Добавление первого теста в проект. Тестирование во время непрерывной интеграции.

Overview

Разберем одну из важнейших практик для любого разработчика при написании кода, благодаря чему качество программного продукта может значительно улучшиться - а именно написание юнит тестов

Юнит тесты - это автоматические тесты, которые проверяют работу отдельных частей программного кода, называемых юнитами. Целью написания юнит тестов является обеспечение корректной работы программы, а также упрощение и ускорение процесса разработки.

Когда вы пишете программу, вы можете сделать ошибку в коде, которую не заметите до того, как запустите приложение. Юнит тесты помогают обнаружить такие ошибки на ранних стадиях разработки и исправить их, что позволяет сэкономить время и снизить риск возникновения проблем в будущем.

Типы тестирования

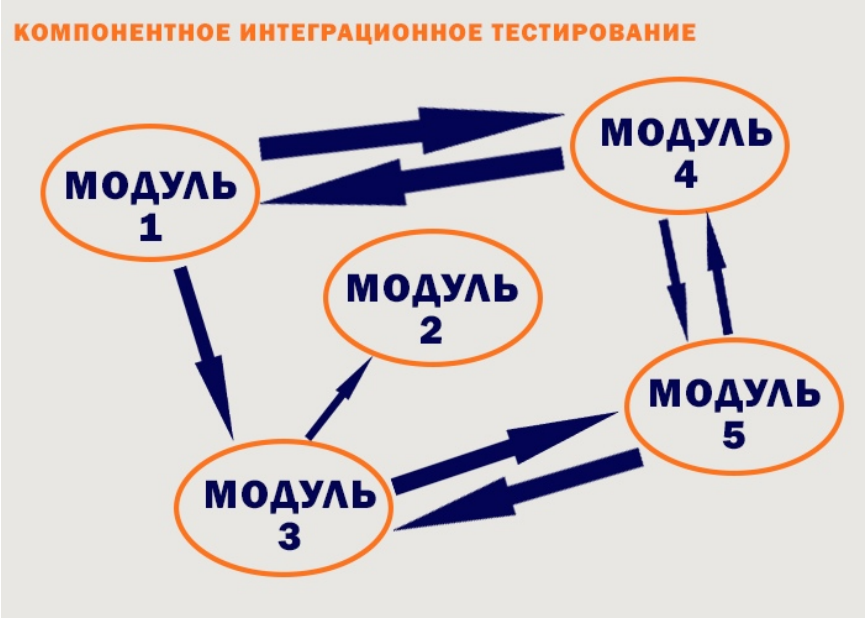
Существует множество уровней тестирования. И в них иногда очень легко заблудиться, поэтому разберем их:

1. Компонентное/модульное тестирование (Component/Unit Testing)

   Этот вид тестирования выполняется на самой ранней стадии разработки программы — во время написания кода. Обычно его выполняет сам программист, который пишет код. Следовательно, ошибки, в большинстве случаев, исправляются сразу же и не попадают к специалистам по тестированию.

2. Интеграционное тестирование (Integration Testing)

   Интеграционное тестирование необходимо для того ,чтобы тестировать корректную работу модуля, связанный с другими модулями.



3. Системное тестирование (System Testing)
   Системное тестирование — это тестирование еще более высокого уровня. При системном тестировании наша задача уже состоит в том, чтобы убедиться в корректности работы в целом всей системы. Программа в этом случае должна быть максимально приближена к конечному результату. А наше внимание должно быть сосредоточено на общем поведении системы с точки зрения конечных пользователей.

4. Приемочное тестирование (e2e)

Приемочное тестирование — наиболее высокий уровень тестирования. Оно, также как и системное тестирование, необходимо для проверки работы программы в целом.

Этот уровень тестирования используется для подтверждения готовности продукта и проводится преимущественно в самом конце цикла разработки программы.

Таким образом, процесс тестрование выстраивается следующим образом

В связи с тем, что именно мы пишем код, то наиболее востребованным для нас, как разработчиков, является именно модульное тестирование (unit-testing). Поэтому наибольшее внимание уделим именно им.

Фрейморки тестирования

Для написания юнит-тестов мы можем сами создавать весь необходимый функционал, использовать какие-то свои способы тестирования, однако, как правило, для этого применяются специальные фреймворки. Некоторые из них:

- xUnit.net: фреймворк тестирования для платформы .NET. Наиболее популярный фреймворк для работы именно с .NET Core и ASP.NET Core
- MS Test: фреймворк юнит-тестирования от компании Microsoft, который по умолчанию включен в Visual Studio и который также можно использовать с .NET Core
- NUnit: портированный фреймворк с JUnit для платформы .NET

В связи с тем, что наиболее популярным является именно xUnit для тестирования на платформе .NET 5 и выше, разбирать процесс тестирования будем именно на нём.

Разработка через тестирование (Test-Driven-Development)

Отдельно стоит сказать о концепции TDD или разработка через тестирование. TDD представляет процесс применения юнит-тестов, при котором сначала пишутся тесты, а потом уже программный код, достаточный для выполнения этих тестов.

Использование TDD позволяет снизить количество потенциальных багов в приложении. Создавая тесты перед написанием кода, мы тем самым описываем способ поведения будущих компонентов, не связывая себя при этом с конкретной реализацией этих тестируемых компонентов (тем более что реализация на момент создания теста еще не существует). Таким образом, тесты помогают оформить и описать API будущих компонентов.

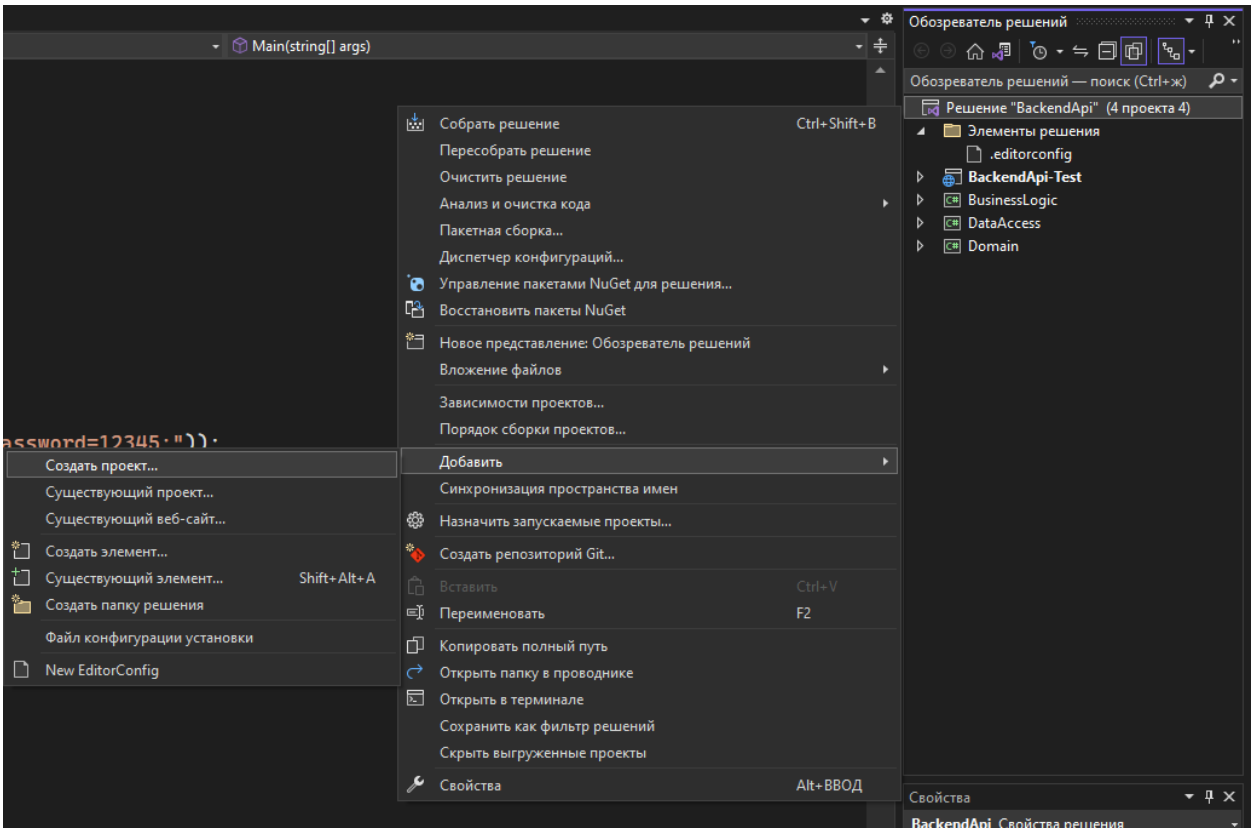
Порядок написания кода при TDD довольно прост:

1. Пишем юнит-тест
2. Запускаем его и видим, что он завершился неудачей (программный код ведь еще не написан)
3. Пишем некоторое количество кода, достаточное для запуска теста
4. Снова запускаем тест и видим его результаты

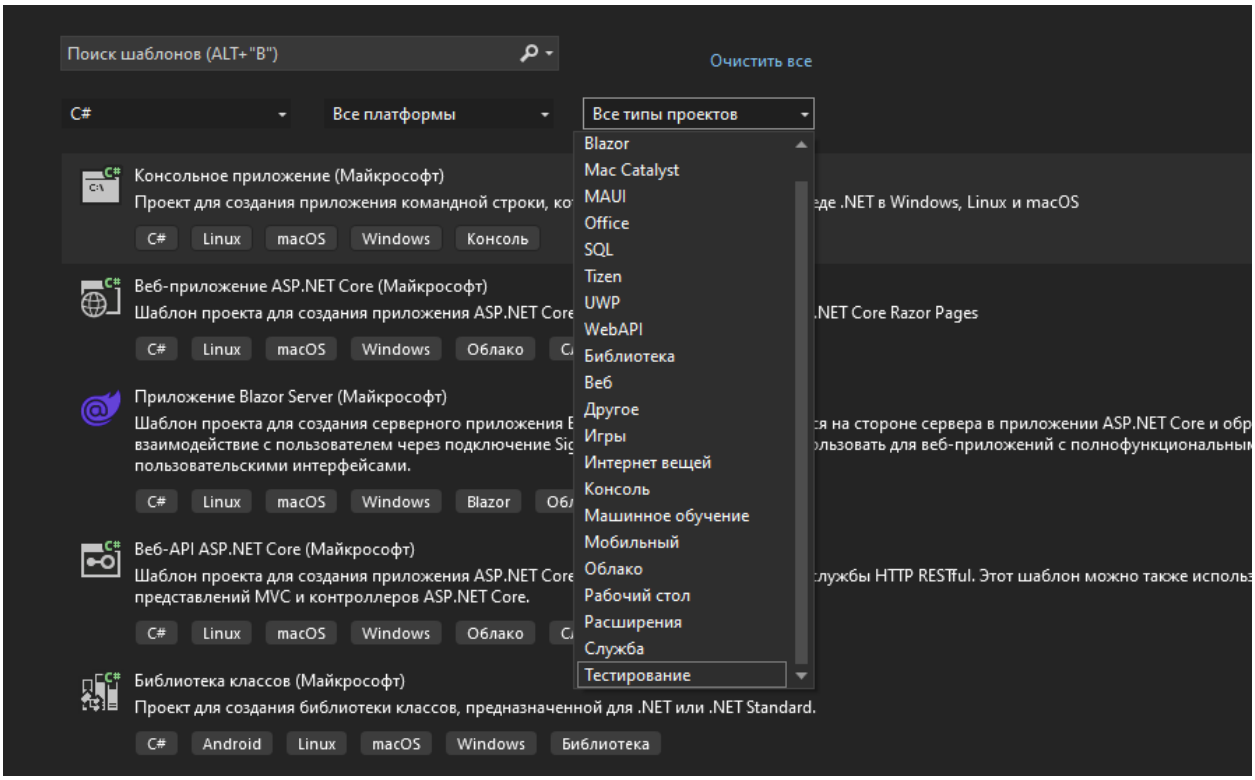
Этот цикл повторяется снова и снова, пока не будет закончена работа над программным кодом. Так как большинство фреймворков юнит-тестирования помечают неудавшиеся тесты с красного цвета (например, выводится текст красного цвета), а удачный тест отмечается зеленым цветом (опять же выводится текст зеленого цвета), то данный цикл часто называют красным/зеленым циклом.

Добавление первого теста. Применение подхода TDD.

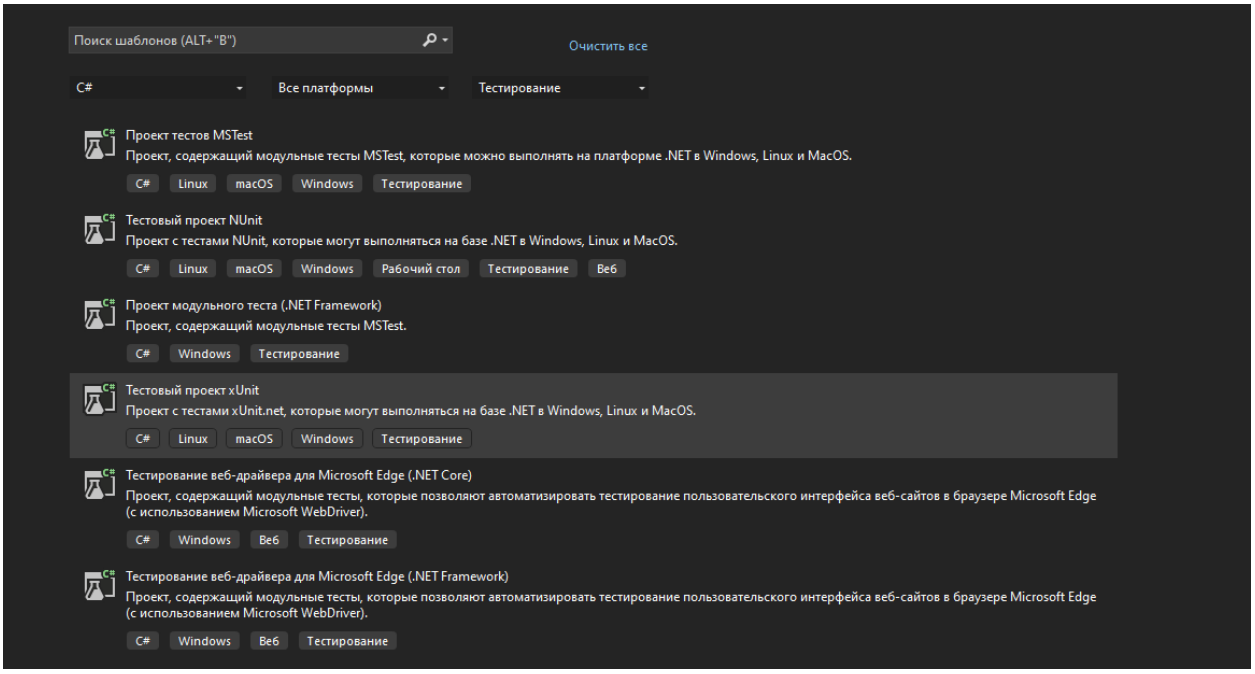
Откройте проект с вашим API. Выберите пункт “Создать проект…”



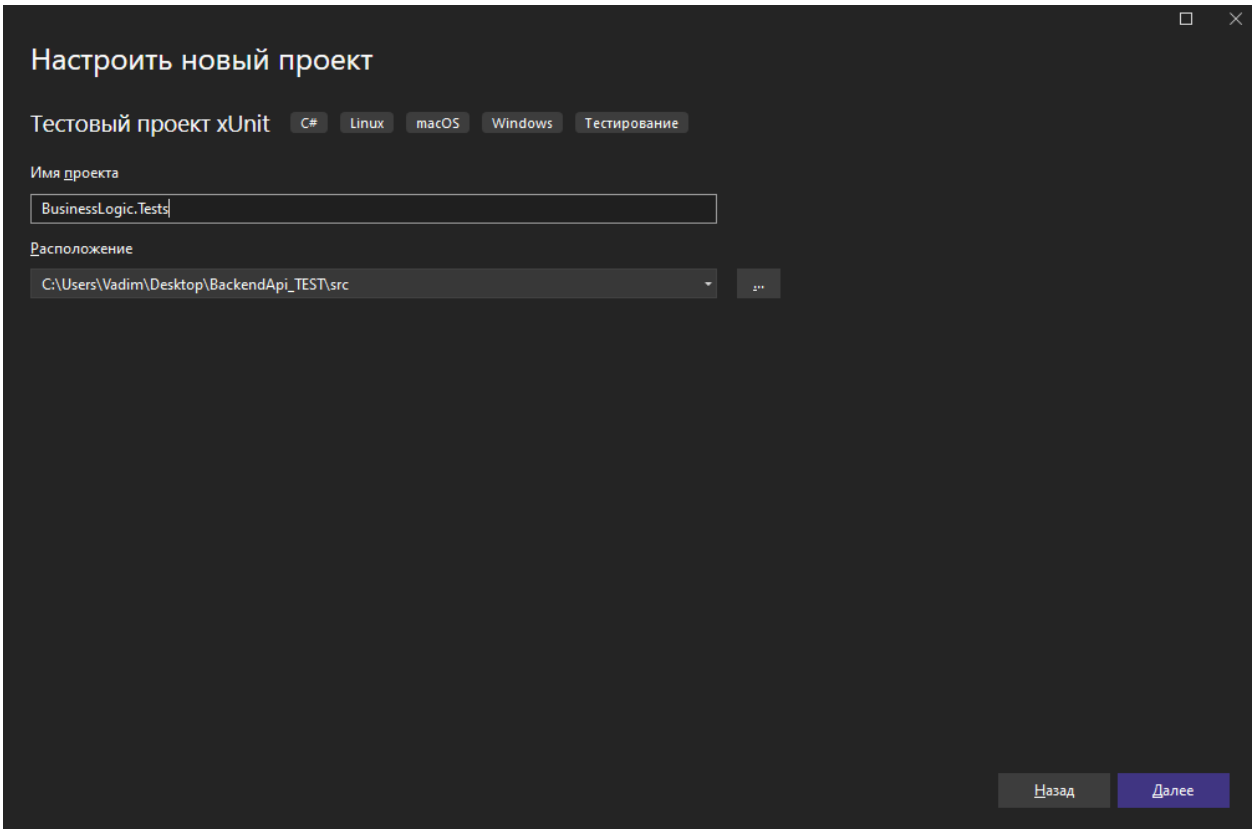
В окне создания проекта выберите тип проекта “Тестирование”



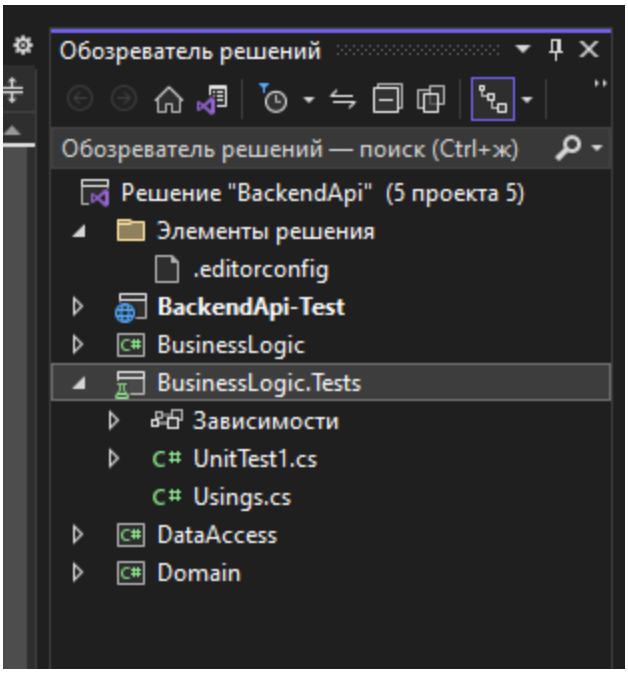
Выберите шаблон проекта “Тестовый проект xUnit”



Проверять мы будем бизнес-логику нашего проекта (Слой BusinessLogic). Поэтому назовем тестовый проект как “BusinessLogic.Tests”.

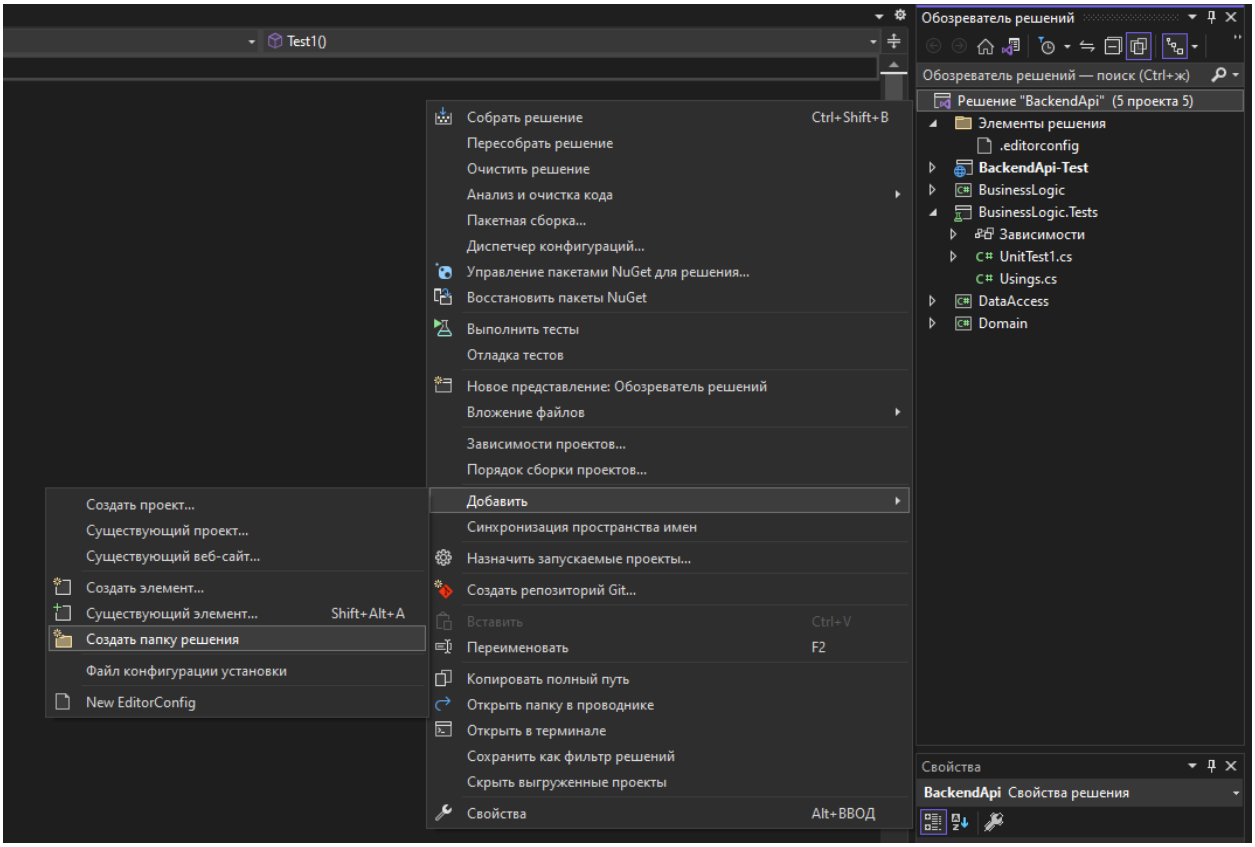


На данный момент в решении не так много проектов, однако позднее их будет немного больше.

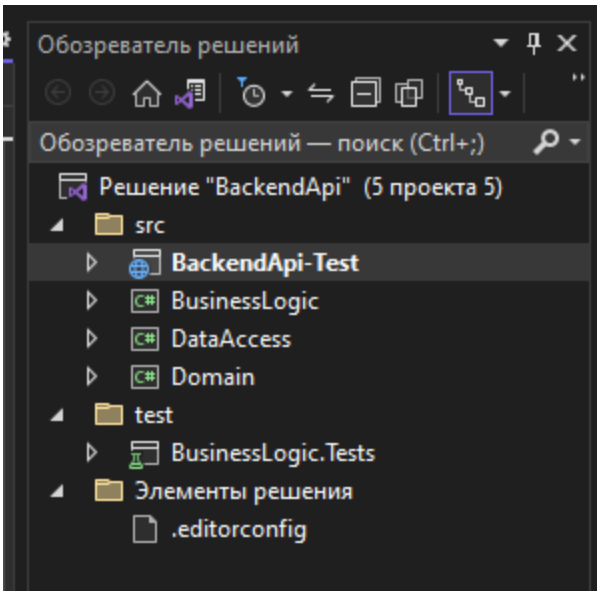


Для того, чтобы было удобно ориентироваться в решении проекта и не было путаницы между проектами - распределим проекты по папкам.

Нажмите правой кнопкой мыши по решению и выберите пункт “Создать папку решения”

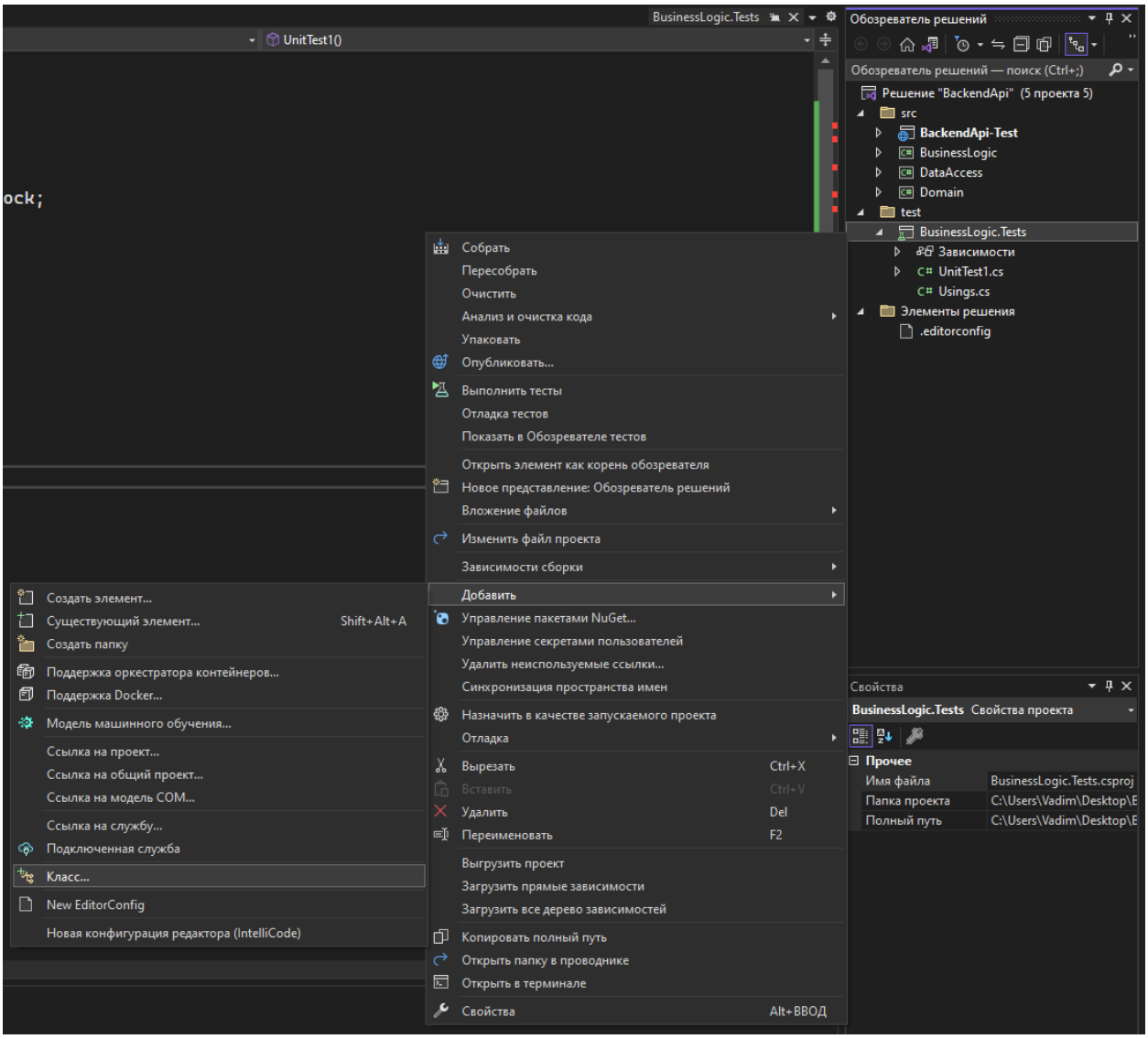


И приведите решение к следующему виду

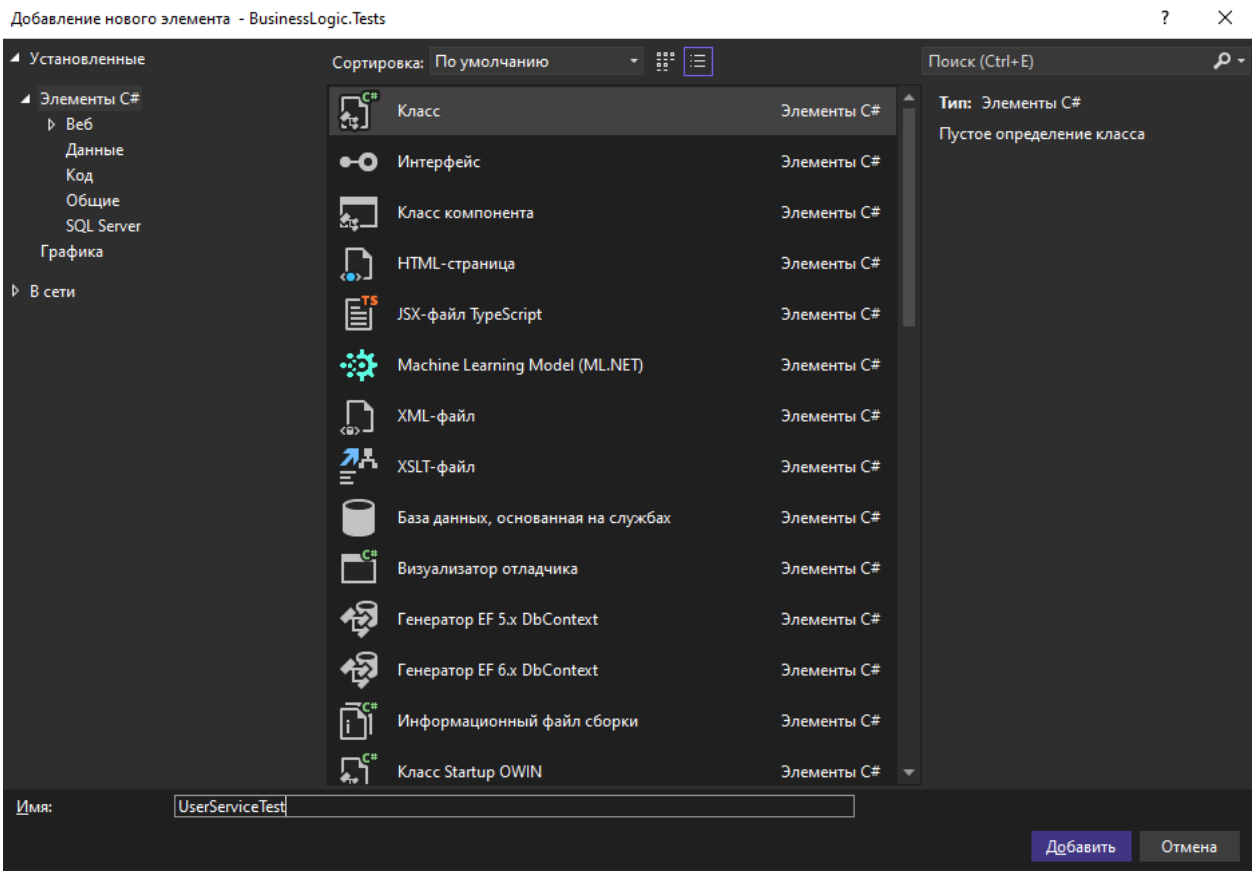


Написание тестов для UserService

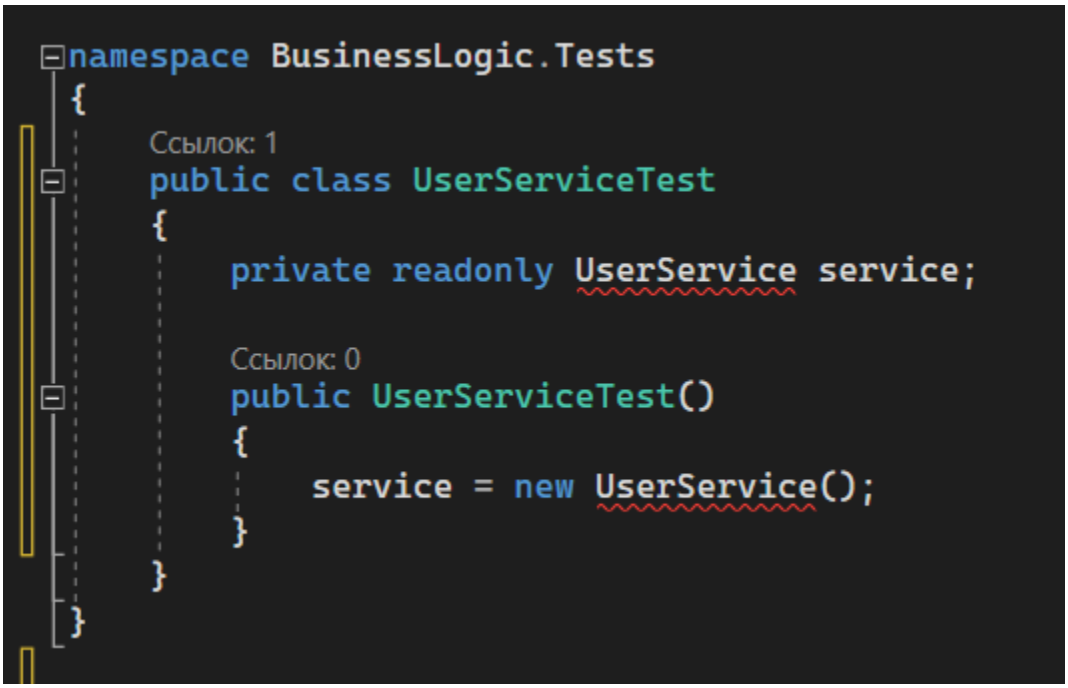
Создадим класс для тестирования работы класса UserService. Создайте новый класс.



Так как будем тестировать работу UserService, то назовем его как UserServiceTest

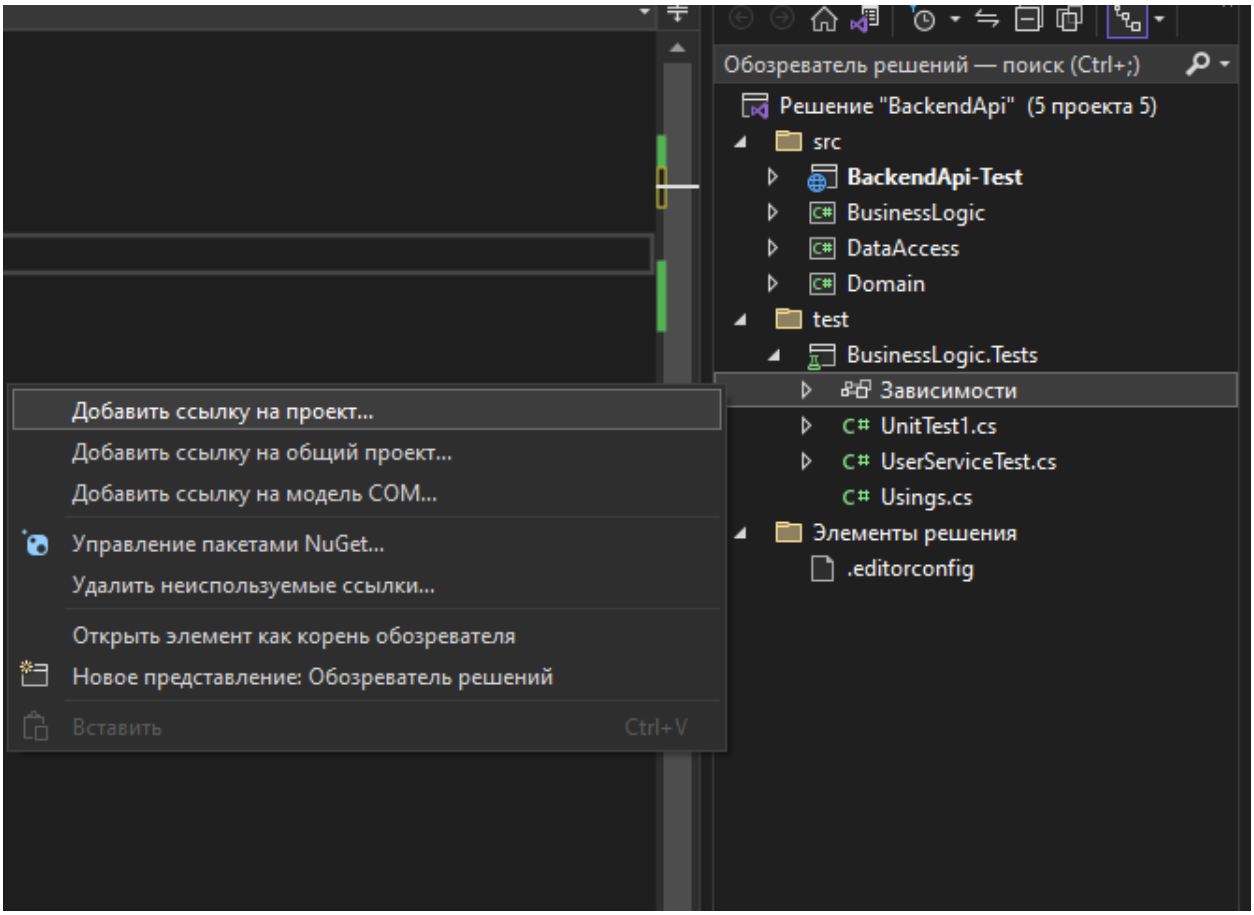


Теперь создайте конструктор класса UserServiceTest, для того, чтобы мы могли инициализировать класс UserService.

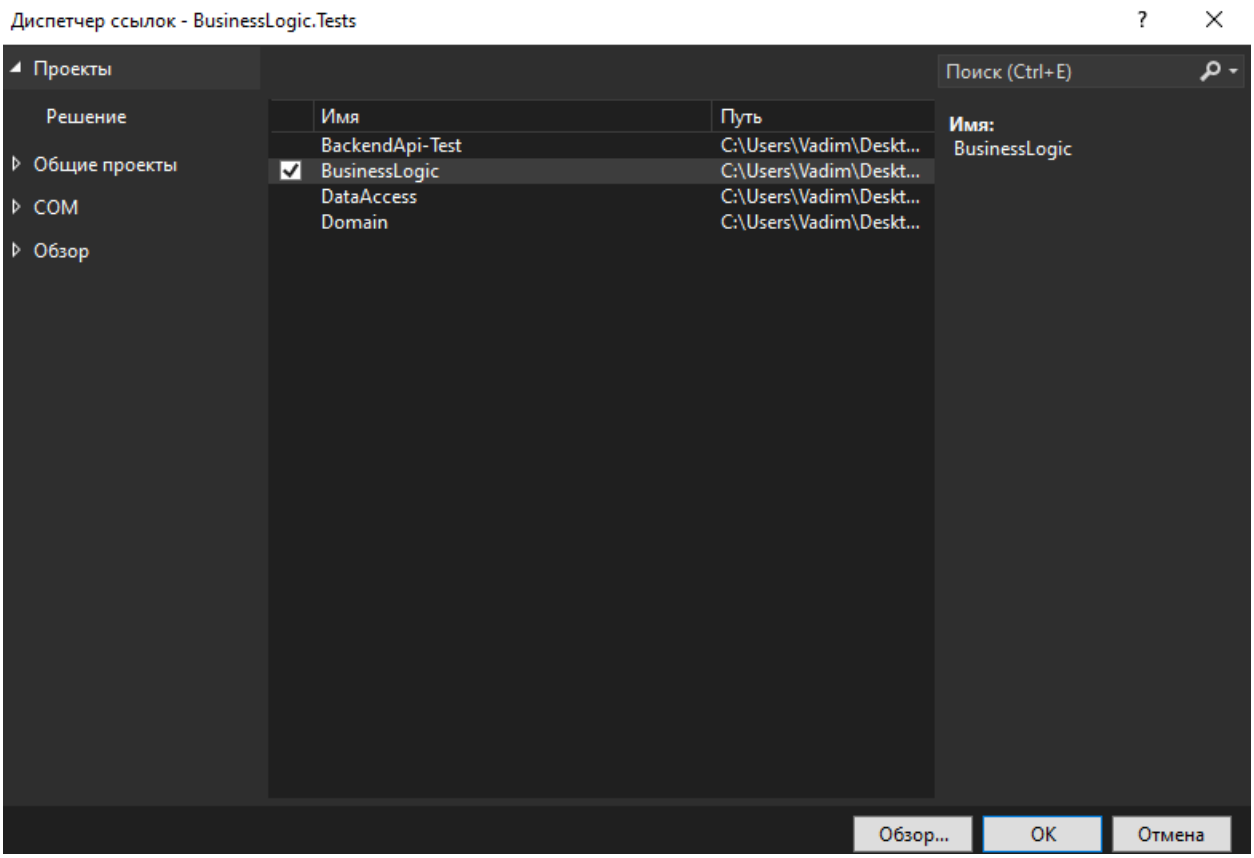


Однако в связи с тем, что мы не включили в качестве зависимости проект BusinessLogic, у нас появились следующие ошибки.

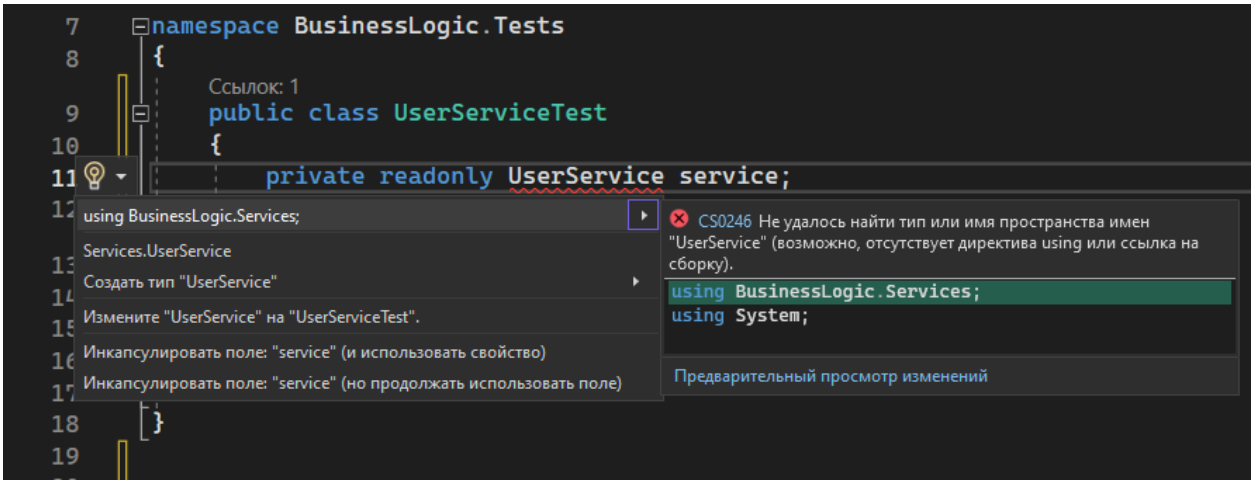
Добавим зависимость проекту



Укажем проект BusinessLogic



Исправим ошибку при помощи лампочки



Теперь у нас появилась следующая проблема. А именно то, что данный сервис требует в качестве атрибута интерфейс IRepositoryWrapper (То есть зависит напрямую от данного модуля)

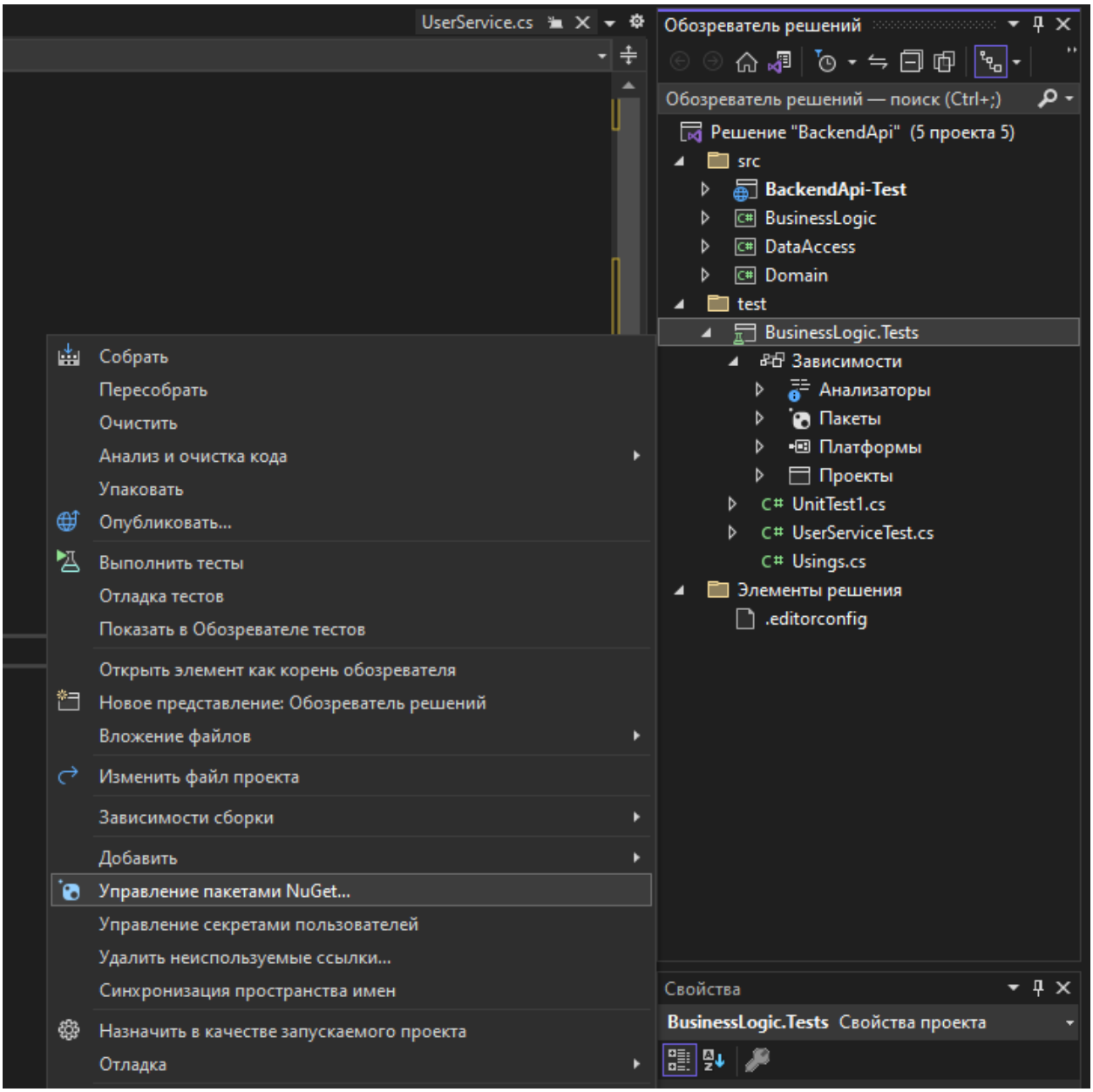
Так как принцип юнит-тестирования гласит о независимости модулей друг от друга - воспользуемся инструментов для создания заглушек Moq.

ℹ️ **Moq** - это библиотека для создания макетов (mocks) объектов в тестах, которая используется вместе с фреймворком xUnit. Макеты объектов - это объекты, которые имитируют поведение реальных объектов, но не имеют реальной реализации. Они создаются для того, чтобы тестировать функциональность других объектов, которые зависят от этих макетов.

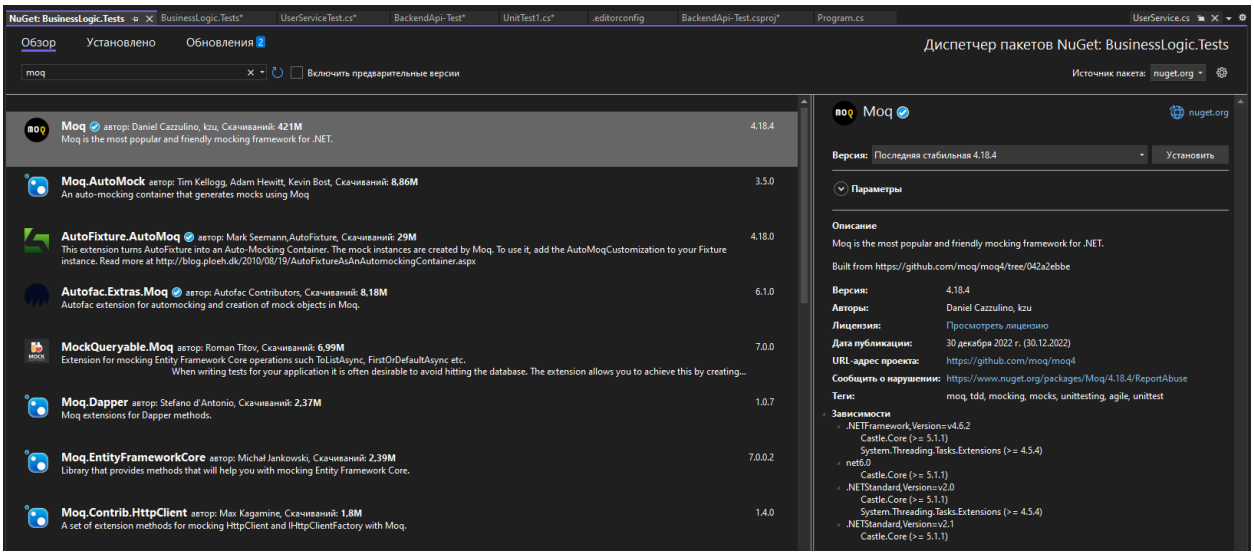
В контексте xUnit, Moq позволяет создавать макеты объектов и задавать им поведение, которое будет использоваться в тестах. Например, вы можете создать макет объекта базы данных и задать ему поведение, которое будет имитировать реальную базу данных. Это позволит вам тестировать функциональность вашего кода, который зависит от базы данных, без необходимости реально подключаться к базе данных.

Таким образом, использование Moq вместе с xUnit помогает упростить написание тестов и ускорить процесс разработки, так как вы можете тестировать функциональность своего кода без необходимости создавать реальные объекты.

Перейдите в управление пакетами Nuget



Найдите в поиске пакет Moq и установите его



Таким образом конструктор класса будет выглядеть следующим образом.

```
private readonly UserService service;
private readonly Mock<IUserRepository> userRepositoryMoq;

Ссылок: 0
public UserServiceTest()
{

    var repositoryWrapperMoq = new Mock<IRepositoryWrapper>();
    userRepositoryMoq = new Mock<IUserRepository>();

    repositoryWrapperMoq.Setup(x => x.User)
        .Returns(userRepositoryMoq.Object);

    service = new UserService(repositoryWrapperMoq.Object);
}
```

Проверка метода Create

Теперь создадим тесты, которые будут проверять работу метода Create

Проверка на null

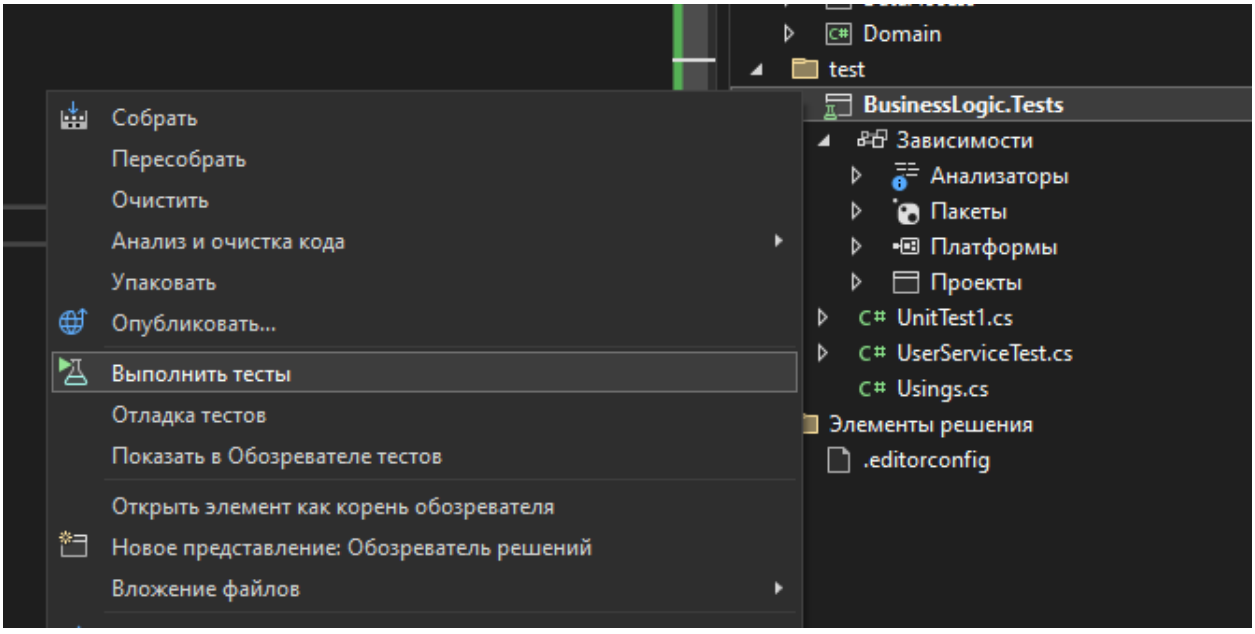
Первая проверка, которая обязательно должна быть - это проверка на null. Самая частая и болезненная проблема, которую игнорируют разработчики.

Для того, чтобы проверять результат работы метода - используется класс Assert.

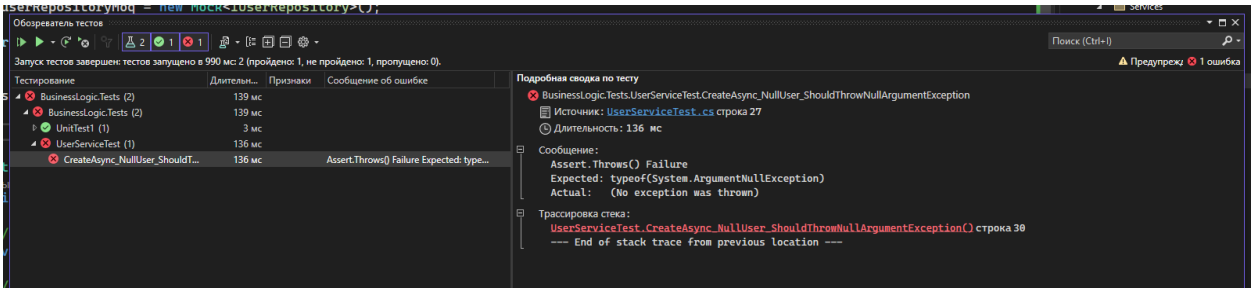
В нашем случае потребуется проверить работу исключения для null, поэтому используем метод ThrowsAny с указанием типа исключения ArgumentNullException. Так как метод Create является асихронным - метод используется соответствующий метод (с подписью Async)

```
[Fact]
| Ссылок: 0
public async Task CreateAsync_NullUser_ShouldThrowNullArgumentException()
{
    // act
    var ex = await Assert.ThrowsAnyAsync<ArgumentNullException>(() => service.Create(null));

    // assert
    Assert.IsType<ArgumentNullException>(ex);
    userRepositoryMoq.Verify(x => x.Create(It.IsAny<User>()), Times.Never);
}
```

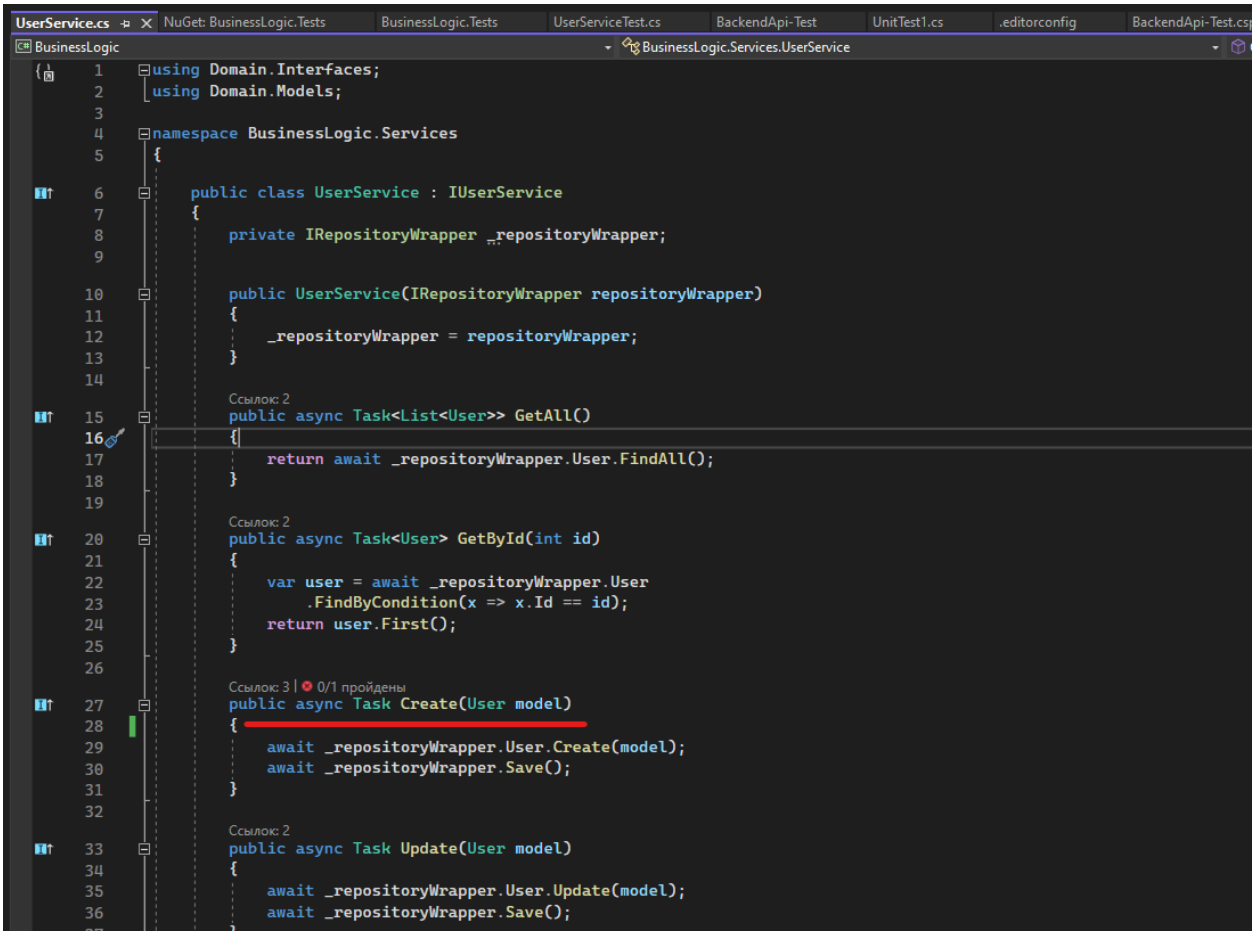
Проверим работу метода - запустим тест.



Результат проверки говорит - что тест упал.



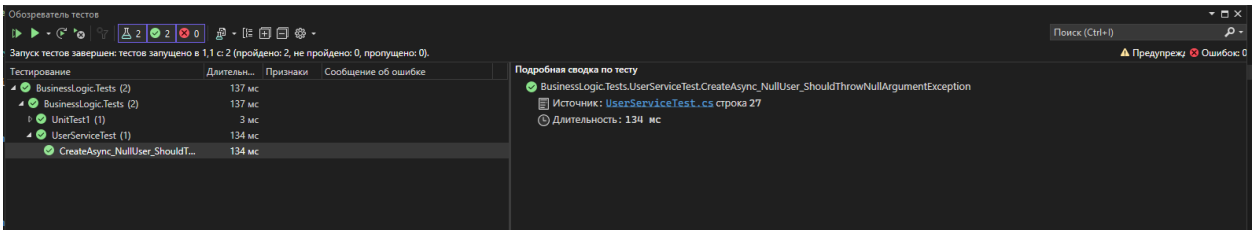
Поправим метод непосредственно в UserService



Добавим обработку на null с вызовом исключения throw new ArgumentNullException

```
Ссылок: 3 | 0/1 пройдены
public async Task Create(User model)
{
    if (model == null)
    {
        throw new ArgumentNullException(nameof(model));
    }

    await _repositoryWrapper.User.Create(model);
    await _repositoryWrapper.Save();
}
```

Теперь при запуске теста - результат положительный



Проверка работы валидации данных

Теперь приступим к проверке входных данных пользователя. Проверим работу добавления пользователя с неккоректными значениями.

```
[Fact]
| Ссылок: 0
public async Task CreateAsyncNewUserShouldNotCreateNewUser()
{
    // arrange
    var newUser = new User()
    {
        Firstname = "",
        Lastname = "",
        Middlename = "",
        Birthdate = DateTime.MaxValue,
        Login = "",
        Email = "",
        Password = ""
    };

    // act
    var ex = await Assert.ThrowsAnyAsync<ArgumentException>(() => service.Create(newUser));

    // assert
    userRepositoryMoq.Verify(x => x.Create(It.IsAny<User>()), Times.Never);
    Assert.IsType<ArgumentException>(ex);
}
```

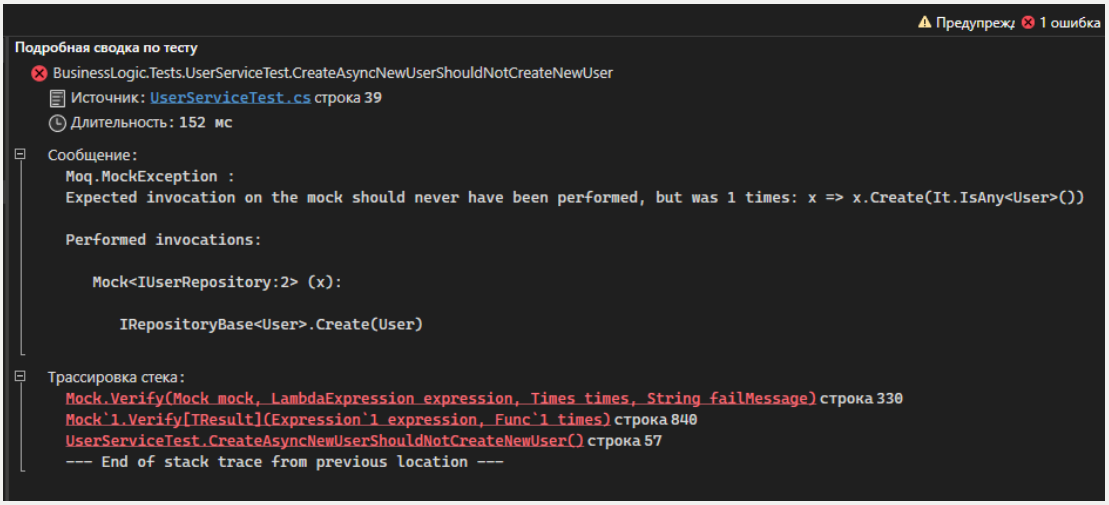
Ожидается, что если пользователь будет без имени или фамилии и так далее… То должна быть вызвана ошибка.

Также в данном тесте есть проверка на вызов метода для записи данных в БД.

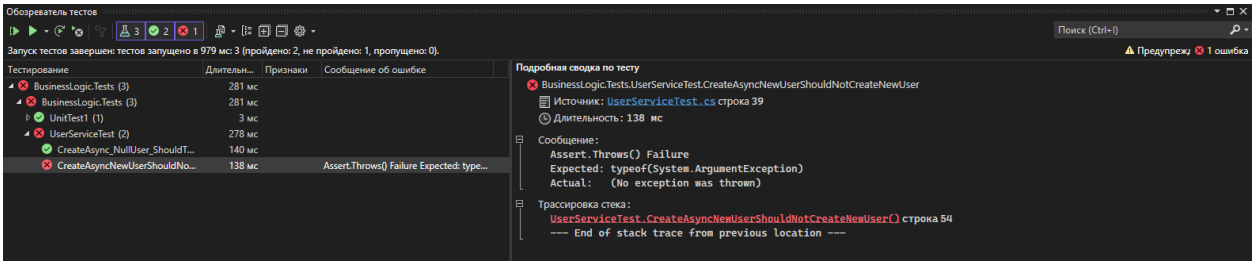
```
// assert
userRepositoryMoq.Verify(x => x.Create(It.IsAny<User>()), Times.Never);
```

Данный метод ни в коем случае не должен быть вызван, поэтому проверяем, что он не должен быть вызван хотя бы раз.

Пример ошибки, в случае, если данный метод будет вызван.



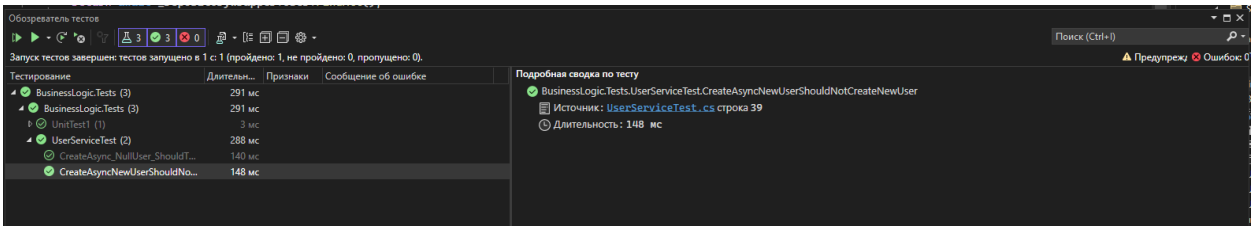
Проверим работу валидации. Ожидаемо, произошла ошибка:



Исправим работу для метода Create. Добавим валидацию для имени.

```
Ссылок: 4 | 1/2 пройдены
public async Task Create(User model)
{
    if (model == null)
    {
        throw new ArgumentNullException(nameof(model));
    }

    if (string.IsNullOrEmpty(model.Firstname))
    {
        throw new ArgumentException(nameof(model.Firstname));
    }

    await _repositoryWrapper.User.Create(model);
    await _repositoryWrapper.Save();
}
```

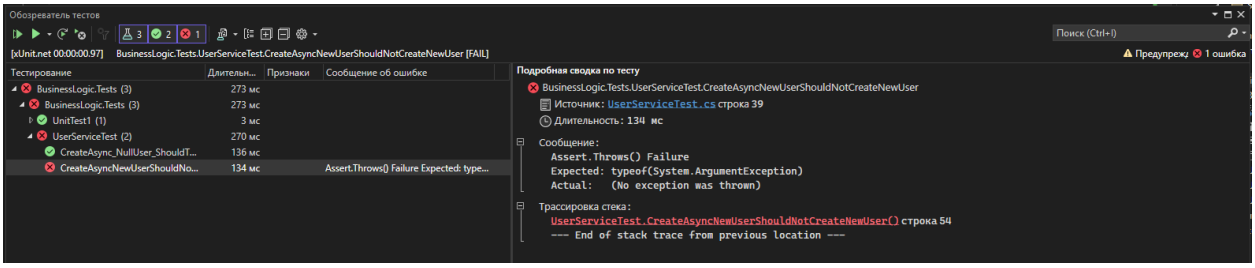
В таком случае, обозреватель тестов рапортует об успешной работе метода.



Однако, если мы поправим только фамилию

```
[Fact]
| Ссылок: 0
public async Task CreateAsyncNewUserShouldNotCreateNewUser()
{
    // arrange
    var newUser = new User()
    {
        Firstname = "Test",
        Lastname = "",
        Middlename = "",
        Birthdate = DateTime.MaxValue,
        Login = "",
        Email = "",
        Password = ""
    };
```

Ошибка никуда не денется - а пользователи и дальше будут с пустыми фамилиями, логином, паролем добавляться в базу данных.



Поэтому, чтобы проверить все возможные варианты, сделаем следующее

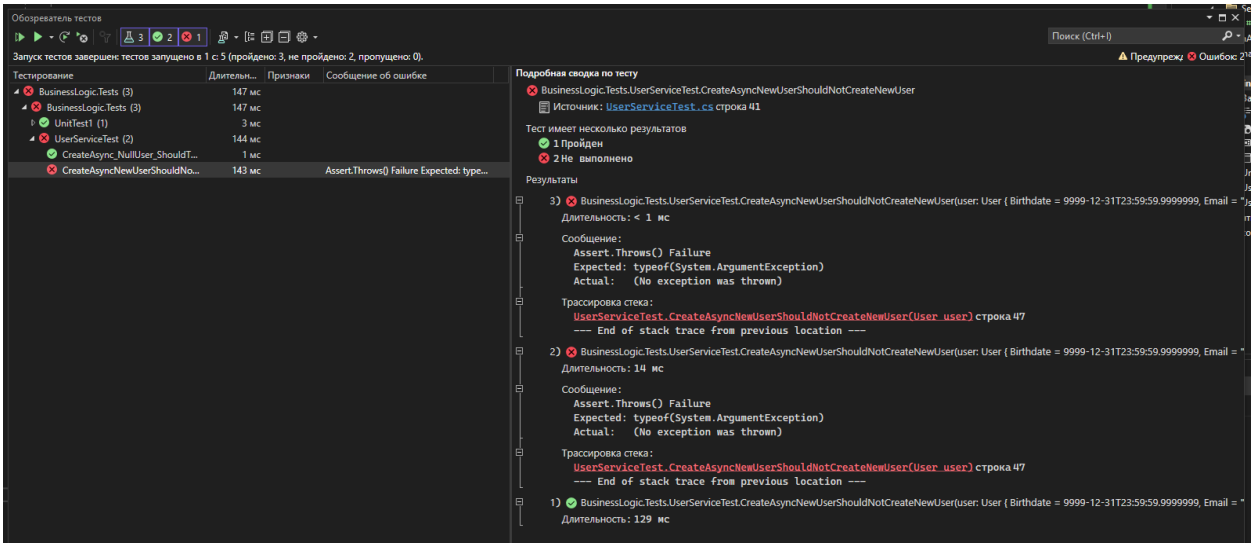
1. Переделаем тестовый метод в следующий вид (Заменим атрибут Fact на Theory)

```
[Theory]
[MemberData(nameof(GetIncorrectUsers))]
| Ссылок: 0
public async Task CreateAsyncNewUserShouldNotCreateNewUser(User user)
{
    // arrange
    var newUser = user;

    // act
    var ex = await Assert.ThrowsAnyAsync<ArgumentException>(() => service.Create(newUser));

    // assert
    userRepositoryMoq.Verify(x => x.Create(It.IsAny<User>()), Times.Never);
    Assert.IsType<ArgumentException>(ex);
}
```

2. Создадим метод, который будет выдавать комбинации с невалидными данными пользователей

```
Ссылок: 1
public static IEnumerable<object[]> GetIncorrectUsers()
{
    return new List<object[]>
    {
        new object[] { new User() { Firstname = "", Lastname = "", Middlename = "", Birthdate = DateTime.MaxValue, Login = "", Email = "", Password = "" } },
        new object[] { new User() { Firstname = "Test", Lastname = "", Middlename = "", Birthdate = DateTime.MaxValue, Login = "", Email = "", Password = "" } },
        new object[] { new User() { Firstname = "Test", Lastname = "Test", Middlename = "", Birthdate = DateTime.MaxValue, Login = "", Email = "", Password = "" } },
    };
}
```

3. Таким образом можно будет проверить все возможные сценарии, которые могут случится при создании пользователя.



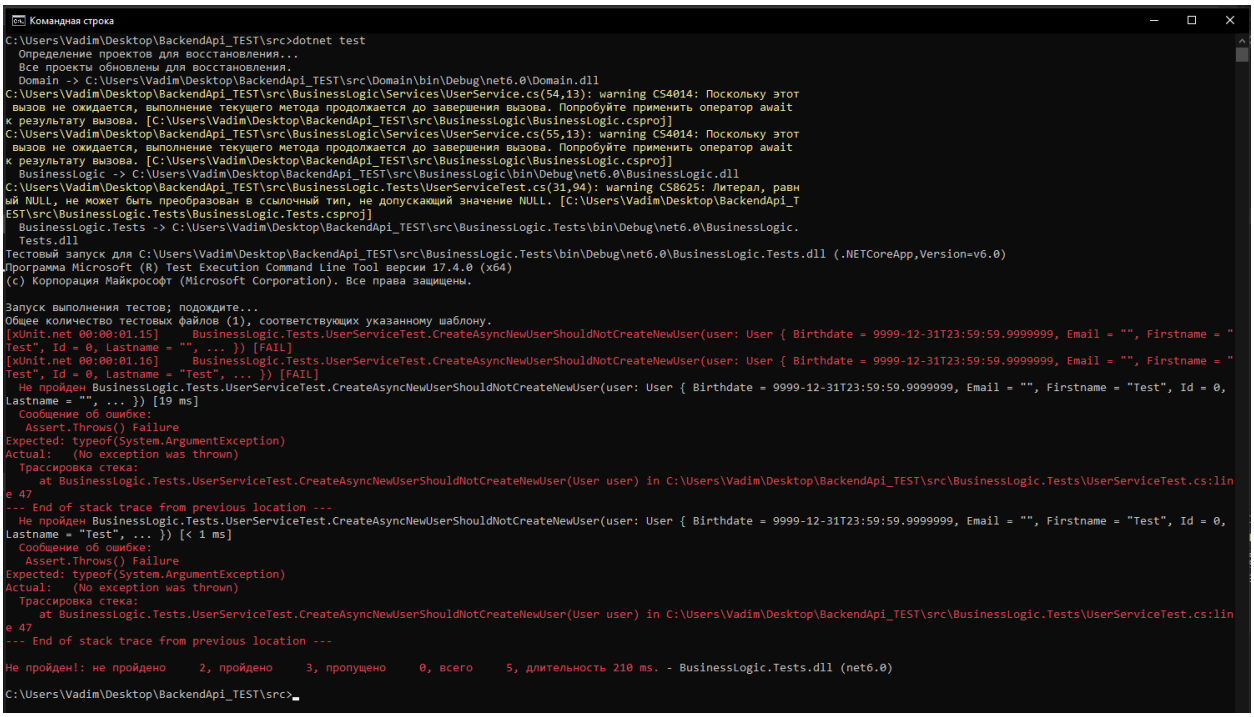
Проверка успешного создания пользователя

Помимо проверки на ошибки - также нужно создать тесты, которые будут проверять работу с корректными данными. Создадим тест, который проверит создание пользователя. В нашем случае - данные о пользователе не возвращаются. Поэтому просто проверим, что метод для добавления записи нового пользователя сработал.

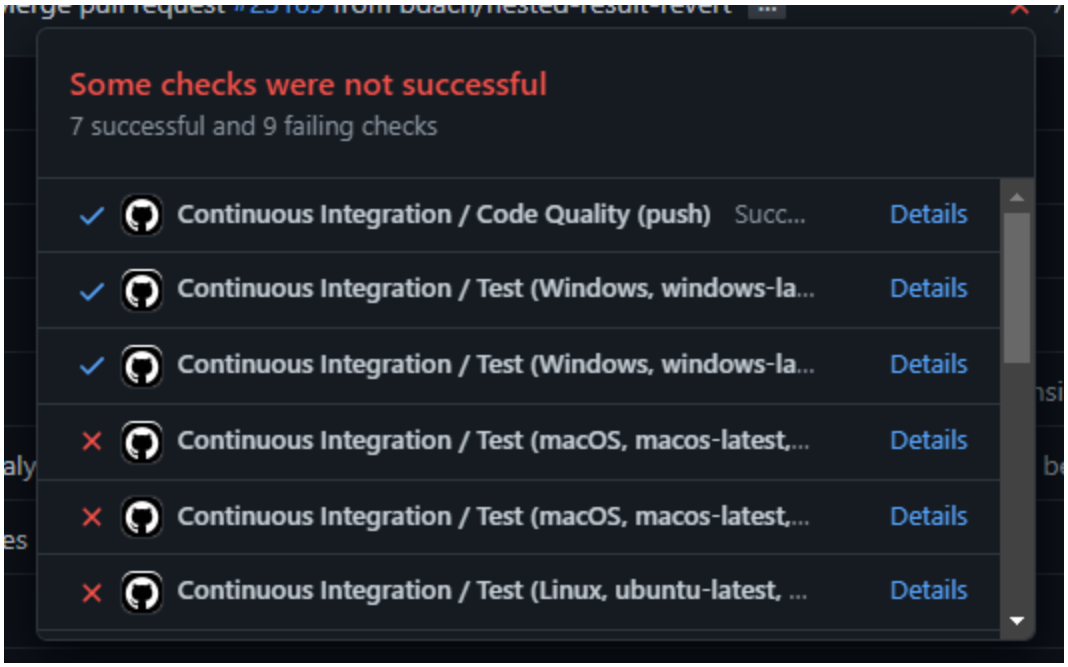
```
[Fact]
Ссылок: 0
public async Task CreateAsyncNewUserShouldCreateNewUser()
{
    var newUser = new User()
    {
        Firstname = "Test",
        Lastname = "Test",
        Middlename = "Test",
        Birthdate = DateTime.Now,
        Login = "Test",
        Email = "test@test.com",
        Password = ""
    };

    // act
    await service.Create(newUser);

    // assert
    userRepositoryMoq.Verify(x => x.Create(It.IsAny<User>()), Times.Once);
}
```

Тестирование через dotnet-cli

Помимо этого, также есть возможность проводить тесты через консоль. Для этого нужно написать команду dotnet test в папке проекта.



Таким образом, можно обеспечить проверку кода и на самом GitHub-репозитории.



Задание

1. Добавьте yml-файл, который будет выполнять тесты при помощи команды dotnet test в вашем репозитории на GitHub
2. Покройте тестами ваш проект для каждого сервиса, который обеспечивает работу с таблицами БД (Например товары, пользователи, сотрудники и т.д.).