

Тестируем код, взаимодействующий с базой данных



Гурий
Самарин
Росатом

DOTNEXT

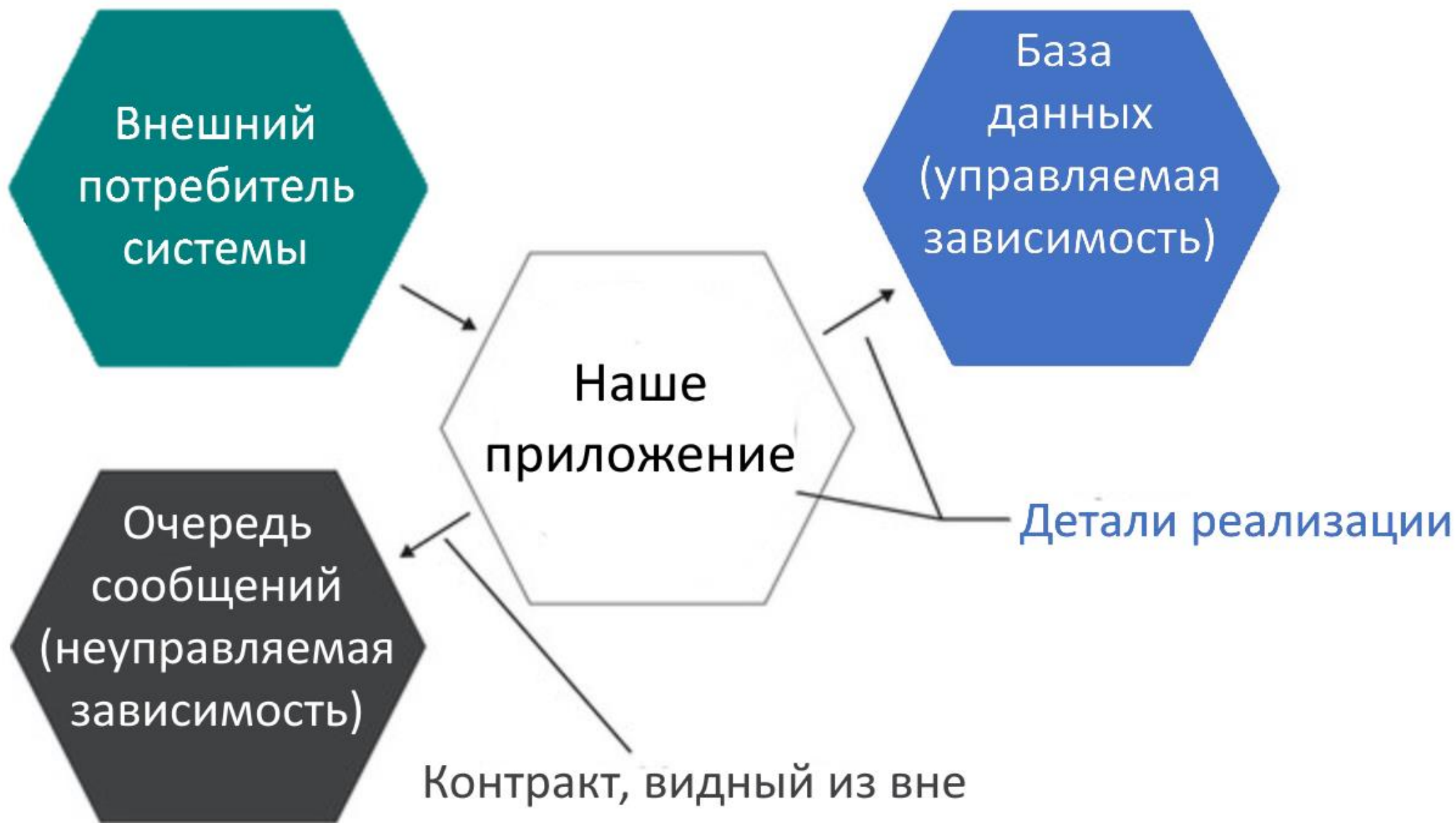
План доклада

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

Какие зависимости мы тестируем?

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

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



Тестирование как на production

Минусы:

- Сложно сделать идентично production
- Медленнее

Плюсы:

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

Если невозможно использовать продбазу

Возможно ли, что вы не можете использовать реальную базу данных в интеграционных тестах?

Должны ли вы в любом случае имитировать базу данных, несмотря на то, что это управляемая зависимость?

Нельзя тестировать базу as is – не тестируй

Сосредоточьтесь исключительно на модульном тестировании модели предметной области

Если база неуправляемая

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

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!**)

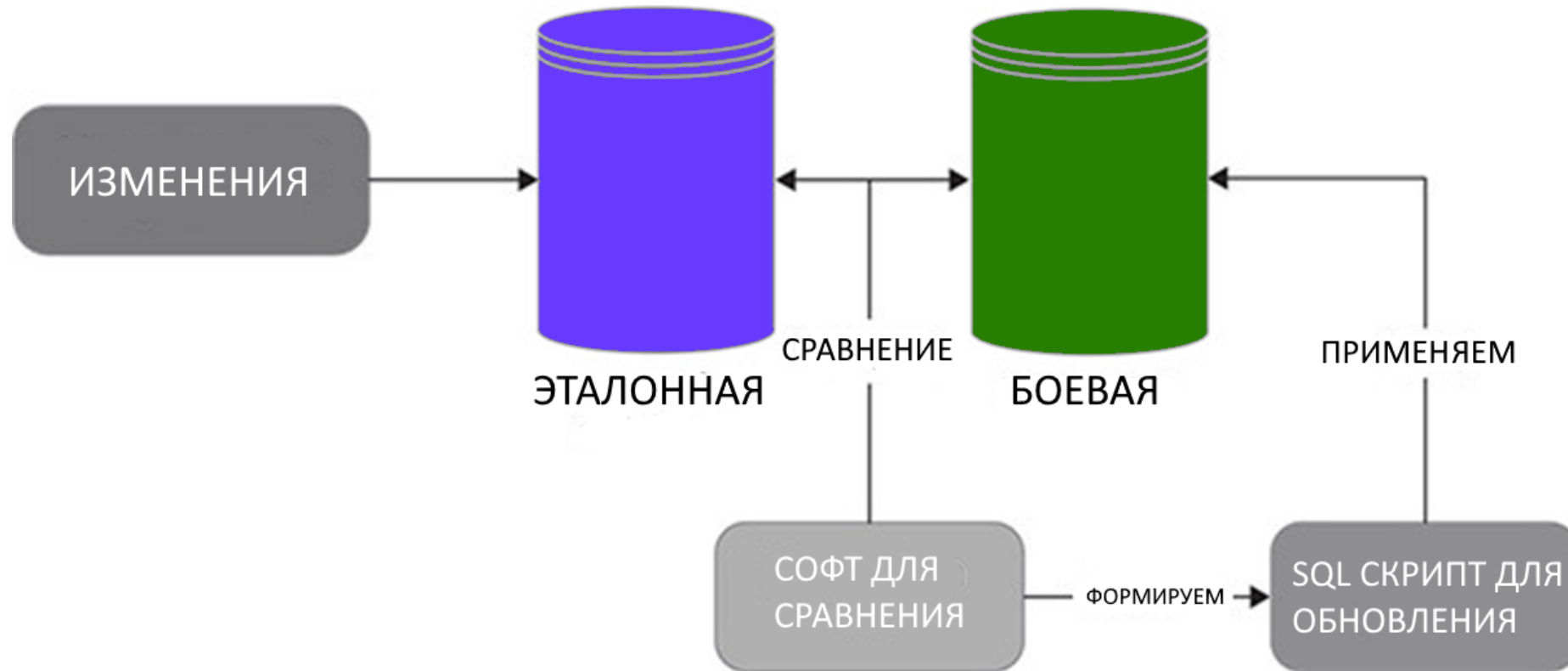
Как хранить схему бд?

- Храним схему базы данных в системе управления версиями
- Применяем миграции для изменения схемы

Схема базы в git'e

Первым шагом на пути к тестированию базы данных является работа со схемой базы данных как с обычным кодом – хранение ее 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

Подход, основанный на состоянии

У вас есть SQL-скрипты, которые вы можете использовать для создания базы данных. Скрипты хранятся в системе управления версиями.

Подход, основанный на миграции

Использование явных миграций, которые переводят базу данных из одной версии в другую.

Не редактируйте и не удаляйте миграции.

Миграции vs состояния

	состояние базы данных	скрипт миграции
подход с хранением состояния	 явное	 неявное
подход на миграциях	 неявное	 явное

Merge конфликты VS трансформации данных

Трансформация данных - это процесс изменения формы существующих данных таким образом, чтобы они соответствовали новой схеме базы данных.

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

Compare objects

Compare database objects

Settings of objects compare

Objects

Name	Type	Fully qualified name	
 production	Database	production	
 development	Database	development	

Compare settings

☒ Skip system objects

☐ Compare expensive properties

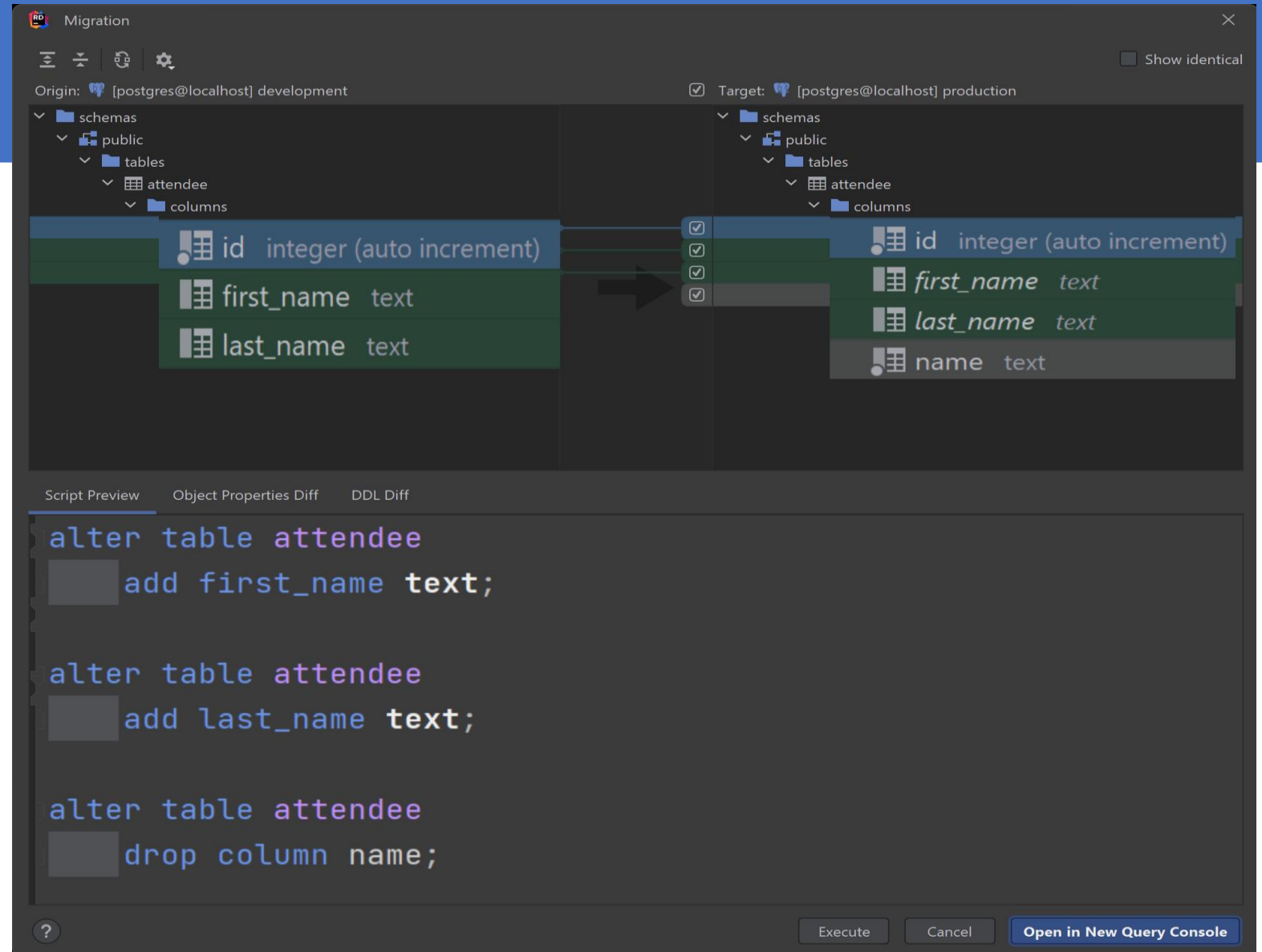
☐ Compare only structure (ignore properties)

☐ Compare scripts/procedures

364 objects compared

Structure	production	development
Database	production	development
Name	production	development
Allow Connect	false	true
Connection Limit	0	-1
Schemas	Schemas	Schemas
Schema	public	public
Tables	Tables	Tables
Table	attendee	attendee
Object ID	16399	16386
Columns	Columns	Columns
Column	name	N/A
Column	N/A	first_name
Column	N/A	last_name
Sequences	Sequences	Sequences
Sequence	attendee_id_seq	N/A

DataGrip



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 [<https://flywaydb.org>]
- Liquibase [<https://liquibase.org>]

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. Создайте новую
 - Исключения – возможная потеря данных

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.
- Обеспечиваем предсказуемое наполнение.

Управление жизненным циклом данных

- Выполнение интеграционных тесты последовательно.
- Удаление оставшихся данных между тестовыми запусками.
- Приведение базы к начальному состоянию в самих тестах.

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

- В нашем тесте мы можем использовать расширения Setup/TearDown или Before/AfterTest для открытия внешней транзакции и последующего ее отката.
- Одним из побочных эффектов этого является то, что, поскольку наша транзакция автоматически откатывается, если нам нужно отладить наши тестовые данные после запуска теста, мы не можем этого сделать, поскольку данные исчезли.

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. Создайте новую
 - Исключения – возможная потеря данных
- ✓ Изоляция тестов
 - У каждого разработчика должна быть своя база
 - Очищаем данные перед каждым тестом

Как же очищать данные?

- В базовый класс для интеграционных тестов помещаем сценарий удаления.
- Отключаем все внешние ключи, очищаем каждую таблицу и восстанавливаем внешние ключи.
- 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 3 сценария

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

	как в production	SQLite in-memory	Заглушка вместо базы данных
+	<ul style="list-style-type: none"> · все как в production · полная поддержка SQL 	<ul style="list-style-type: none"> · быстрый запуск · актуальная схема · стартует пустой 	<ul style="list-style-type: none"> · полный контроль за доступом к данным · быстрый запуск
-	<ul style="list-style-type: none"> · нужны уникальные экземпляры db на тест · медленно создается/очищается 	<ul style="list-style-type: none"> · частичная поддержка SQL · все как в production · нюансы относительно production 	<ul style="list-style-type: none"> · не все тестируется · много кода
когда нужно	используется SQL	используется только LINQ	тестируется сложная бизнеслогика

SQLite – быстрее и лимитированный

- Схема базы данных всегда актуальна.
- База данных пуста, что является хорошей отправной точкой для теста.
- Параллельное выполнение тестов работает, потому что каждая база данных хранится локально в каждом тесте.
- Ваши тесты будут успешно выполняться в любом pipeline'e без каких-либо дополнительных настроек.
- 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();

    //Rest of test is left out
}
```

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.");
}
```


Убеждаемся, что тест как в production

Рассмотрим проблему с 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;
    // Should call 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();
    //VERIFY
    context.ChangeTracker.Clear();
    context.Books.Include(b => b.Reviews).Single(b => b.BookId = setupBooks.Last().BookId)
        .Reviews.Count.ShouldEqual(3);
}
```

Лучшие данные для тестирования

Сериализуем конкретные данные из существующей базы данных и сохраняем их в виде файла JSON

Seed from Production

Функция "Seed from Production" позволяет сделать snapshot существующей (производственной) базы в JSON, который можно использовать для воссоздания тех же данных в новой базе данных для тестирования приложения.

Создание базы в контейнере

- Помещаем базу данных в образ Docker
- Создаем новый экземпляр контейнера из этого образа для каждого интеграционного теста

Правила запуска в Docker

- Каждый тест запускаем в отдельном контейнере.
- Пакетный запуск тестов.
- Останавливаем и удаляем использованные контейнеры.

Testcontainers-dotnet

- Легкие, временные экземпляры баз данных в Docker-контейнере.
- 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

Минусы:

- множество нюансов уже реализованных в готовых инструментах

Плюсы:

- 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());

connection.ConnectionString = connection
    .ConnectionString.Replace(FakePort, freePort);

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. Создайте новую
 - Исключения – возможная потеря данных
- ✓ Изоляция тестов
 - У каждого разработчика должна быть своя база
 - Очищаем данные перед каждым тестом
- ✓ Библиотеки
 - Respawn
 - EfCore.TestSupport
 - Testcontainers-dotnet

Следует ли тестировать чтения?

Тестируем только самые сложные или важные операции чтения.

Тестирование чтения

- Для чтения нет необходимости в модели предметной области.
- Если вы решите протестировать чтение, сделайте это с помощью интеграционных тестов на реальной базе данных.

Следует ли вам тестировать репозитории?

- Репозитории обеспечивают полезную абстракцию поверх базы данных.
- Должны ли вы тестировать репозитории независимо от других интеграционных тестов?

Тестирование репозитория

- Высокие затраты на поддержку.
- Нет преимуществ перед обычными интеграционными тестами.
- Лучший способ действий при тестировании репозитория - извлечь небольшую сложность, которой он обладает, в автономный алгоритм и протестировать исключительно этот алгоритм.

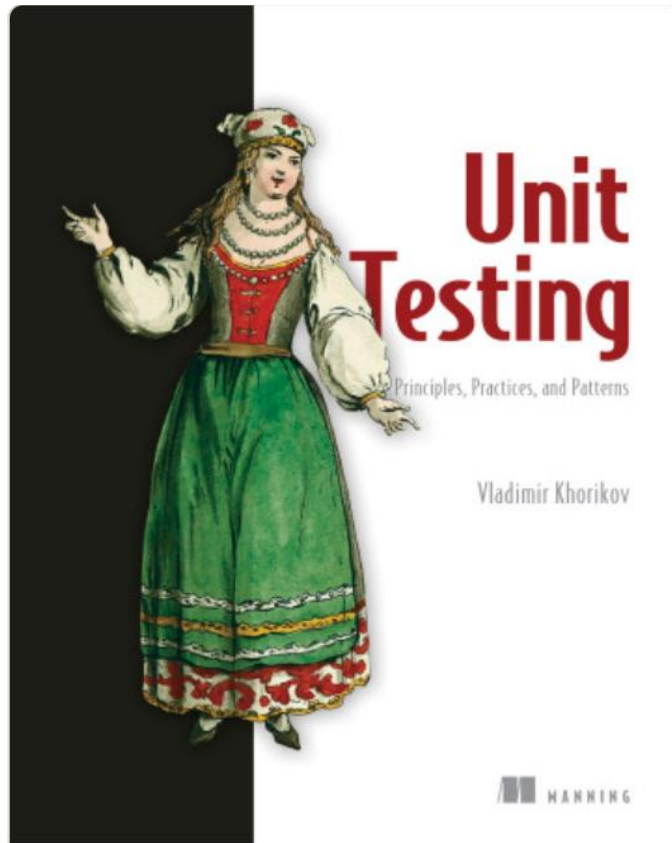
Что мы уже узнали?

- ✓ 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. Создайте новую
 - Исключения – возможная потеря данных
- ✓ Изоляция тестов
 - У каждого разработчика должна быть своя база
 - Очищаем данные перед каждым тестом
- ✓ Библиотеки
 - Respawn
 - EfCore.TestSupport
 - Testcontainers-dotnet
- ✓ Советы
 - Не тестируем чтения
 - Не пишем модульные тесты на репозитории

Ссылки

Unit Testing Principles, Practices, and Patterns Vladimir Khorikov

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



ССЫЛКИ

Respawn

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

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

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

ССЫЛКИ

EfCore.TestSupport

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

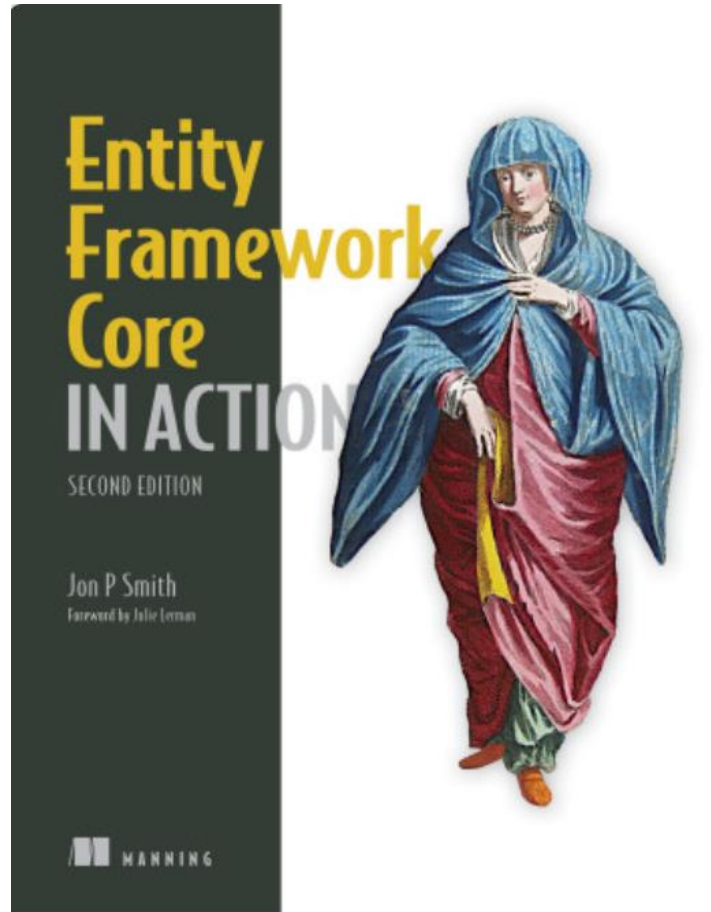
<https://www.thereformedprogrammer.net/getting-better-data-for-unit-testing-your-ef-core-applications/>

<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



Недавно вышла в переводе
<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