Тестируем код, взаимодействующий с базой данных

Гурий Самарин

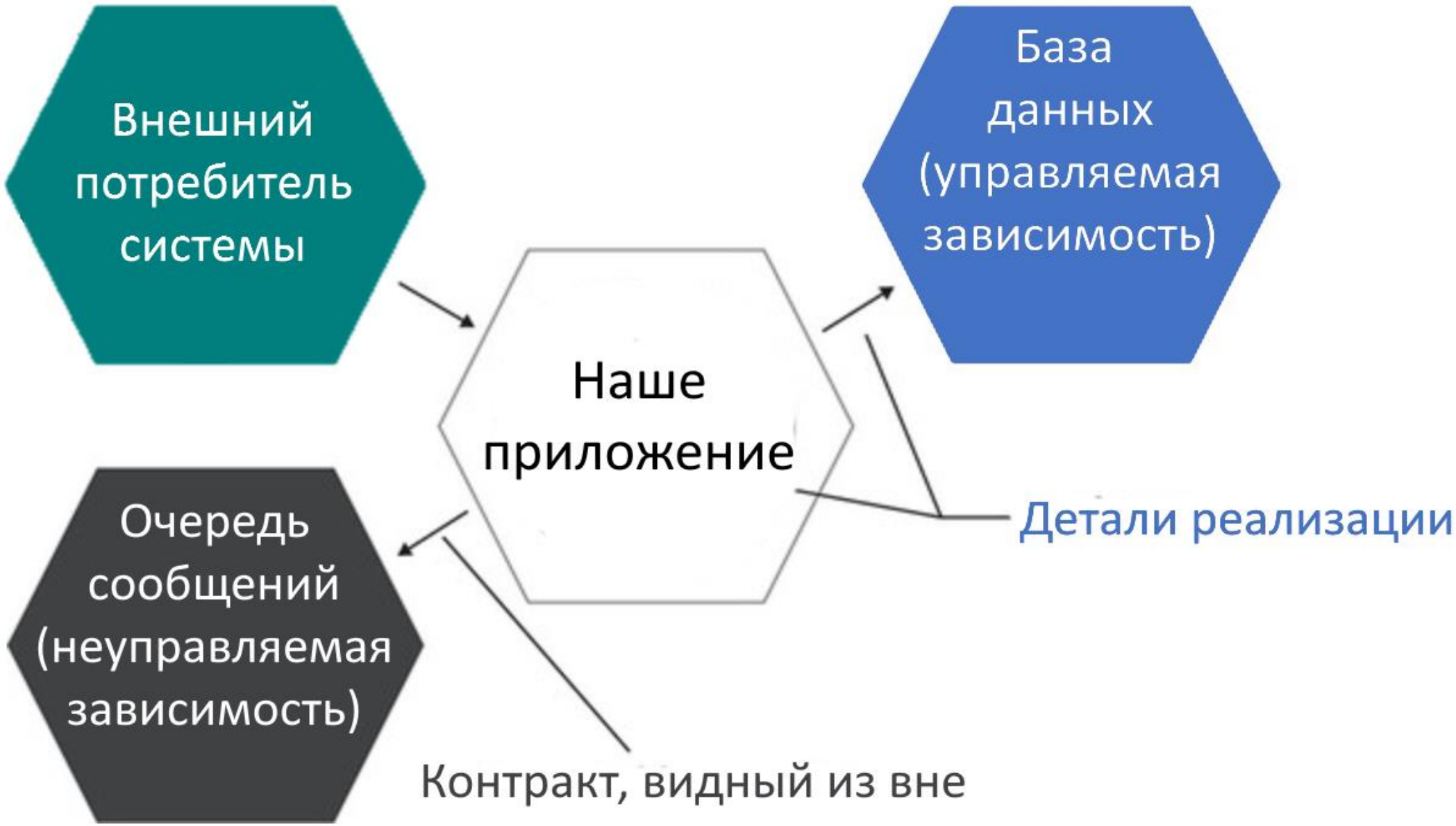
План доклада

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

Какие зависимости мы тестируем?

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

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


Внешний потребитель системы
База данных (управляемая зависимость)
Наше приложение
Очередь сообщений (неуправляемая зависимость)
Детали реализации
Контракт, видный из вне

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
  - Надо тестировать в продбазе (естественно на другом эземпляре)

Как эффективно тестировать связь с бд?

- Храним схему базы данных в системе управления версиями
- Используем отдельные экземпляры базы данных для каждого разработчика
- Применяем миграции для изменения схемы

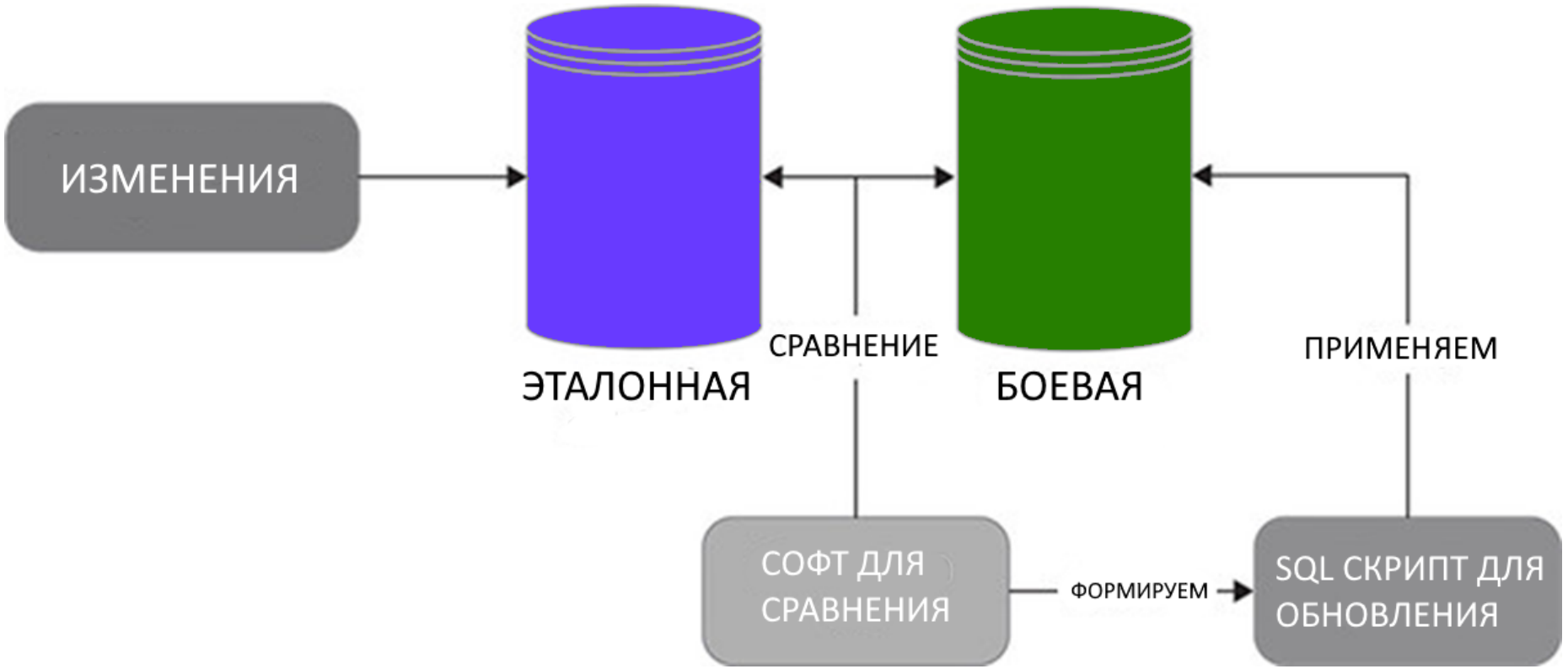
Схема базы в git’е

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

Выделенный экземпляр базы данных, который служит отправной точкой (моделью базы данных).

Наличие выделенного экземпляра в качестве эталонной базы данных



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** 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 $EF$;
```

Доставка 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

Подход, основанный на миграции, подчеркивает использование явных миграций, которые переводят базу данных из одной версии в другую.

Миграции vs состояния

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

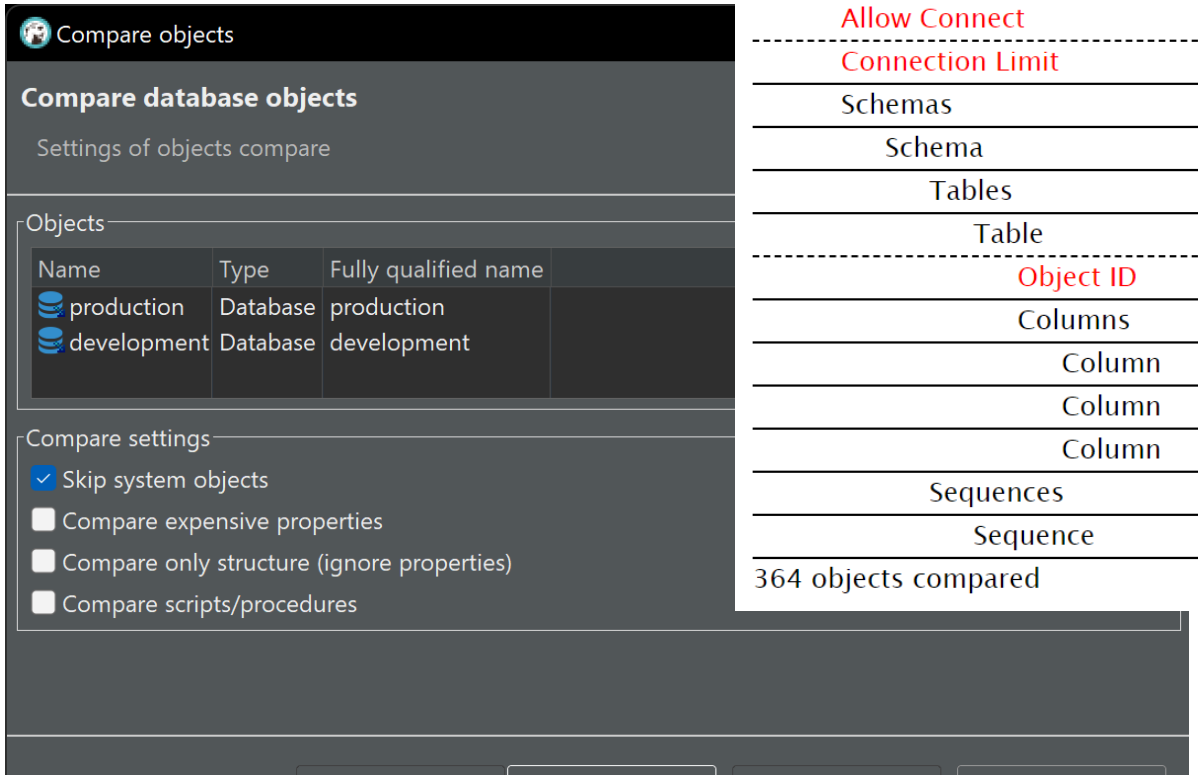
Mergeконфликты VS трансформации данных

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

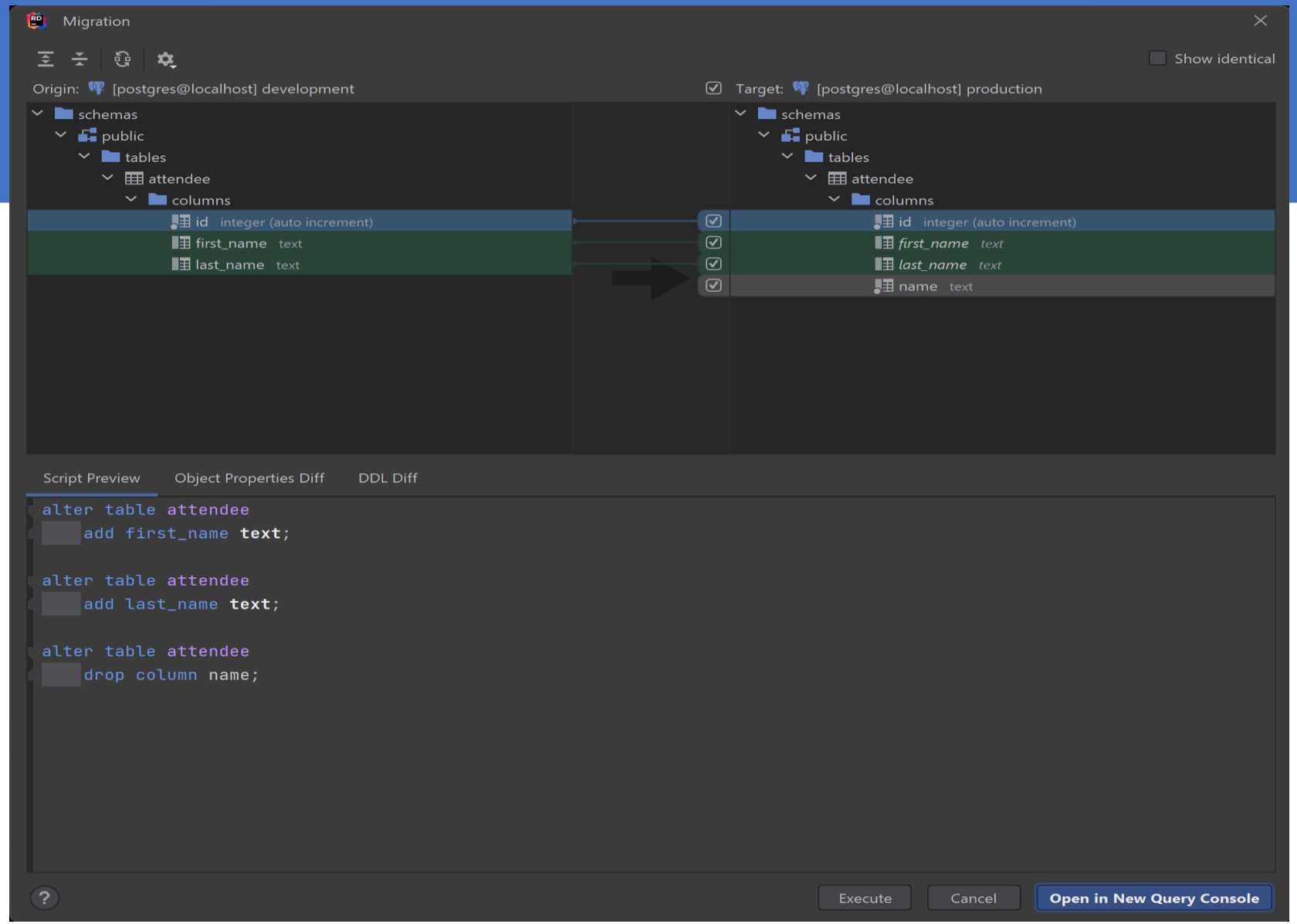
DBeaver



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

364 objects compared

DataGrip


Migration
Show identical
Origin: [postgres@localhost] development
Target: [postgres@localhost] production
schemas
public
tables
attendee
columns
id integer (auto increment)
first_name text
last_name text
name text
Script Preview
Object Properties Diff
DDL Diff
alter table attendee
add first_name text;
alter table attendee
add last_name text;
alter table attendee
drop column name;
Execute
Cancel
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;
```

Миграции прямо перед production

тестовые данные не так уж важны, и 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>
```

Итог

- Используйте миграции
- Не изменяйте миграции. Создайте новую
- Исключения – возможная потеря данных

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
  - Надо тестировать такой же базе, как в проде (естественно на другом эземпляре)
- 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.

- Обратно несовместимые изменения могут блокировать работу других разработчиков.

Управляем состоянием базы в тестах

- изолированные тестовые инстансы
- предсказуемое наполнение

Параллельное vs последовательное

Параллельное выполнение интеграционных тестов требует значительных усилий.

Большинство фреймворков модульного тестирования позволяют определять отдельные тестовые коллекции и выборочно отключать в них распараллеливание.

Коллекции в xUnit

```
[CollectionDefinition(nameof(NotThreadSafeResourceCollect
ion), DisableParallelization = true)]
public class NotThreadSafeResourceCollection { }

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

Управляем жизненным циклом данных

- Выполняйте интеграционные тесты последовательно.
- Удалите оставшиеся данные между тестовыми запусками.
- Приведите базу к начальному состоянию в самих тестах.

Очистка между тестовыми запусками

Четыре варианта очистки оставшихся данных между тестовыми запусками:

- Восстановление резервной копии базы данных перед каждым тестированием
- Очистка данных в конце теста

Очистка между тестовыми запусками

Оборачиваем каждый тест в транзакцию и делаем Rollback

Особенности реализации

- 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; } =
TransactionScopeAsyncFlowOption.Enabled;
        public IsolationLevel IsolationLevel { get; set; } =
IsolationLevel.Unspecified;
        public TransactionScopeOption ScopeOption { get; set; } =
TransactionScopeOption.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);
        }
}
```

Проблемы

- Несколько транзакций - подход не сработает

- Если простого отката недостаточно - нужна простая очистка базы данных перед каждым тестом

Очистка между тестовыми запусками

Очистка данных в начале теста — это лучший вариант.

Очистка данных

Удаление данных должно выполняться в определенном порядке, чтобы соответствовать ограничениям внешнего ключа базы данных.

Рекоммендация

Введите базовый класс для всех интеграционных тестов и поместите туда сценарий удаления. С таким базовым классом скрипт будет запускаться автоматически в начале каждого теста.

Следует ли тестировать чтения?

Тестируем только самые сложные или важные операции чтения.

Тестирование чтения

Для чтения также нет необходимости в модели предметной области.

Поскольку в чтениях практически нет уровней абстракции (модель предметной области является одним из таких уровней), модульные тесты там бесполезны.

Если вы решите протестировать свои чтения, сделайте это с помощью интеграционных тестов в 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 техническое обслуживание.
- Тесты репозитория не имеют 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.

Сброс базы данных перед каждым тестом

- восстановление из заведомо “хорошей” резервной копии
- отключение всех внешних ключей, очистка каждой таблицы и восстановление внешних ключей
- найти “правильный” порядок удаления данных на основе взаимосвязей и удалить данные из каждой таблицы по порядку

Восстановление из резервной копии

- если база данных меняется не часто
- полезно только в случае использования производственной базы данных в качестве тестовой базы данных

Отключение внешних ключей

- медленно – 3 команды на таблицу
- можно создать только пустую базу

“Правильный” порядок удаления данных

- наиболее эффективное решение
- наиболее сложное в реализации

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
  - Надо тестировать такой же базе, как в проде (естественно на другом эземпляре)
- 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
  - Не тестируем чтения
  - Не пишем модульные тесты на репозитории

Respawn by J. Bogard

- построение ориентированного графа по внешним ключам
- обход ориентированного графа - порядок, в котором мы удаляем таблицы
- в случае цикла отключаем ограничения только в нем и удаляем

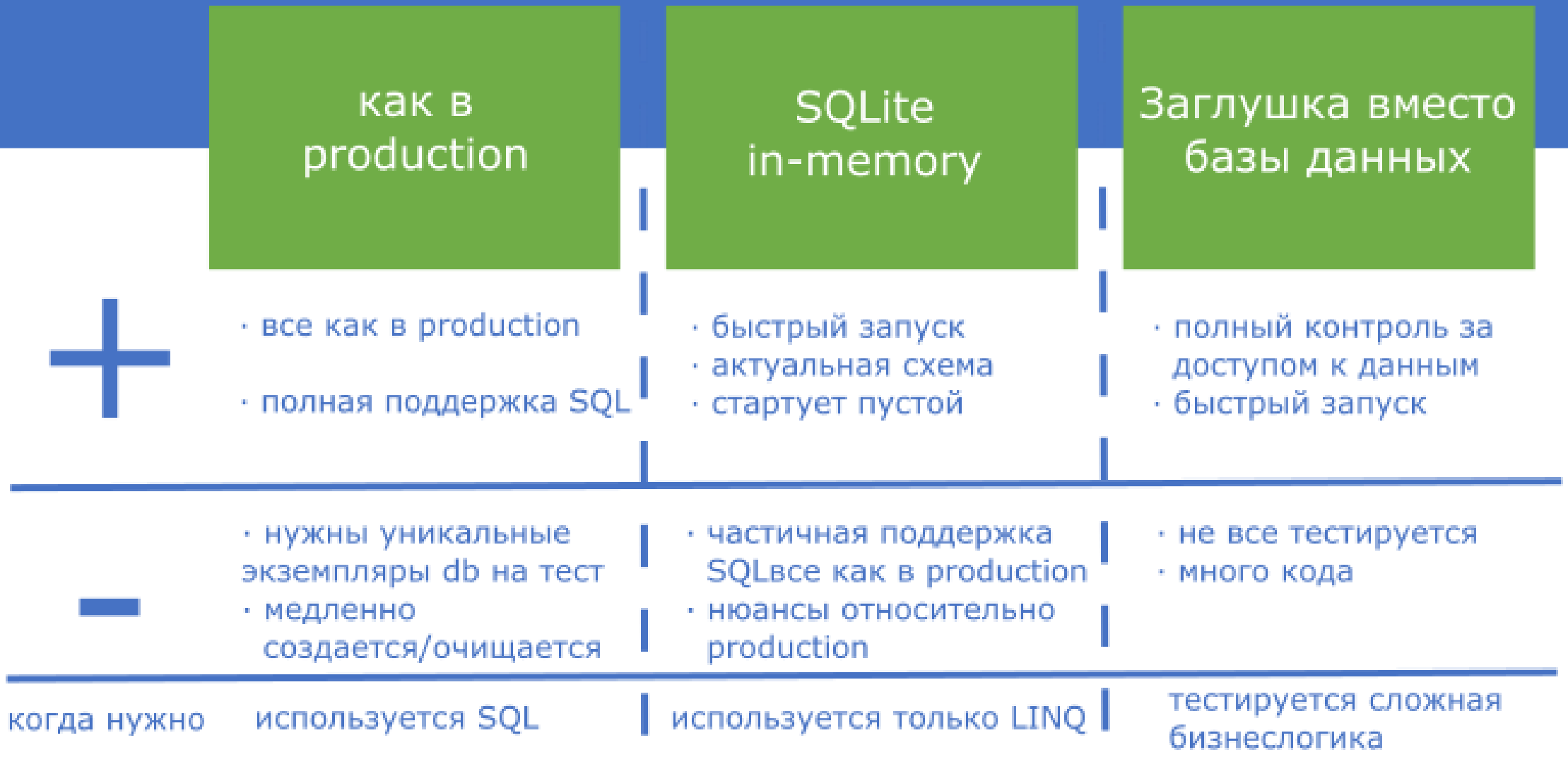
Пример использования

```
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

сравним 3 сценария

- SQLite в памяти.
- мок репозитория.
- та же база данных, что в production.


как в production
SQLite in-memory
Заглушка вместо базы данных
+
· все как в production
· полная поддержка SQL
· быстрый запуск
· актуальная схема
· стартует пустой
· полный контроль за доступом к данным
· быстрый запуск
-
· нужны уникальные экземпляры db на тест
· медленно создается/очищается
· частичная поддержка SQLвсе как в production
· нюансы относительно production
· не все тестируется
· много кода
когда нужно
используется SQL
используется только LINQ
тестируется сложная бизнеслогика

SQLite – быстрейший 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’е без каких-либо дополнительных настроек.

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();

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

Как 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();

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

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()
{    //ARRANGE
    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 вам записать ”моментальный снимок" существующей (производственной) базы данных в файл JSON, который вы можете использовать для воссоздания тех же данных в новой базе данных для тестирования вашего приложения.

Контейнеры

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

Docker

- Каждый тест запускаем в отдельном контейнере.
- Запускаем тесты пачками.
- Останавливаем и удаляем использованные контейнеры.

Testcontainers-dotnet

Testcontainers - это библиотека для поддержки тестов, позволяющая создавать одноразовые экземплярами контейнеров Docker.

Docker.DotNet: тестируем в докере

Минусы:

- множество нюансов уже реализованных в готовых инструментах

Плюсы:

- автоматизация
- полное управление

Получаем список контейнеров

```
// create container, if one doesn't already exist
var contList = await dockerClient
                    .Containers.ListContainersAsync(new
                         ContainersListParameters { All = true });
```

Создаем контейнер

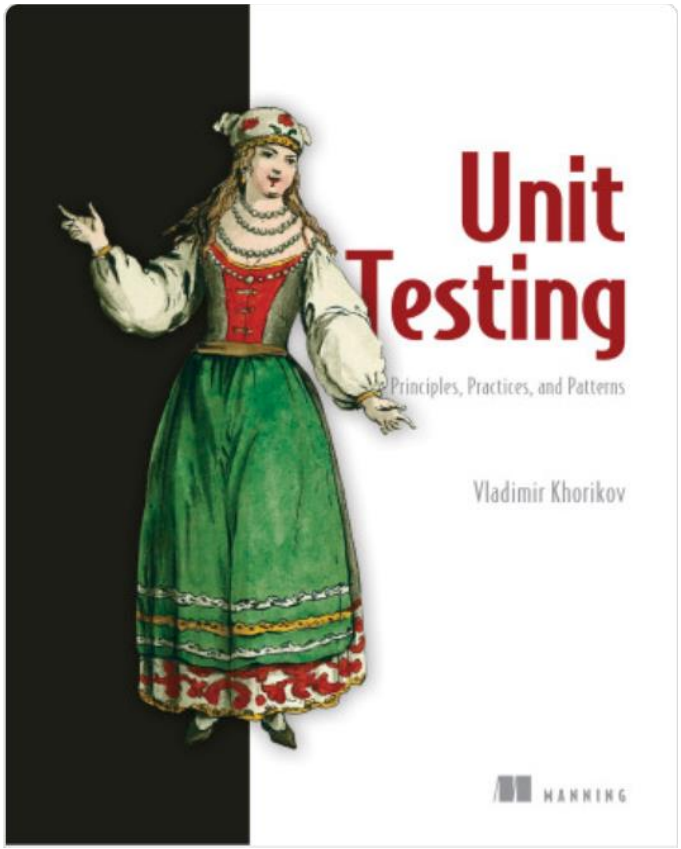
```
var sqlContainer = await dockerClient.Containers
                .CreateContainerAsync(new CreateContainerParameters
                {
                    Name = _dbContainerName,
                    Image = DbImage,
                    Env = Env,
                    HostConfig = new HostConfig
                    {
                        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);

return (sqlContainer.ID, freePort);
```

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/

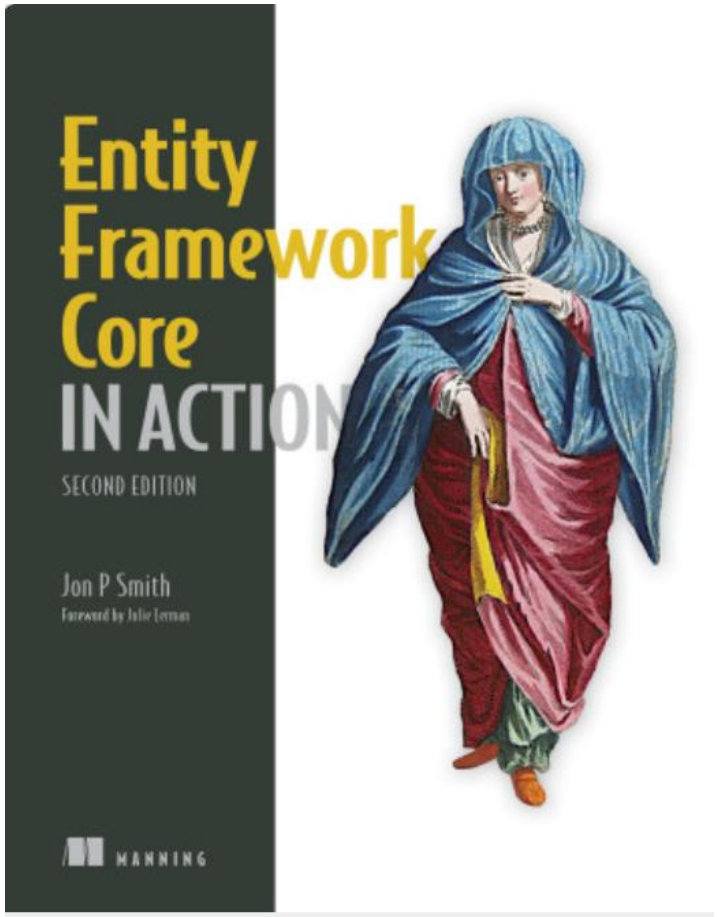
https://www.thereformedprogrammer.net/getting-better-data-for-unit-testing-your-ef-core-applications/

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