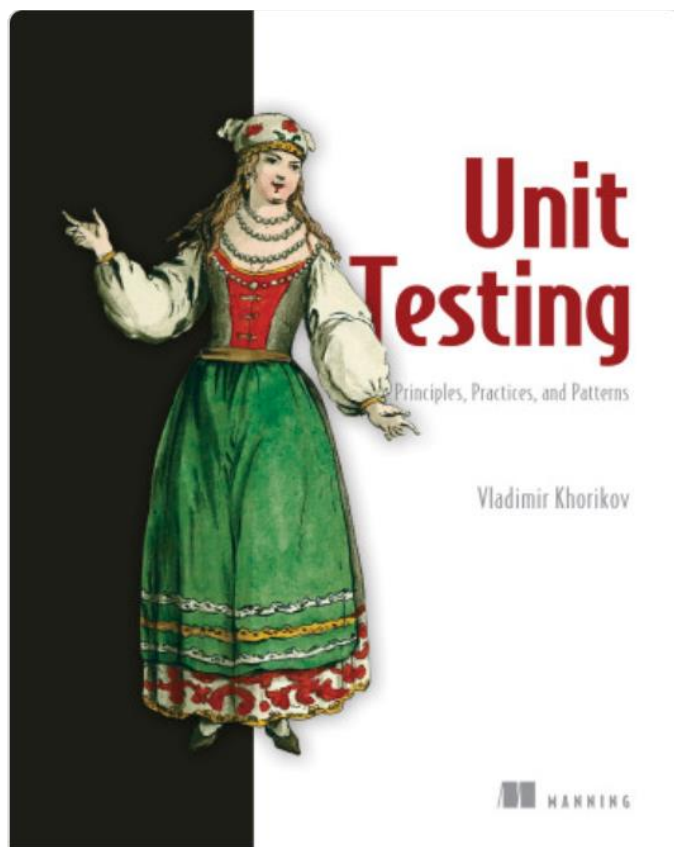
# Что мы уже узнали?

- ✓ Введение или о типах зависимостей
  - База данных – управляемая зависимость, а значит деталь реализации.
  - Необходимо выделять управляемую часть из неуправляемого взаимодействия с базой.
  - Тестируем в продбазе (**на другом экземпляре!**).
- ✓ Хранение схемы и доставка изменений
  - Используйте миграции.
  - Не изменяйте миграции. Создавайте новую.
  - Накатывайте миграции специальными инструментами.
- ✓ Изоляция тестов
  - У каждого разработчика должна быть своя база.
  - Очищайте данные перед каждым тестом.
- ✓ Библиотеки
  - Respawn.
  - EfCore.TestSupport.
  - Testcontainers-dotnet.
- ✓ Рекомендации
  - Не тестируйте чтения.
  - Не пишите модульные тесты на репозитории.



# Ссылки

## *Unit Testing Principles, Practices, and Patterns V. Khorikov*



<https://www.manning.com/books/unit-testing>

*перевод*

<https://www.labirint.ru/books/777259/>

# ССЫЛКИ

## *Respawn*

<https://github.com/jbogard/respawn>

<https://lostechies.com/jimmybogard/2013/06/18/strategies-for-isolating-the-database-in-tests/>

<https://jimmybogard.com/how-respawn-works>

# ССЫЛКИ

## *EfCore.TestSupport*

<https://www.thereformedprogrammer.net/new-features-for-unit-testing-your-entity-framework-core-5-code/>

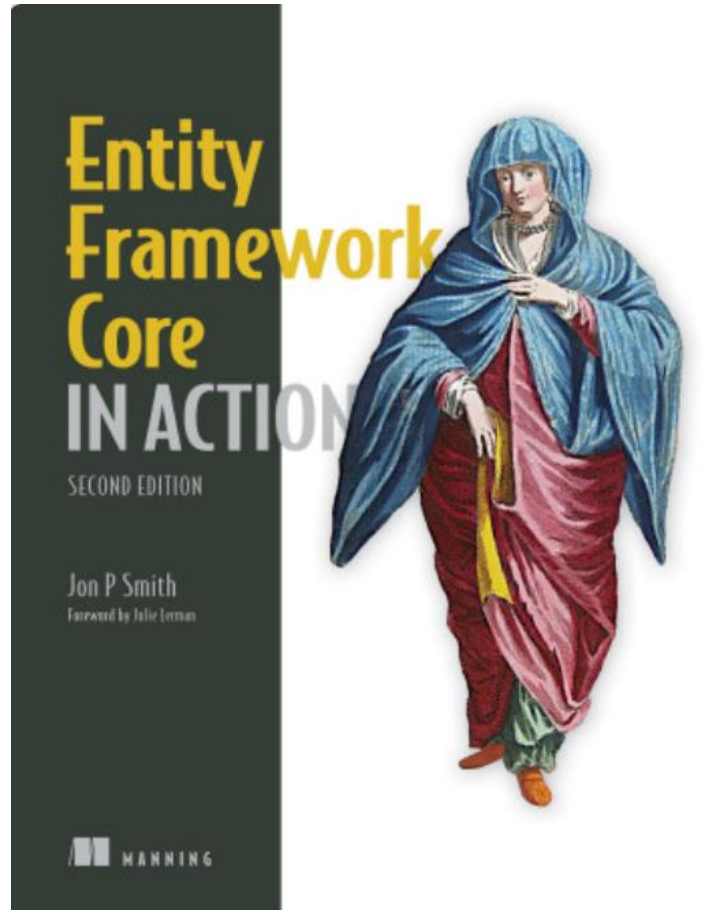
<https://www.thereformedprogrammer.net/getting-better-data-for-unit-testing-your-ef-core-applications/>

<https://github.com/JonPSmith/EfCore.TestSupport>

<https://github.com/JonPSmith/EfCore.TestSupport/wiki/Using-SQLite-in-memory-databases>

<https://www.thereformedprogrammer.net/using-postgresql-in-dev-part-2-testing-against-a-postgresql-database/>

# Entity Framework Core in Action, Second Edition Jon P. Smith



<https://www.manning.com/books/entity-framework-core-in-action-second-edition>

***перевод***

<https://habr.com/ru/company/jugru/blog/691664/>

# Ссылки для контейнеров

- <https://github.com/testcontainers/testcontainers-dotnet>
- xUnit:<https://blog.dangl.me/archive/running-sql-server-integration-tests-in-net-core-projects-via-docker/>
- NUnit:<https://wrapt.dev/blog/integration-tests-using-sql-server-db-in-docker>

«Давайте сюда ваши ответы» (с)

# Контакты



<https://github.com/Sa1Gur>



@guriy\_samarin



@guriy\_samarin

