**Интеграционное тестирование. Написание тестовых сценариев.**

Как было сказано в модуле “Интеграционное тестирование”, целью данного тестирования является проверка корректности работы методов, которые обращаются непосредственно к базе данных.

Прежде чем начать писать тестовые сценарии, мы должны определиться с тем, как мы будем обращаться к базе данных. У нас есть несколько вариантов развития событий, некоторые из них: использование локальной базы данных, подмена базы данных на репозиторий. Так как база данных, которую мы сейчас используем, небольшая и содержит малое количество таблиц и записей, мы будем обращаться непосредственно к ней. **НО** перед написанием каждого тестового сценария, мы будем наполнять БД тестовыми данными. Так как эти данные будут находится в БД, нам нужно будет их чистить, чтобы избежать пересечения с другими тестовыми сценариями. Чтобы избежать громоздкости кода, мы воспользуемся классом *TransactionScope* из пространства имён *System.Transactions.* Данный класс позволяет нам создать область транзакции, в которой мы будем создавать тестовые данные и обращаться к базе данных. Когда тестовый сценарий будет выполнен, мы будем вызывать метод *Dispose()*, который завершает область транзакции и указывает, что все изменения в БД будут отменены, то есть произойдёт откат.

Для начала создадим новый класс тестов в проекте *TestDataBase,* я назвал его *IntegrationTests*. Так как мы будем обращаться непосредственно к базе данных, для того чтобы наполнить её тестовыми данными, помимо пространства имён NUnit.Framework, нужно подключить System.Data.SqlClient., и не забудем про System.Transactions.

using NUnit.Framework;

using System.Data.SqlClient;

using System.Transactions;

Для начала определим класс тестов и объявим несколько переменных:

[TestFixture]

class IntegrationTestDataBase

{

    DataBase db;

    TransactionScope scope;

}

Как и в случае с unit-тестами мы определим метод с атрибутом [SetUp], однако вдобавок нам необходимо создать метод с атрибутом [TearDown]. Данный атрибут говорит о том, что метод с этим атрибутом будет выполнятся после каждого теста. Именно в этом методе мы будем вызывать функцию Dispose(). Эти методы будут иметь следующий вид:

[SetUp]

public void SetUp()

{

    scope = new TransactionScope();

    db = new DataBase();

    db.OpenConnection();

}

[TearDown]

public void TearDown()

{

    db.CloseConnection();

    scope.Dispose();

}

Перейдём к написанию первого интеграционного теста.

Основными методами, которые необходимо протестировать, являются методы CRUD: Create, Read, Update, Delete. Первым делом протестируем создание записи. Создадим метод с атрибутом [Test], и выполним тот метод, который хотим протестировать, а именно метод **создания** новой записи. В моём случае это метод CreateNewUser. Приведу код теста и метода:

[TestCase("vlad", "vlad")]

public void TestCreateNewUser(string login, string password)

{

    db.CreateNewUser(login, password);

}

public void CreateNewUser(string login, string password)

{

    OpenConnection();

    string Query = QueryNewUser(login, password);

    ExecuteSqlCommandCreate(Query);

    CloseConnection();

}

Выполнив этот метод, мы ожидаем, что в таблице появится новая запись. Чтобы это проверить мы вновь обратимся к базе данных, но уже в рамках теста, и считаем запись с следующими данными: login = vlad; password = vlad. Таким образом, весь тест у нас будет выглядеть так:

[TestCase("vlad", "vlad")]

public void TestCreateNewUser(string login, string password)

{

    db.CreateNewUser(login, password);

    var table = new DataTable();

    var command = new SqlCommand($"select login\_user, password\_user from Register where login\_user = '{login}' and password\_user = '{password}'", db.GetConnection());

    var adapter = new SqlDataAdapter();

    adapter.SelectCommand = command;

    adapter.Fill(table);

    Assert.AreEqual(1, table.Rows.Count);

    Assert.AreEqual(login, table.Rows[0].ItemArray[0]);

    Assert.AreEqual(password, table.Rows[0].ItemArray[1]);

}

Создадим тестовый сценарий, в котором мы проверим **чтение** записи. Действуем по той же самой формуле: . Для проверки мы создаём таблицу из данных, которые мы считываем. Код теста:

[TestCase("10", "log", "pass", false)]

public void TestGetUsers(int id, string login, string password, bool isAdmin)

{

    DataTable expected = new DataTable();

    for (int i = 0; i < 4; i++) expected.Columns.Add();

    DataRow r = expected.NewRow();

    r[0] = id; r[1] = login; r[2] = password; r[3] = isAdmin;

    expected.Rows.Add(r);

    var command = new SqlCommand($"insert into Register (login\_user, password\_user, is\_admin) values ('{login}','{password}',0)",db.GetConnection());

    command.ExecuteNonQuery();

    DataTable actual = db.GetUsers(login, password);

    Assert.AreEqual(expected.Rows[0].ItemArray[1], actual.Rows[0].ItemArray[1].ToString().Trim());

    Assert.AreEqual(expected.Rows[0].ItemArray[2], actual.Rows[0].ItemArray[2].ToString().Trim());

    Assert.AreEqual(expected.Rows[0].ItemArray[3], actual.Rows[0].ItemArray[3].ToString().Trim());

}

Для **модификации** записи, порядок действий не меняется, мы так же вставляем новые данные, так как у ID\_psg (первичный ключ в таблице Passenger) заполняется в таблице автоматически, мне необходимо его каждый раз считывать, выполняем тестируемый метод, и проверяем результат. Код:

[TestCase("Kidow J.L.", "Kidow S.A"]

public void TestUpdatePassenger(string name, string newName)

{

    var command = new SqlCommand($"insert into Passenger(name) values ('{name}')", db.GetConnection());

    command.ExecuteNonQuery();

    var table = new DataTable();

    var adapter = new SqlDataAdapter($"select ID\_psg from Passenger where name = '{name}'", db.GetConnection());

    adapter.Fill(table);

    var ID\_psg = table.Rows[0].ItemArray[0].ToString().Trim();

    db.UpdatePass(ID\_psg, newName);

    adapter.SelectCommand = new SqlCommand($"select name from Passenger where ID\_psg = '{ID\_psg}'", db.GetConnection());

    table = new DataTable();

    adapter.Fill(table);

    Assert.AreEqual(newName, table.Rows[0].ItemArray[0].ToString().Trim());

}

Для **удаления** код отличается немногим, в Assert.AreEqual(…) мы будем сравнивать количество строк, которые возвратятся при запросе, в котором мы попытаемся получить удалённые тестовые данные из таблицы. Код:

[TestCase("Kidow J.L.", RowState.Deleted)]

public void TestDeletePassenger(string name, RowState \_rw)

{

    var command = new SqlCommand($"insert into Passenger(name) values ('{name}')", db.GetConnection());

    command.ExecuteNonQuery();

    var table = new DataTable();

    var adapter = new SqlDataAdapter($"select ID\_psg from Passenger where name = '{name}'", db.GetConnection());

    adapter.Fill(table);

    var ID\_psg = table.Rows[0].ItemArray[0].ToString().Trim();

    db.UpdatePass(ID\_psg, name, \_rw);

    adapter.SelectCommand = new SqlCommand($"select \* from Passenger where ID\_psg = '{ID\_psg}'", db.GetConnection());

    table = new DataTable();

    adapter.Fill(table);

    Assert.AreEqual(0, table.Rows.Count);

}