**Памятка по демо**

(Быстрая навигация по оглавлению через Ctrl)

Оглавление

[Общие советы: 4](#_Toc120481118)

[Мы все сдадим 4](#_Toc120481119)

[Времени меньше, чем вы думаете 4](#_Toc120481120)

[Открывать сразу всё 4](#_Toc120481121)

[Что можно сделать во время проверки компьютеров: 4](#_Toc120481122)

[Бумажные шпоры 4](#_Toc120481123)

[Импорт данных: 4](#_Toc120481124)

[Создание БД 4](#_Toc120481125)

[Файлы .txt и .csv 4](#_Toc120481126)

[Импорт неструктурированного файла 5](#_Toc120481127)

[Файлы .xls и .xlsx 8](#_Toc120481128)

[Импорт данных 8](#_Toc120481129)

[Zip-архив и прочие картинки 12](#_Toc120481130)

[Скрипты 12](#_Toc120481131)

[Работа с данными в SQL Management Studio: 12](#_Toc120481132)

[Select запросы 12](#_Toc120481133)

[Основа (Select .. from): 12](#_Toc120481134)

[Условия (Where): 13](#_Toc120481135)

[Объединение (Group by): 14](#_Toc120481136)

[Упорядочивание (Order by): 14](#_Toc120481137)

[Добавление первичного ключа к таблице 14](#_Toc120481138)

[Замена названий на id 15](#_Toc120481139)

[Перенос данных в существующую таблицу 15](#_Toc120481140)

[Добавление внешнего ключа 15](#_Toc120481141)

[Код в Visual Studio: 15](#_Toc120481142)

[Разделение кода по папкам 15](#_Toc120481143)

[Создание класса со строкой подключения 16](#_Toc120481144)

[Как быстро получить строку подключения: 16](#_Toc120481145)

[Пример класса: 18](#_Toc120481146)

[Создание классов для БД 18](#_Toc120481147)

[Пример: 19](#_Toc120481148)

[Создание класса для работы с данными 19](#_Toc120481149)

[Пример: 19](#_Toc120481150)

[Шаблон функции: 19](#_Toc120481151)

[Перенос главной формы в папку и изменение стартовой формы 20](#_Toc120481152)

[Импорт изображений 20](#_Toc120481153)

[Создание формы авторизации 21](#_Toc120481154)

[Внешний вид: 21](#_Toc120481155)

[Функция для получения пользователя по его логину и паролю: 21](#_Toc120481156)

[Код формы: 22](#_Toc120481157)

[Шифрование пароля 22](#_Toc120481158)

[Проверка роли пользователя 24](#_Toc120481159)

[Загрузка вариантов (например, для фильтрации) в ComboBox 24](#_Toc120481160)

[Функция: 24](#_Toc120481161)

[Код: 24](#_Toc120481162)

[Вывод данных в DataGrid 25](#_Toc120481163)

[Функция для получения данных: 25](#_Toc120481164)

[Использование в формах: 26](#_Toc120481165)

[Вывод данных через UserControl 26](#_Toc120481166)

[Создание UserControl: 26](#_Toc120481167)

[Функция: 26](#_Toc120481168)

[Форма для отображения: 27](#_Toc120481169)

[Универсальный запрос на получение данных (учитывает фильтрацию, сортировку, поиск и страницы) 28](#_Toc120481170)

[Фильтрация: 28](#_Toc120481171)

[Поиск: 29](#_Toc120481172)

[Сортировка: 30](#_Toc120481173)

[Постраничный вывод: 31](#_Toc120481174)

[Код формы: 32](#_Toc120481175)

[TextBox: 32](#_Toc120481176)

[ComboBox: 33](#_Toc120481177)

[RadioButton: 33](#_Toc120481178)

[Кнопки пролистывания страниц: 33](#_Toc120481179)

[Как выбирать строку из DataGrid 33](#_Toc120481180)

[Создание формы добавления/редактирования 33](#_Toc120481181)

[Функция для добавления данных 34](#_Toc120481182)

[Функция для обновления данных 34](#_Toc120481183)

[Код формы: 35](#_Toc120481184)

[Генерация рандомного ID 36](#_Toc120481185)

[Удаление данных 37](#_Toc120481186)

[Функция: 37](#_Toc120481187)

[Код в форме: 37](#_Toc120481188)

[Динамическая работа с изображениями 38](#_Toc120481189)

[Функция для отображения изображений 38](#_Toc120481190)

[Как открывать проводник для поиска изображения 39](#_Toc120481191)

[Функция для сохранения изображения в папку 39](#_Toc120481192)

[Обращение к реальному времени (доброе утро/день/вечер из второго пробника) 39](#_Toc120481193)

[Unit-тесты 40](#_Toc120481194)

[Создание проекта с тестами: 40](#_Toc120481195)

[Метод 1 (более правильный): 40](#_Toc120481196)

[Метод 2 (более быстрый): 41](#_Toc120481197)

[Примеры тестов: 43](#_Toc120481198)

[Ожидается определенное значение(Assert.AreEqual(expected, result);) 43](#_Toc120481199)

[Ожидается экземпляр определенного типа(Assert.IsInstanceOfType(result, type);) 43](#_Toc120481200)

[Ожидается не null(Assert.IsNotNull(result);) 43](#_Toc120481201)

[Ожидается true(Assert.IsTrue(result);) или ожидается false(Assert.IsFalse(result);) 43](#_Toc120481202)

[Ожидание ошибки (Assert.ThrowsException<ExceptionType>(function);) 44](#_Toc120481203)

[Загрузка на гит с помощью команд 44](#_Toc120481204)

# Общие советы:

Мы все сдадим. Лучше не думать о другом, потому что лишнее волнение тратит силы, которые могли бы быть потрачены более эффективно. Если волнение начинает полностью накрывать, переключитесь на что угодно. Попробуйте порадовать себя: любой вкусный перекус, серия сериала, партия в игру, пара глав книги, просто сон – сделайте то, что доставляет вам удовольствие. Если все равно не получается успокоиться – нет ничего плохого в том, чтобы выпить какое-нибудь успокоительное (соблюдая дозировку, понятное дело). Перечислять не буду, сами что-нибудь да найдете.

Времени меньше, чем вы думаете. Это касается и подготовки, и самого демо. Если чувствуете, что начинаете застревать на каком-либо задании – пропустите его. Пусть у вас будет не самый лучший импорт, зато появится чуть больше форм в приложении. Не помните, как шифровать пароли – пропустите шифрование, может, вспомните позже.

Открывать сразу всё может быть хорошей идеей, особенно если вы понимаете, что будете делать несколько вещей параллельно. Но не забудьте проверить, может ли компьютер вытянуть одновременно визуалку и management studio.

## Что можно сделать во время проверки компьютеров:

Это более общие шаги, которые будут применимы к любому проекту. Да, два часа на них – это очень много, но зато они работают для всего.

Во-первых, создать базу данных. Ничего внутрь положить не получится, но хотя бы будет, куда складывать. Если вы собираетесь использовать какие-нибудь свои функции, можно их тоже записать сейчас.

Во-вторых, создать проект. Скорее всего, так и будет выглядеть проверка работоспособности, поэтому стоит этим воспользоваться. Создайте папки, перенесите стартовую форму, создайте класс со строкой подключения (БД уже есть, поэтому получится подключиться). Можете, опять же, принести какие-нибудь универсальные функции, например, работу с изображениями или шифрование.

В-третьих, сделайте тест-проект. Опять же, да, туда ничего не положить сейчас, кроме очень общих тестов, но меньше времени потратите на создание во время работы. Сразу сделайте 5 файлов, потом дадите имена.

Разумеется, если вариант уже будет, постарайтесь написать и сделать как можно больше по делу. Например, импортируйте таблицы, но не правьте их. Сделайте нужные формы, разместите элементы, попробуйте сделать переходы между ними. Если вы выносите функции в отдельный класс, попробуйте представить, что вам понадобится и просто насоздавать пустых функций с нужными именами. В тестах можете написать имена этих функций. Два часа – это достаточно времени, чтобы повысить оценку.

Бумажные шпоры могут оказаться полезными, поскольку мы не можем быть уверены в том, что на компьютерах будет помощь (а её, скорее всего, не будет). Только не пытайтесь записать вообще всё, столько не спрячете. Лучше записывать те функции, которые IDE не подскажет (а оно умное, даже очень, и было бы ещё умнее, если бы у нас стояла последняя визуалка, но увы), например, шифрование пароля, открытие проводника для поиска картинок, какие-то сложные запросы.

# Импорт данных:

## Создание БД

Перед импортом важно не забыть создать базу данных:

create database \*название базы\*

Например:

create database DemoExplanation

## Файлы .txt и .csv

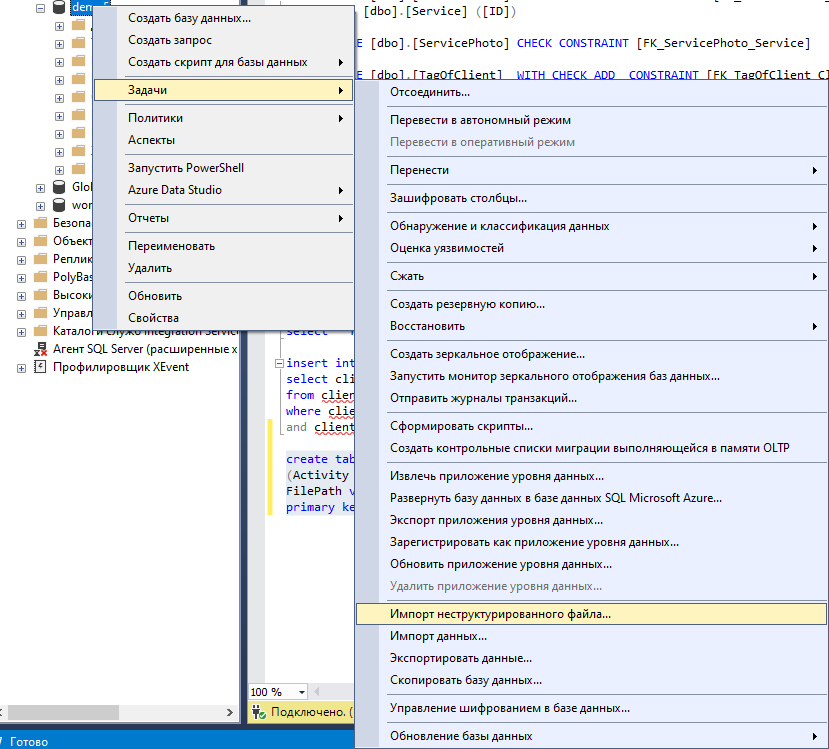
Если даны файлы **.txt** или **.csv**, в них, скорее всего, нужно что-то исправлять. Самые частые ошибки: **дата, пол, ФИО, стоимость, время, *единицы измерения там, где они не нужны***. Самое противное – американский формат даты (мм.дд.гггг). Его сложно заметить, но он легко испортит дальнейшую работу. И в стандартном блокноте, и в Notepad`е можно просто выделять нужный фрагмент и нажимать **Ctrl+H** (замена). Импорт этих файлов делается через **Импорт неструктурированного файла**.

При редактировании данных часто имеет смысл просмотреть их все, чтобы не делать лишнюю работу (Например, в одном из вариантов длительность услуги была нужна в секундах, но понять это можно было только из скрипта).

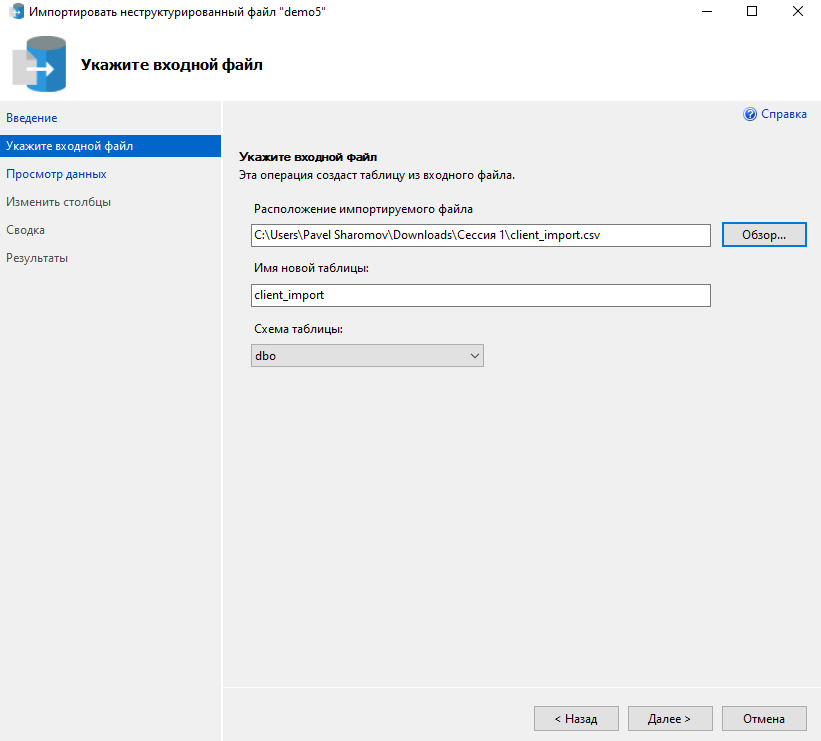
!!! НЕКОТОРЫЕ ДАННЫЕ ЛУЧШЕ ОТРЕДАКТИРОВАТЬ РУКАМИ, НАПРИМЕР, ЗНАК РУБЛЯ ПРИ ПОПЫТКЕ ЗАМЕНИТЬ ЕГО НА ПУСТОТУ УБИРАЕТ ВСЕ ПРОБЕЛЫ В ФАЙЛЕ !!!

## Импорт неструктурированного файла

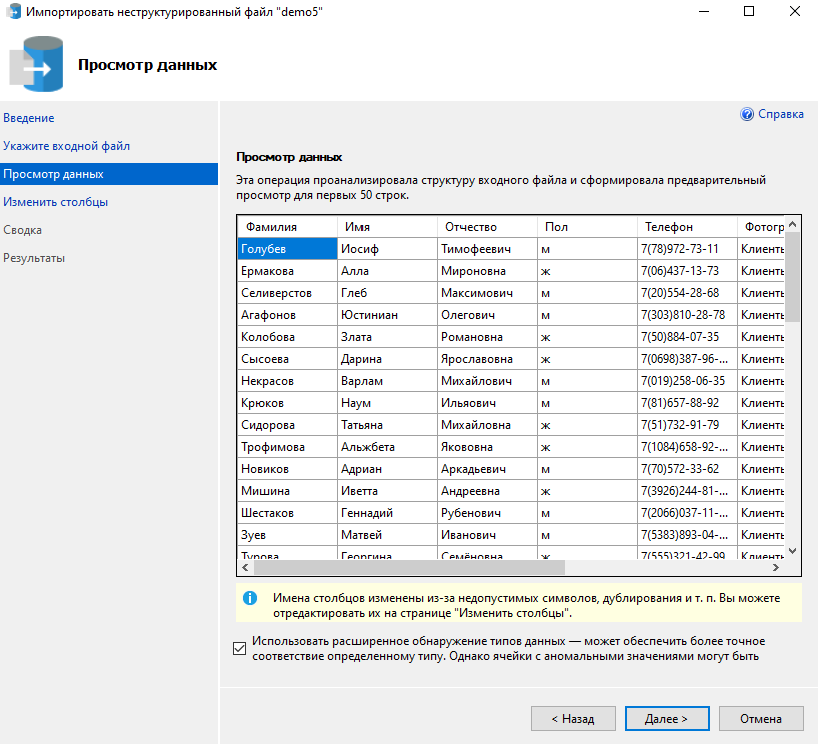
Чтобы сделать импорт неструктурированного файла (.txt и .csv), нажмите правой кнопкой по имени вашей БД, в меню выберите «Задачи» -> «Импорт неструктурированного файла».



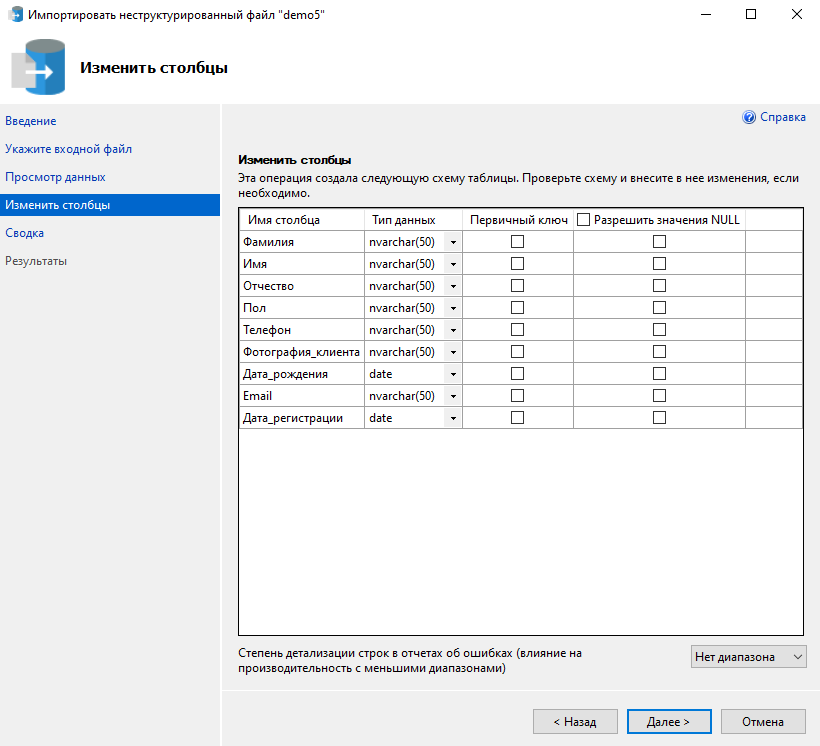
Выберите нужный файл. Если вас не устраивает название таблицы, можете его поменять. Если у вас есть другая схема, которую вы создали для демо (но зачем), выберите её. Если вы не знаете, что такое схема или не делали новых, оставьте dbo.



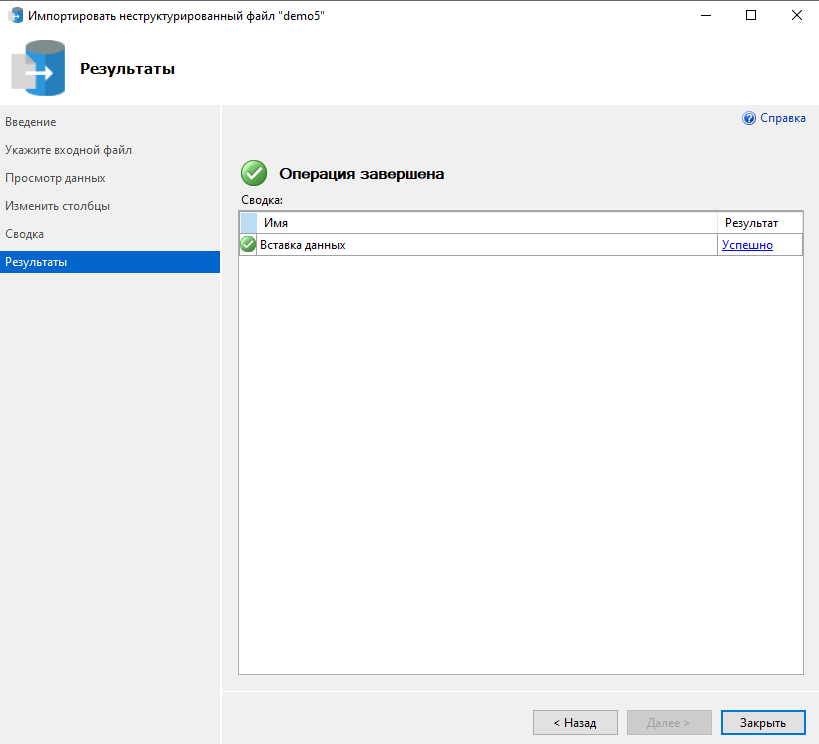
Тщательно просмотрите данные. Здесь можно увидеть, пропустили ли вы что-нибудь, когда редактировали документ, потому что в случае ошибки полетят все столбцы. Если сбилась кодировка – поменяйте её в notepad`е на LE-BOM.



Проверьте, чтобы типы столбцов были нужными вам. Если они не совпадают (например, столбец с числом подписан как текстовый), значит, скорее всего, есть ошибка в данных. В некоторых пределах типы столбцов можно редактировать, например, заменить int на любой другой численный формат, в т.ч. money. Если вы собираетесь дальше работать с этой таблицей, не перенося данные, проставьте первичные ключи (по возможности). Первичным ключом может быть артикул, ИНН или КПП, но они обычно длинные, и с ними может быть неудобно работать. Проставьте, какие столбцы могут иметь значение NULL, если такие есть.



Сводка просто повторит базовые сведения: что за файл, куда импортируем, под каким именем. Ее, в большинстве случаев, можно почти полностью игнорировать. Нажмите «Готово» и ждите результатов. Если нет ошибок – поздравляю, импорт закончен. Если начинают появляться ошибки, есть, обычно, три причины: проблема в данных, попытка каким-то странным способом изменить тип столбца и проблемы самой программы. С первыми двумя вы можете справиться, ради третьего зовите наблюдающих, и пытайтесь использовать отсрочку с пользой.



## Файлы .xls и .xlsx

В файлах **.xls** ошибок в данных (тех, которые можно исправить через замену) обычно **НЕТ**. Их можно просто импортировать (через **Импорт данных**).

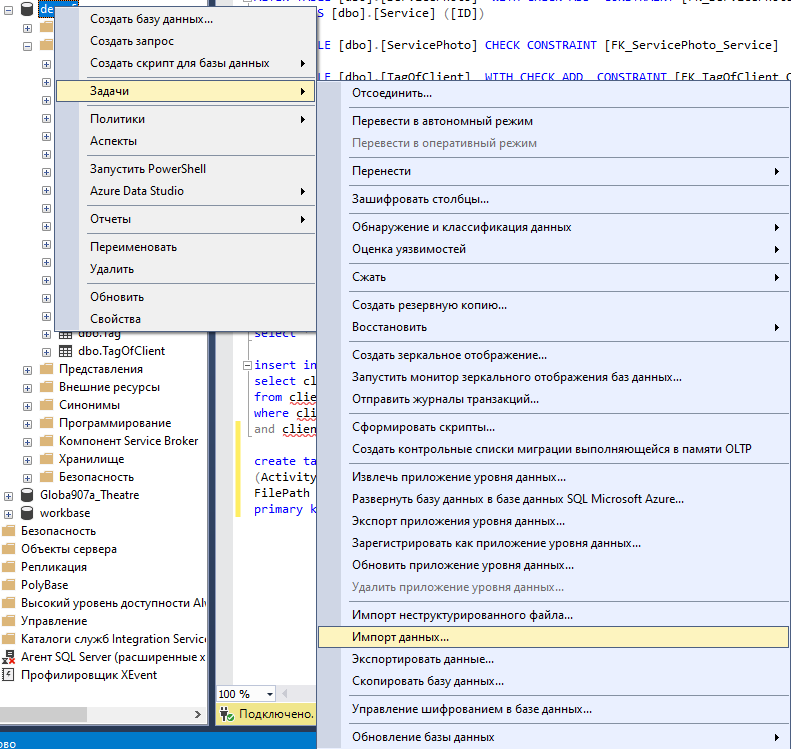
Если не получается импортировать файл **.xlsx**, можно сохранить его как **.xls** (то есть как **более старый формат**).

Если в задании даны **исключительно .xls** файлы, то ошибки в них **будут**. Ctrl+H работает и в Экселе, поэтому замена будет быстрой. Стандартные ошибки такие же, как и в неструктурированных файлах (.txt и .csv).

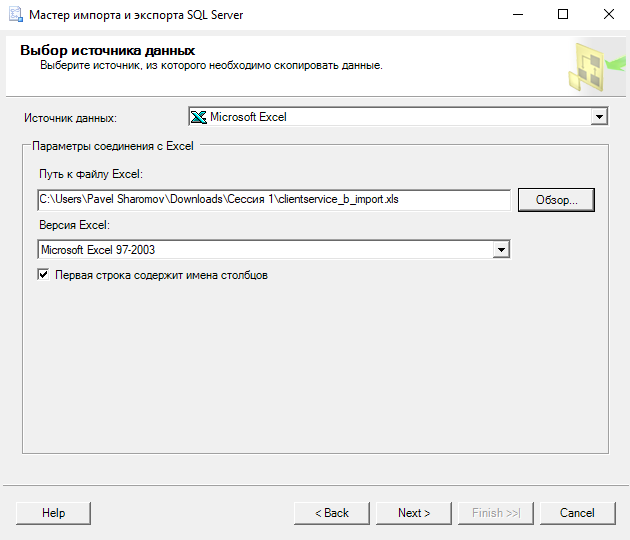
Иногда файлы сохраняются криво, т.е. не с первого столбца. Если есть чувство, что чего-то не хватает, посмотрите, можно ли пролистать вбок. Да, звучит очевидно, но в стрессовой ситуации такое сложно заметить, поэтому лучше лишний раз сказать, что такое может быть.

## Импорт данных

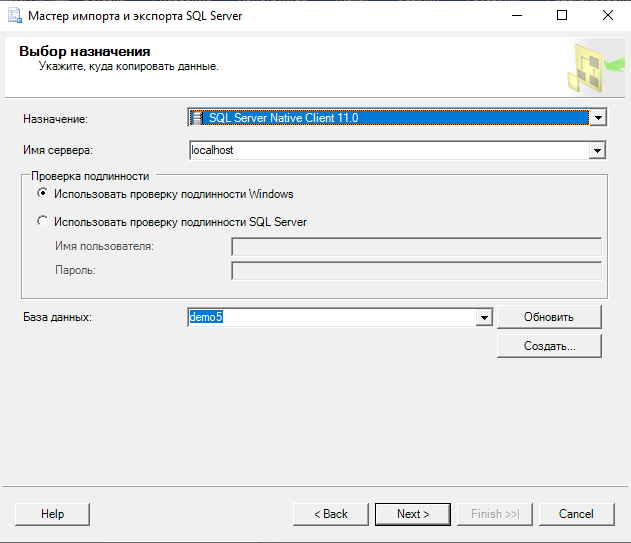
Чтобы сделать импорт данных (.xls и .xlsx), нажмите правой кнопкой по имени вашей БД, в меню выберите «Задачи» -> «Импорт данных».



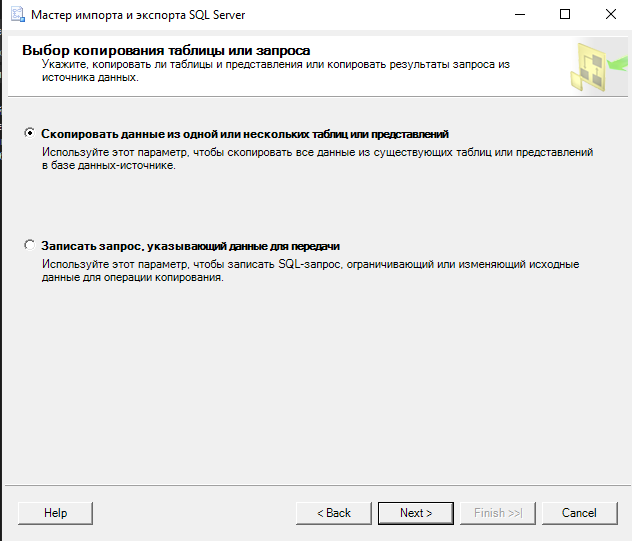
В качестве источника данных выберите «Microsoft Excel» и найдите нужный файл. Если в первой строке даны названия столбцов, отметьте checkbox.



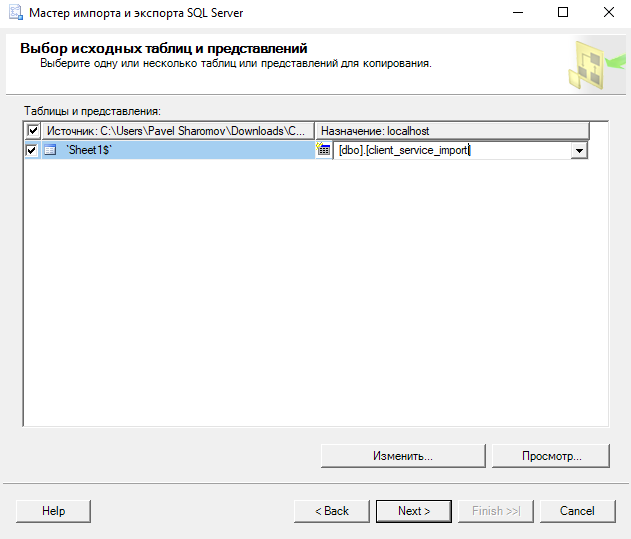
В качестве назначения выберите «SQL Server Native Client» (находится внизу списка). Обычно ничего дальше менять не надо, имя сервера и БД подставятся автоматически.



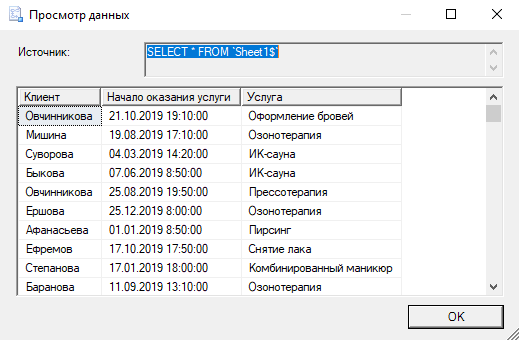
Если вы уверены, что сможете сразу написать нужный запрос, который вытащит ваши данные, выберите второй вариант. Если нет, оставляйте первый.



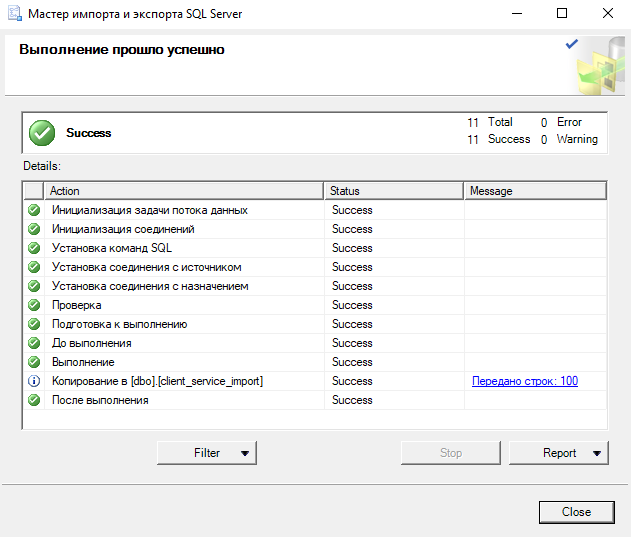
Далее нужно выбрать, какие листы из эксель файла вы хотите перенести в базу. Здесь же можно дать нормальное название таблицам *(или выбрать существующую: нужна проверка*).



Если нажать на кнопку «Просмотр», можно посмотреть, в каком виде будут вставлены данные. Советую это сделать, чтобы ещё раз убедиться в том, что вы все отредактировали.



Дальше можно просто со всем соглашаться и нажать «Finish». Если в конце всё успешно, то отлично, импорт закончен. Если нет – скорее всего, либо ошибка в данных, либо вы что-то не так выбрали/настроили во время импорта, либо Management studio не может достать файл, тогда его нужно попробовать импортировать из другого места. Обычно он говорит. Где случилась ошибка, а иногда и какая. Если же это точно проблема самого Management studio, зовите наблюдающих и, опять же, используйте отсрочку с пользой.



## Zip-архив и прочие картинки

Zip-архив просто распаковывается и загружается в проект в Visual Studio, поэтому о нем можно забыть на время импорта.

Иногда количество фотографий не совпадает с количеством записей в таблице. Если их больше, можно не волноваться, при необходимости их потом можно удалить. Если меньше – продублировать и подписать новые номера.

## Скрипты

Если даны готовые скрипты, лучше в них не вмешиваться, особенно если ничего не понятно. Если даны **И запрос, И таблицы для импорта**, обычно проще дополнительно импортировать таблицы, а потом засунуть их данные в те, которые создаются скриптом.

Если скрипт не работает, то надо проверить, запускаете ли вы его из своей базы данных, или из master. Из второго работать не будет.

# Работа с данными в SQL Management Studio:

Здесь будут самые частые операции с данными, которые могут встретиться в демо.

## Select запросы

Эта часть больше пригодится для работы в VS, однако лучше объяснить сразу. Здесь примеров не будет, только шаблоны. Если что-то указано во множественном числе, значит, таких элементов МОЖЕТ быть несколько, но не обязательно. Если указано в единственном – то ТОЛЬКО в единственном.

Каждый раз, когда пишете какой-нибудь запрос, с которым должна работать VS, проверяйте его сначала в Management Studio.

### Основа (Select .. from):

SELECT \*столбцы\*   
FROM \*таблицы\*;

Если необходимо вытащить все данные, то можно использовать **\*:**

SELECT \* FROM \*таблицы\**;*

Если нужны только различные значения, используется **select distinct**:

SELECT DISTINCT \*столбцы\*  
FROM \*таблицы\**;*

Используя **select top n** можно выбрать первые n значений (или первые n% значений, если после n использовать **percent**):

SELECT TOP n\*столбец\*  
FROM \*таблицы\*WHERE\*условия\**;*

Чтобы выбрать минимальное/максимальное значение столбца используется **min/max:**

SELECT MIN/MAX(\*столбец\*)  
FROM \*таблицы\*  
WHERE \*условия\*;

**Count** подсчитывает количество записей:

SELECT COUNT(\*столбец\*)  
FROM \*таблицы\*  
WHERE \*условия\*;

**Avg** подсчитывает среднее значение для **численных** значений:

SELECT AVG(\*столбец\*)  
FROM \*таблицы\*  
WHERE \*условия\*;

**Sum** подсчитывает сумму для **численных** значений:

SELECT SUM(\*столбец\*)  
FROM \*таблицы\*  
WHERE \*условия\*;

**As** позволяет дать другое имя значению внутри запроса (также работает для таблиц):

SELECT \*столбец\* AS \*новое имя\*  
FROM \*таблицы\*;

### Условия (Where):

SELECT \*столбцы\*  
FROM \*таблицы\*  
WHERE \*условия\**;*

Несколько условий можно использовать с **and, or** и **not** (примеры приводить не буду, они довольно очевидные).

Проверка является ли значение **null**:

SELECT \*столбцы\*  
FROM \*таблицы\*WHERE \*проверяемый столбец\* IS (NOT) NULL;

Чтобы сравнить строку с шаблоном используется **Like:**

SELECT \*столбцы\*  
FROM \*таблицы\*  
WHERE \*проверяемый столбец\* LIKE \*шаблон\*;

В шаблонах могут использоваться следующие специальные символы (чаще всего используется %):

|  |  |  |
| --- | --- | --- |
| Символ | Описание | Пример |
| % | Любое количество любых символов | **bl%** находит bl, black, blue и blob |
| \_ | Один любой символ | **h\_t** находит hot, hat и hit |
| [] | Один любой символ из находящихся внутри скобок | **h[oa]t** находит hot и hat, но не hit |
| ^ | Один любой символ, не находящийся в скобках | **h[^oa]t** находит hit, но не hot или hat |
| - | Один любой символ из указанного в скобках диапазона | **c[a-b]t** находит cat и cbt |

**In** проверяет на соответствие любому значению в скобках (также работает с другим select запросом):

SELECT \*столбцы\*  
FROM \*таблицы\*  
WHERE \*проверяемый столбец\* IN (\*значения\*);

SELECT \*столбцы\*  
FROM \*таблицы\*  
WHERE \*проверяемый столбец\* IN (SELECT запрос);

**Between** проверяет, находится ли значение между двумя другими:

SELECT \*столбцы\*  
FROM \*таблицы\*  
WHERE \*проверяемый столбец\* BETWEEN \*значение 1\* AND \*значение 2\*;

### Объединение (Group by):

Группировка в большинстве случаев не нужна, пока SQL не начнет ругаться на какой-нибудь сложный запрос:

SELECT \*столбцы\*  
FROM \*таблицы\*  
WHERE \*условия\*  
GROUP BY \*столбцы, по которым идет группировка\*;

### Упорядочивание (Order by):

**Asc/Desc** упорядочивает по возрастанию/убыванию. По умолчанию стоит Asc, поэтому его указывать не обязательно:

SELECT \*столбцы\*  
FROM \*таблицы\*

WHERE \*условия\*  
GROUP BY \*столбцы, по которым идет группировка\*  
ORDER BY \*столбцы, по которым идет сортировка\* ASC/DESC;

**Offset..Fetch next** используется для постраничного вывода. **Offset** пропускает указанное количество рядов, **fetch next** возвращает заданное количество рядов. **Fetch** нельзя использовать в одном запросе с **top** (во вложенном запросе top работает нормально):

SELECT \*столбцы\*  
FROM \*таблицы\*

WHERE \*условия\*  
GROUP BY \*столбцы, по которым идет группировка\*

ORDER BY \*столбцы, по которым идет сортировка\*  
OFFSET \*число пропускаемых рядов\* ROWS FETCH NEXT \*число возвращаемых рядов\* ROWS ONLY

## Добавление первичного ключа к таблице

При создании таблицы соответствующий столбец прописывается как

\*Название поля\* \*тип\* primary key,

Например,

ID int identity(1,1) primary key not null,

Для добавления **столбца к существующей таблице** используется alter table

alter table \*таблица\*

add \*название столбца\* \*тип столбца\*

Пример добавления столбца с первичным ключом:

alter table Client

add ID int identity(1,1) primary key

Да, он будет стоять не первой колонкой, но функцию свою выполнять будет. Просто прописывайте столбцы в запросах, чтобы наверняка. Identity работает для уже существующих таблиц.

Если нужно сделать уже существующее поле (или несколько) первичным ключом, можно воспользоваться

alter table \*таблица\*

add primary key (\*перечисление столбцов\*)

Также конструкцией primary key (\*перечисление столбцов\*) можно пользоваться при создании таблицы, например:

create table ActivityRecources

(Activity int not null references Activity(ID),

FilePath varchar(255),

primary key (Activity, FilePath))

## Замена названий на id

В таком случае проще создать новую таблицу с правильными не только данными, но и типами в столбцах, а потом вставить в неё результаты подобного запроса:

insert into \*новая таблица\* (\*поля новой таблицы\*)

select \*таблица с ID\*.ID, \*старая таблица\*.\*данные, которые переносятся\*

from \*старая таблица\*, \*таблица с ID\*

where \*старая таблица\*.\*название\* = \*таблица с ID\*.\*название\*

Пример для замены двух столбцов:

insert into ClientService (ClientID, StartTime, ServiceID)

select client.ID, client\_service\_import.[Начало оказания услуги], [Service].ID

from client\_service\_import, Client, [Service]

where client\_service\_import.[Клиент] = client.LastName

and client\_service\_import.[Услуга] = [Service].Title

## Перенос данных в существующую таблицу

Этот запрос используется, например, когда были даны скрипт и данные для импорта, либо, если есть время/желание или так понятнее, можно так перенести в новые таблицы все данные.

insert into \*новая таблица\* (\*поля таблицы\*)

select (\*соответствующие поля таблицы\*) from \*таблица с импортом\*

Пример вставки:

insert into [Service] (Title, DurationInSeconds, Cost, Discount)

select \* from service\_b\_import

## Добавление внешнего ключа

**При создании таблицы** можно воспользоваться такой записью (максимально короткая для экономии времени:

\*Название поля\* \*тип\* references \*другая таблица\*(\*название столбца\*),

Например,

ParticipantID int not null references [User](ID),

Добавление внешнего ключа **к существующей таблице** делается через команду alter table

alter table \*название таблицы\*

add foreign key \*название столбца\* references \*другая таблица\*(\*название столбца\*)

Пример:

alter table Invoice

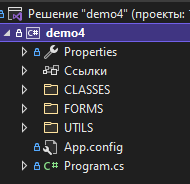
add foreign key (CustomerID) references Customer(CustomerID)

# Код в Visual Studio:

Все функции приведены для **WinForms без LINQ**, потому что это самый очевидный и прямолинейный вариант, ради которого обычно много думать не надо.

## Разделение кода по папкам

Не знаю, сколько конкретно это дает баллов, но это может быть полезно в том числе и просто для написания кода. Как только вы создали проект, сделайте 3 папки: **CLASSES, FORMS** и **UTILS**. В первой будут лежать классы по сущностям из БД, во второй все формы, в третьей строка подключения и, если вы его делаете, класс со всеми запросами.



Чтобы обратиться к классу в папке из другого места, можно написать **\*название папки\*.\*название класса\*** или **using \*название проекта\*.\*название папки\***.

Примеры:

private CLASSES.Task selectedTask = null;

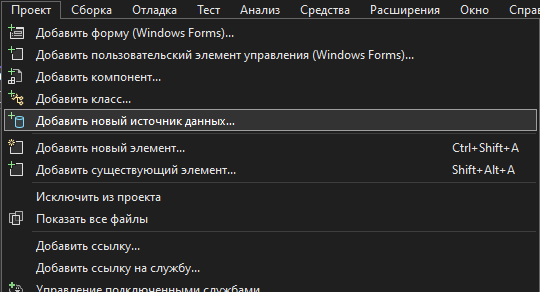
using demo4.UTILS;

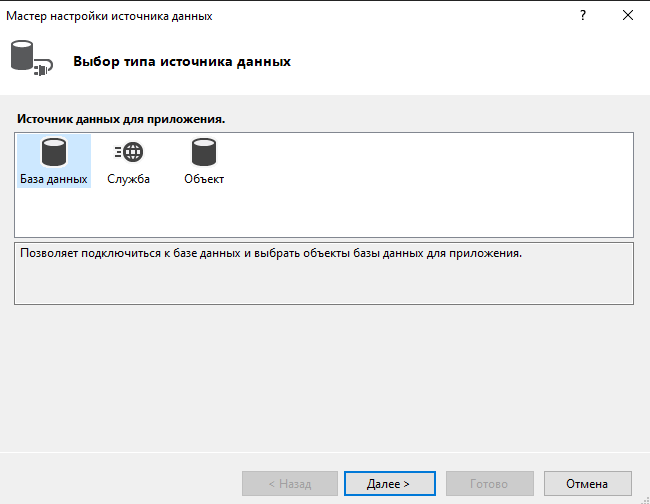
## Создание класса со строкой подключения

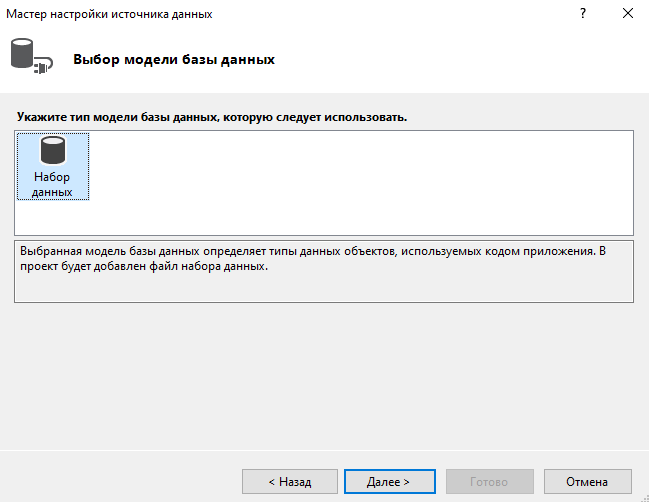
Этот класс делается в папке UTILS. Формально, он состоит из одной публичной статичной переменной, в которой лежит строка подключения к вашей БД. Не забывайте, что названия должны отражать суть.

### Как быстро получить строку подключения:

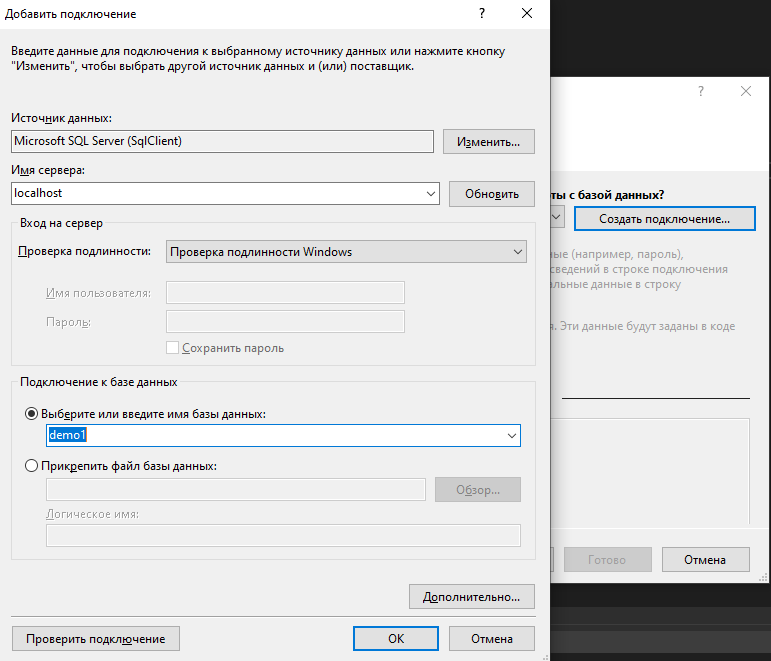
Найти «добавить новый источник данных». Если слева сбоку нет вкладки «Источники данных», можно найти сверху во вкладке «Проект»



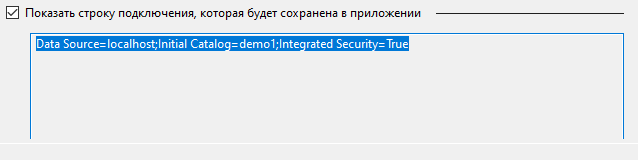




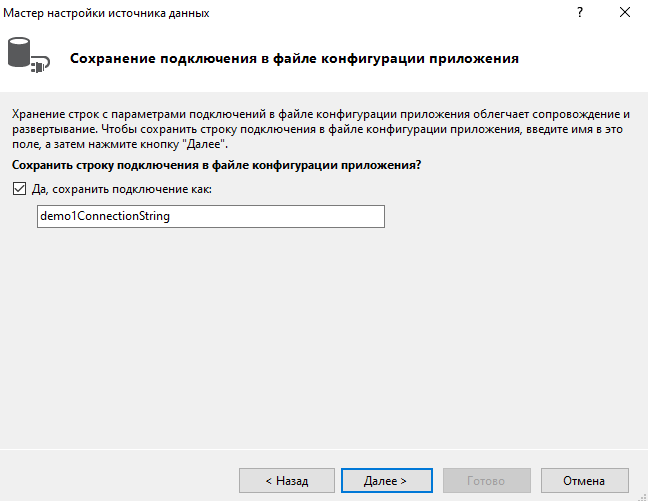
Если в списке не будет необходимой базы данных, нажать «Создать подключение…», в появившемся окне ввести имя сервера (на демо localhost не сработает, скрины сделаны на домашнем компьютере) и выбрать/ввести имя БД. Не забудьте проверить подключение. Если проверка не проходит, значит неправильно написано имя сервера.

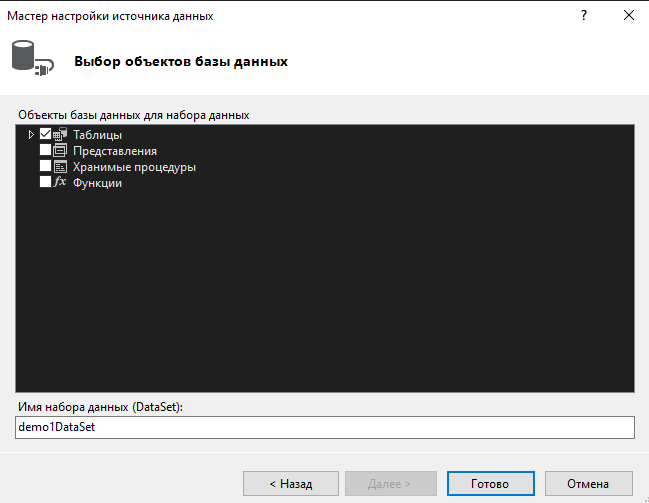


На окне с выбором появится название вашей БД. Если нажать эту галочку, появится строчка, которую можно скопировать.



Формально, на этом можно и закончить, но, если захочется, можно пойти дальше и добавить DataSet.





### Пример класса:

namespace demo4.UTILS

{

internal class ConnectionString

{

public static string ConnStr = @"Data Source=localhost;Initial Catalog=demo1;Integrated Security=True";

}

}

Если объявить элемент класса как static, это позволяет не создавать экземпляр класса, когда он вам нужен.

@ перед строкой означает, что она будет использоваться именно в таком виде, игнорируя все специальные символы, применяемые для форматирования (например, /), внутри.

## Создание классов для БД

По-хорошему, на каждую сущность (таблицу) в БД создается свой класс. На практике это может не понадобиться, поэтому, если чувствуете, что вам не хватает времени, делайте только те классы, которые необходимы.

Класс в программе просто повторяет все поля таблицы. Да, это всё. Как обычно, имена должны отражать суть каждого поля. Обратите внимание, что это **свойства** класса, а не просто переменные, поэтому они пишутся с заглавной буквы. У свойств есть более полная запись, но зачем тратить на это время, если краткая работает абсолютно также?

### Пример:

|  |  |
| --- | --- |
| Таблица (получена через «Выбрать первые 1000 строк»): | Класс: |
| SELECT TOP (1000) [ID]  ,[Password]  ,[FirstName]  ,[MiddleName]  ,[LastName]  ,[Login]  ,[IsDeleted]  FROM [demo1].[dbo].[User] | public class User  {  public int ID { get; set; }  public string Password { get; set; }  public string FirstName { get; set; }  public string MiddleName { get; set; }  public string LastName { get; set; }  public string Login { get; set; }  public bool IsDeleted { get; set; }  } |

Как класс, так и его свойства должны быть **public**, чтобы другие части кода могли с ними работать.

## Создание класса для работы с данными

Опять же, я не знаю, сколько баллов дадут за это (и дадут ли), но этот класс довольно хорошо «вычищает» код от запросов, оставляя только понятные названия функций. Еще такой подход позволяет спокойно писать только запросы или только код в формах, не задумываясь о другой части. В первом случае вы просто потом используете эти методы, во втором закрываете комментариями нерабочую часть и открываете, когда нужная функция появляется. К тому же, зная, где лежат все запросы, их легко редактировать.

**Все дальнейшие объяснения подразумевают, что у вас есть такой класс**. Если вы не хотите его делать, принцип работы функций будет похожий, достаточно заменить return на какое-либо нужное вам действие.

Класс создается в **UTILS**. Я всегда его называю DataWork, но это, понятное дело, дело предпочтения. К созданным using надо добавить **using System.Data.SqlClient;** (это получится единственный класс, в котором нужна эта библиотека) и **using \*название проекта\*.CLASSES** (для быстрого доступа). Сразу же можно сделать глобальную переменную для строки подключения (берется из класса со строкой). Также можно в целом иметь строку подключения в этом классе, но примеры берутся из готовых работ, которые я сейчас переделывать не собираюсь.

### Пример:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Data;

using System.Data.SqlClient;

using System.Security.Cryptography;

using demo4.CLASSES;

namespace demo4.UTILS

{

internal class DataWork

{

static string ConnStr = ConnectionString.ConnStr;

}

}

### Шаблон функции:

public static \*тип возвращаемого значения\* \*название\*(\*параметры\*)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = \*запрос\*;

\*действия с запросом\*

if (\*успешно\*)

{

return \*значение\*;

}

throw new Exception(\*текст исключения\*);

}

}

Исключение нужно, во-первых, чтобы визуалка не ругалась, что «не все пути к коду возвращают значение», когда функция любого типа, **кроме void**, во-вторых, так можно сразу задать сообщение, которое будет выводиться пользователю при ошибке.

## Перенос главной формы в папку и изменение стартовой формы

После того, как были созданы папки, можно перенести форму, созданную при генерации проекта, в нужную папку. Однако, просто перенести файл в папку почему-то не работает. Нужно перейти в файл и подписать название папки после namespace \*название проекта\*

namespace demo4.FORMS

{

public partial class LoginForm : Form

Теперь в program.cs нужно изменить параметр в Application.Run на необходимую форму:

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

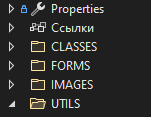
Application.Run(new FORMS.LoginForm());

}

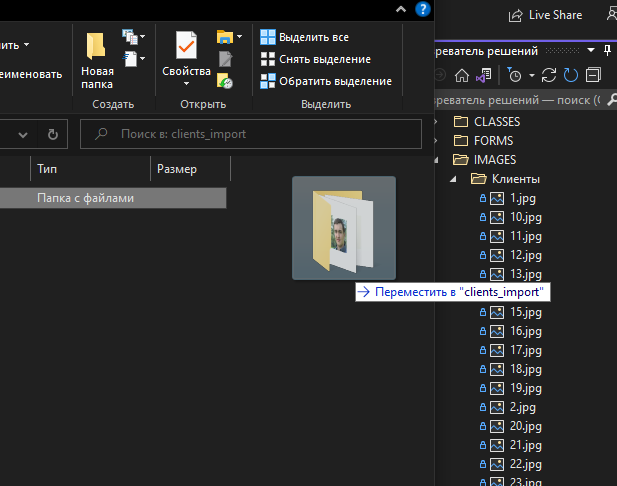
Визуалка будет ругаться, причем на очень много вещей сразу. Чинится простым запуском программы и перепросмотром файлов. Почему – вопрос хороший.

## Импорт изображений

Если в импорте были даны изображения, создайте еще одну папку для них:



Далее просто перетащите папку с импортом (или только изображения) в проект:

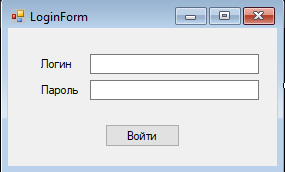


Теперь изображения будут сохранены в проекте.

## Создание формы авторизации

### Внешний вид:

Для начала нужно создать форму. Обычно на ней не много элементов, достаточно двух textbox`ов (с label к каждому из них) и кнопки.



### Функция для получения пользователя по его логину и паролю:

public static User GetUser(string login, string password)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "SELECT [ID], [Password], [FirstName], [MiddleName], [LastName], [Login], [IsDeleted] " +

"FROM [User] " +

"WHERE [Login] = @login " +

"AND [Password] = @password " +

"AND [IsDeleted] = 0";

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.AddWithValue("login", login);

cmd.Parameters.AddWithValue("password", password);

SqlDataReader reader = cmd.ExecuteReader();

if (reader.Read())

{

User user = new User

{

ID = reader.GetInt32(0),

Password = reader.GetString(1),

FirstName = reader.GetString(2),

MiddleName = reader.GetString(3),

LastName = reader.GetString(4),

Login = reader.GetString(5),

IsDeleted = reader.GetBoolean(6)

};

return user;

}

throw new Exception("Пользователь не найден. Проверьте правильность данных и повторите попытку входа.");

}

}

Сначала все соответствует шаблону. Используется подключение, оно же открывается. В запросе стоит **перечислить все получаемые поля**, это пригодится позже.

**@login** и **@password** – это параметры. Они гораздо безопаснее просто вставки строк. В остальном ничем не отличаются.

Поскольку это просто запрос, который не должен получать DataSet, создается **команда cmd**. После этого можно присвоить параметрам значения, используя **AddWithValue("имя параметра", значения)**. Учтите, что имя параметра должно писаться полностью идентично запросу, но **без символа @**. После присвоения параметрам значений создается reader.

Если этот **reader** читает что-нибудь, создается новая переменная типа User, заполняется через функции **Get\*название типа\*(ряд)** и возвращается. Если же ничего не прочиталось, кидается исключение, что такого пользователя нет.

### Код формы:

Весь код на данной форме находится в кнопке:

private void LoginBtn\_Click(object sender, EventArgs e)

{

User user;

try

{

user = DataWork.GetUser(LoginTxt.Text, PasswordTxt.Text);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

return;

}

this.Hide();

MainForm MF = new MainForm(user);

MF.ShowDialog();

LoginTxt.Text = String.Empty;

PasswordTxt.Text = String.Empty;

this.Show();

}

Код состоит, по сути, из **трех частей**, не считая объявления переменной **user**.

Первая часть пытается с помощью **try..catch** присвоить пользователю значение, используя написанную ранее функцию. Если у неё не получается, catch выводит сообщение полученной ошибки и завершает работу функции (**return**).

Вторая часть **прячет** данную форму, **создает** следующую и **показывает** **в режиме диалогового окна** (это важно). Заодно, при создании следующей формы, мы передаем ей полученного пользователя, чтобы она могла дальше с ним работать.

Третья часть срабатывает тогда, когда **закрывается** следующая форма, в данном случае – **MainForm**. Она **очищает** поля и снова **показывает** форму авторизации.

## Шифрование пароля

Для начала нужно подключить библиотеку: using System.Security.Cryptography;

Для шифрования используйте функцию:

public static string GetHash(string login, string password)

{

string salt1 = "^8{-";

string salt2 = "&>nm";

string pass = login + salt1 + password + salt2;

using (var hash = SHA256.Create())

{

return string.Concat(hash.ComputeHash(Encoding.UTF8.GetBytes(pass)).Select(x => x.ToString("X2")));

}

}

**Salt1** и **salt2** – это **соль** – неизменная часть, прибавляющаяся к паролю спереди и/или сзади. Логин передается как изменяемая часть – **перец**. Сама функция шифрует получившуюся строку по алгоритму **SHA256**.

Для того, чтобы проверять пароль с шифрованием, надо изменить параметр, подставляемый вместо пароля в запрос на пользователя:

cmd.Parameters.AddWithValue("password", GetHash(login, password));

Это, правда, может создать проблему того, что теперь в базе пароли тоже должны быть зашифрованы по тому же самому алгоритму. Учитывая, что интернетом для этого пользоваться не получится, можно, например, переписать все пароли такой функцией (объяснений не будет, потому что никто в здравом уме такое делать не будет):

public static void HashAllPasswords()

{

List<User> list = GetUserList();

foreach (var user in list)

{

HashPassword(user);

}

}

public static List<User> GetUserList()

{

List<User> list = new List<User>();

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "select ID, [Login], [Password] " +

"from[User] ";

SqlCommand cmd = new SqlCommand(sql, conn);

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())

{

User user = new User();

user.ID = reader.GetInt32(0);

user.Login = reader.GetString(1);

user.Password = reader.GetString(2);

list.Add(user);

}

return list;

}

}

public static void HashPassword(User user)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "update [User] set [Password] = @password where [User].ID = @id";

SqlCommand cmd = new SqlCommand(sql,conn);

cmd.Parameters.AddWithValue("password", GetHash(user.Login, user.Password));

cmd.Parameters.AddWithValue("id", user.ID);

cmd.ExecuteNonQuery();

}

}

Учитывайте, что её нужно прогнать только **ОДИН** раз.

Другой вариант: проверять, зашифрован ли пароль и шифровать его, если нет. Проблема этого метода в том, что, если действовать самым очевидным методом и просто проверять, совпадает ли напрямую пароль с тем, который уже в БД, если какой-нибудь умник попробует войти с тем шифром, который вы получили в итоге, он не только это сделает, но и перепишет пароль. Поскольку реализация этого займет только больше времени, а результат получается отрицательный, никакого кода для него не будет.

## Проверка роли пользователя

Для того, чтобы узнать роль пользователя, когда они разнесены по разным таблицам, можно воспользоваться таким запросом:

public static bool IsExec(User user)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "select \* from [Executor] where ID = @id";

SqlCommand cmd = new SqlCommand(sql,conn);

cmd.Parameters.AddWithValue("id", user.ID);

SqlDataReader reader = cmd.ExecuteReader();

return reader.HasRows();

}

Функция **bool**, т.е. логическая. Запросом мы выбираем все подходящие поля в другой таблице. В данном случае ID пользователей и ID исполнителей совпадали, поэтому запрос достаточно простой. Здесь используется **HasRows()** – тоже логическая функция, которая возвращает, есть ли ряды в полученных по запросу данных.

Используется эта функция максимально прямо:

if (IsExec(user))

{

\*делаете, что вам нужно\*

}

## Загрузка вариантов (например, для фильтрации) в ComboBox

### Функция:

public static List<string> GetExecList(int managerID)

{

List<string> list = new List<string>();

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "select [User].FirstName, [User].MiddleName, [User].LastName " +

"from[User],[Executor] " +

"where Executor.ID = [User].ID " +

"and Executor.ManagerID = @ManagerID";

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.AddWithValue("ManagerID", managerID);

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())

{

list.Add($"{reader.GetString(0)} {reader.GetString(1)} {reader.GetString(2)}");

}

return list;

}

}

Эта функция возвращает **список из строк**, в данном случае – ФИО исполнителей, подчиняющихся определенному менеджеру. Список создается **до чтения**, чтобы в него можно было добавлять ряды. **Пока (while)** reader читает, он добавляет каждый ряд в список через **list.Add(элемент)**. После того, как reader закончит читать, список возвращается функцией.

### Код:

Этот код должен прогоняться каждый раз, когда вам нужно загрузить варианты в combobox. Обычно это один раз, при создании формы, однако, если в какой-то момент варианты должны меняться, тогда при каждой смене.

ExecutorCmb.Items.Clear();

ExecutorCmb.Items.Add("");

foreach (string execName in DataWork.GetExecList(User.ID))

{

ExecutorCmb.Items.Add(execName);

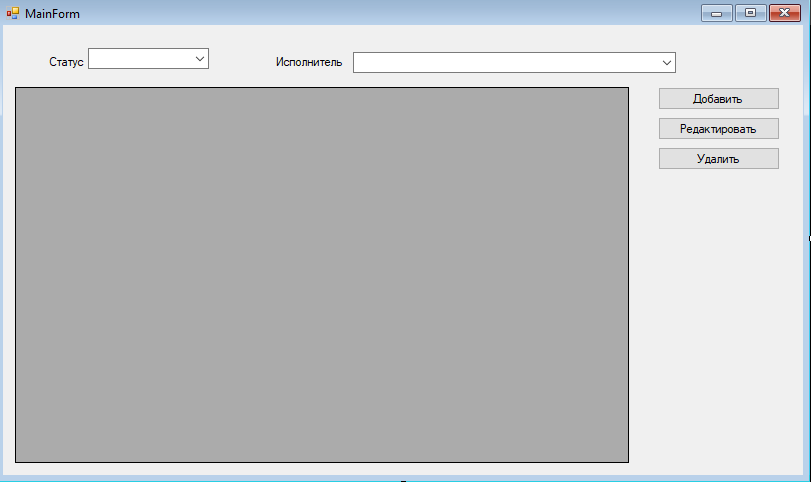
}

Сначала вычищаются все варианты, которые были до этого через **Clear()**. После этого добавляется один пустой вариант (если он нужен), он же может быть вариантом «все». Теперь используется цикл **foreach(\*элемент\* in \*коллекция\*)**, который срабатывает для каждого элемента в списке (список является коллекцией) и добавляет его в варианты combobox`а.

## Вывод данных в DataGrid

Сражу предупреждаю, здесь не особо учитывается сортировка, фильтрация, поиск, постраничный вывод и все прочее, поскольку это просто одна рандомно выбранная функция из нескольких демо. Для меня простым решением является то, что расписано на пару пунктов ниже, если вы не хотите с этим разбираться, я верю, что вы найдете свой способ.

Для начала добавьте элемент DataGrid на форму. Дайте ему нормальное название, это пригодится позже.



### Функция для получения данных:

public static DataSet GetExecTasks(User user, string statusFilter)

{

DataSet ds = new DataSet();

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = " select Task.Title, Task.[Status], [User].FirstName " +

"from[User], Executor, Task " +

"where Task.ExecutorID = Executor.ID " +

"and Executor.ID = [User].ID " +

"and Executor.ID = @id";

if (!String.IsNullOrEmpty(statusFilter))

sql += " and Task.Status = @status";

sql += " order by Task.CreateDateTime desc";

SqlDataAdapter ada = new SqlDataAdapter(sql, conn);

ada.SelectCommand.Parameters.AddWithValue("id", user.ID);

ada.SelectCommand.Parameters.AddWithValue("status", statusFilter);

ada.Fill(ds);

return ds;

}

}

Это единственный случай, когда используется **адаптер**, а не ридер, потому что нужно засунуть данные в **DataSet**. Здесь немного по-другому добавляются параметры, не просто Parameters.AddWithValue(), а **SelectCommand.Parameters.AddWithValue()**.

### Использование в формах:

Поскольку возвращается **DataSet**, можно для отрисовки делать так:

private void Render()

{

TasksDataGrid.DataSource = DataWork.GetExecTasks(this.User, this.statusFilter).Tables[0];

}

Теперь при каждом изменении параметров нужно вызывать эту функцию **Render()**.

## Вывод данных через UserControl

### Создание UserControl:

Если в ТЗ дан какой-то особенный вид, которым должны выводиться данные, стоит воспользоваться **UserControl**. Поскольку всё, что находится на формах, является **Control**`ами, UserControl – это буквально **пользовательский элемент**. Создайте шаблон расположения всех элементов:



Учтите, что все **лэйблы должны иметь имена**, чтобы не путаться в них. Заполняться шаблон будет **при создании** элемента, поэтому ему надо передавать нужные данные, например, используя экземпляр одного из созданных во время работы **классов**:

public ClientService(CLASSES.ClientService service)

{

InitializeComponent();

lblTitle.Text = service.Title;

lblTime.Text = service.StartTime.ToString();

lblFileCount.Text = $"Всего файлов: {service.DocumentCount}";

}

### Функция:

public static List<ClientService> GetClientServiceList(int ClientID)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = " select [Service].Title, ClientService.StartTime, " +

"(select COUNT(\*) from[DocumentByService] where ClientServiceID = @ID) as [count] " +

"from[Service],ClientService " +

"where[Service].ID = ClientService.ServiceID " +

"and ClientService.ClientID = @ID";

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.AddWithValue("ID", ClientID);

SqlDataReader reader = cmd.ExecuteReader();

List<ClientService> list = new List<ClientService>();

while (reader.Read())

{

ClientService clientService = new ClientService

{

Title = reader.GetString(0),

StartTime = reader.GetDateTime(1),

DocumentCount = reader.GetInt32(2),

};

list.Add(clientService);

}

return list;

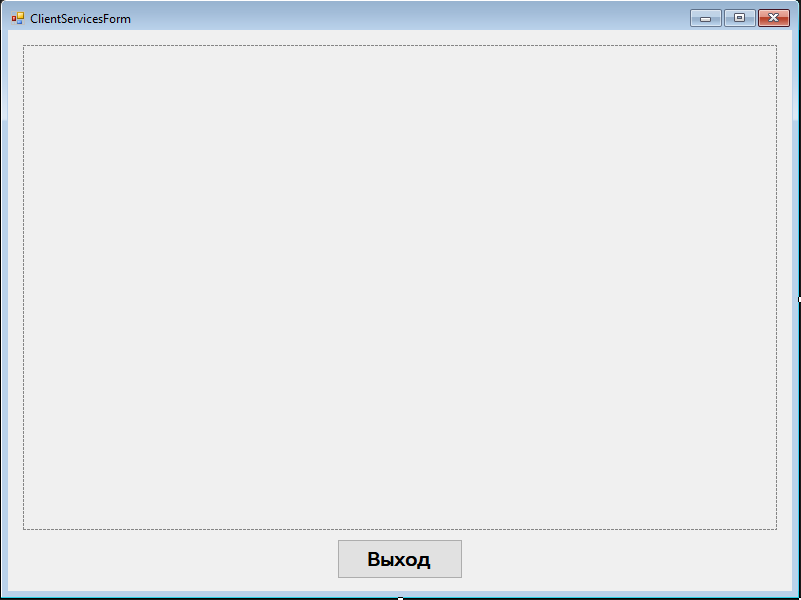
}

}

В целом функция работает так же, как и та, которая создает **список вариантов на фильтрацию**, но создает **список из экземпляров класса**.

### Форма для отображения:

На форму размещается элемент **Panel**, (в данном случае **FlowLayoutPanel**). Panel работает как **контейнер**, в который будут размещены **UserControl**.



Чтобы заполнять панель, нужен подобный код:

try

{

layoutServices.Controls.Clear();

foreach (var service in DataWork.GetClientServiceList(clientID))

{

ClientService item = new ClientService(service);

layoutServices.Controls.Add(item);

}

layoutServices.Refresh();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

**Controls.Clear()** очищает панель. Это необходимо для того, чтобы, при необходимости, к выводу применялись сортировка и т.д.

**Foreach** перебирает каждый элемент из полученного списка. Внутри на основе каждого элемента создается **UserControl** и добавляется в панель через **Controls.Add(item).**

## Универсальный запрос на получение данных (учитывает фильтрацию, сортировку, поиск и страницы)

Окей, это будет долгий пункт. Вместо того, чтобы писать отдельные запросы и/или функции, можно сделать одну функцию с запросом-«конструктором», который будет учитывать все переменные. Начинается все с того же запроса, что и для просто вывода. Не обращайте внимание на его размер, это просто большой запрос. Начнем с базы:

public static DataSet GetClientEntries()

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

DataSet ds = new DataSet();

conn.Open();

string sql = "select Client.ID as Идентификатор, Client.Gender as Пол, Client.FirstName as Фамилия, Client.LastName as Имя, Client.Patronymic as Отчество, " +

" Client.Birthday as [Дата рождения], Client.Phone as Телефон, Client.Email as Email, Client.RegistrationDate as [Дата регистрации], " +

" (select top 1 ClientService.StartTime from ClientService where ClientService.ClientID = Client.ID order by StartTime desc) as [Последнее посещение]," +

" (select COUNT(\*) from ClientService where ClientService.ClientID = Client.ID) as [Количество посещений]" +

" from Client, ClientService";

sql += " group by Client.ID, Client.FirstName, Client.LastName, Client.Patronymic, Client.Birthday, Client.Email, Client.Gender, Client.Phone, Client.PhotoPath, Client.RegistrationDate";

SqlDataAdapter ada = new SqlDataAdapter(sql, conn);

ada.Fill(ds);

return ds;

}

}

**Group by** нужен исключительно для работы этого огромного запроса, но обратите внимание, что он добавляется отдельно.

### Фильтрация:

Теперь надо учитывать **фильтрацию** (в данном случае – по полу). Для этого добавляется дополнительный параметр **filter** и небольшая проверка:

public static DataSet GetClientEntries(string filter)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

DataSet ds = new DataSet();

conn.Open();

string sql = "\*для сокращения далее здесь будет просто слово «запрос»";

if (!string.IsNullOrEmpty(filter))

{

sql += $" WHERE Client.Gender = '{filter}'";

}

sql += " group by …";

SqlDataAdapter ada = new SqlDataAdapter(sql, conn);

ada.Fill(ds);

return ds;

}

}

**string.IsNullOrEmpty()** возвращает true, если переданная ей строка пустая или null.

Если строка filter пришла не пустая, тогда к существующему запросу прибавляется условие фильтрации. **БУДЬТЕ ОЧЕНЬ ВНИМАТЕЛЬНЫ С ПРОБЕЛАМИ.** Всегда ставьте их либо в конце, либо в начале строки. Здесь же и пригодилось то, что мы вынесли group by в отдельную часть, поскольку group by ставится после всех условий.

### Поиск:

Следующий делается поиск, однако для него задается не только сама строка, которая ищется, но и столбец, в котором производится поиск:

public static DataSet GetClientEntries(string filter, string searchString, string searchCategory)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

DataSet ds = new DataSet();

conn.Open();

string sql = "запрос";

if (!string.IsNullOrEmpty(filter))

{

sql += $" WHERE Client.Gender = '{filter}'";

}

switch (searchCategory)

{

case "ФИО":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" CONCAT(Client.LastName,' ', Client.FirstName,' ', Client.Patronymic) like '%{searchString}%'";

break;

}

case "Email":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" Email like '%{searchString}%'";

break;

}

case "Телефон":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" Phone like '%{searchString}%'";

break;

}

}

sql += " group by …";

SqlDataAdapter ada = new SqlDataAdapter(sql, conn);

ada.Fill(ds);

return ds;

}

}

Задание позволяло использовать конструкцию switch..case, потому что там были сказаны столбцы, в которых нужно будет искать. Более универсально это можно реализовать, если вместо всего switch..case сделать sql += $" {searchCategory} like '%{searchString}%'", но тогда надо следить, чтобы варианты в combobox совпадали с названиями столбцов.

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND"; - это **тернарный оператор**. В общем виде он выглядит как **a ? b : c**. По сути – это очень сокращенный if..else. Если условие а истинно, выбирается значение b, иначе значение c.

Здесь такой тернарный оператор нужен для того, чтобы запрос правильно составлялся, т.е. после where не шел where, и не появлялся and без where. По сути, вы проверяете, было ли что-то написано до этой части: если **да** – ставите **and**, если **нет** – ставите **where**. Заметьте, что **в switch нет default**, потому что нам не нужно выполнять никаких действий в любых других случаях.

Небольшое объяснение по запросу: **CONCAT()** соединяет строки, поэтому мы его используем здесь, чтобы получить ФИО. **Like** – это оператор сравнения, но более мягкий. Обратите внимание на **%** вокруг searchString – в SQL они обозначают **любое количество любых символов**.

### Сортировка:

Теперь надо добавить сортировку. Её тоже можно сделать через switch..case:

public static DataSet GetClientEntries(string filter, string sort, string searchString, string searchCategory)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

DataSet ds = new DataSet();

conn.Open();

string sql = "запрос";

if (!string.IsNullOrEmpty(filter))

{

sql += $" WHERE Client.Gender = '{filter}'";

}

switch (searchCategory)

{

case "ФИО":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" CONCAT(Client.LastName,' ', Client.FirstName,' ', Client.Patronymic) like '%{searchString}%'";

break;

}

case "Email":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" Email like '%{searchString}%'";

break;

}

case "Телефон":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" Phone like '%{searchString}%'";

break;

}

}

sql += " group by …";

switch (sort)

{

case "Фамилии":

{

sql += " ORDER BY Client.LastName";

break;

}

case "Дате последнего посещения":

{

sql += " ORDER BY [Последнее посещение] desc";

break;

}

case "Количеству посещений":

{

sql += " ORDER BY [Количество посещений] desc";

break;

}

default:

{

sql += " ORDER BY Client.ID";

break;

}

}

SqlDataAdapter ada = new SqlDataAdapter(sql, conn);

ada.Fill(ds);

return ds;

}

}

Сортировка делается путем добавления к запросу **order by**, который ставится **после group by**. В большинстве случаев столбцы, по которым нужно сортировать, заранее даны в задании, поэтому её достаточно просто делать через switch..case. Если же нужно учитывать ещё и сортировку по возрастанию/убыванию (что не нужно было в том демо, из которого берется пример), можно после самого order by добавлять с помощью тернарного оператора **asc**(по возрастанию) или **desc**(по убыванию):

sql += sortUp ? " ASC" : " DESC";

### Постраничный вывод:

Теперь осталось сделать постраничный вывод (запрос приведен полностью):

public static DataSet GetClientEntries(int page, int pageSize, string filter, string sort, string searchString, string searchCategory)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

DataSet ds = new DataSet();

conn.Open();

string sql = "select Client.ID as Идентификатор, Client.Gender as Пол, Client.FirstName as Фамилия, Client.LastName as Имя, Client.Patronymic as Отчество, " +

" Client.Birthday as [Дата рождения], Client.Phone as Телефон, Client.Email as Email, Client.RegistrationDate as [Дата регистрации], " +

" (select top 1 ClientService.StartTime from ClientService where ClientService.ClientID = Client.ID order by StartTime desc) as [Последнее посещение]," +

" (select COUNT(\*) from ClientService where ClientService.ClientID = Client.ID) as [Количество посещений]" +

" from Client, ClientService";

if (!string.IsNullOrEmpty(filter))

{

sql += $" WHERE Client.Gender = '{filter}'";

}

switch (searchCategory)

{

case "ФИО":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" CONCAT(Client.LastName,' ', Client.FirstName,' ', Client.Patronymic) like '%{searchString}%'";

break;

}

case "Email":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" Email like '%{searchString}%'";

break;

}

case "Телефон":

{

sql += string.IsNullOrEmpty(filter) ? " WHERE" : " AND";

sql += $" Phone like '%{searchString}%'";

break;

}

}

sql += " group by Client.ID, Client.FirstName, Client.LastName, Client.Patronymic, Client.Birthday, Client.Email, Client.Gender, Client.Phone, Client.PhotoPath, Client.RegistrationDate";

switch (sort)

{

case "Фамилии":

{

sql += " ORDER BY Client.LastName";

break;

}

case "Дате последнего посещения":

{

sql += " ORDER BY [Последнее посещение] desc";

break;

}

case "Количеству посещений":

{

sql += " ORDER BY [Количество посещений] desc";

break;

}

default:

{

sql += " ORDER BY Client.ID";

break;

}

}

if (pageSize > 0)

{

sql += $" offset {(page - 1) \* pageSize} rows fetch next {pageSize} rows only";

}

SqlDataAdapter ada = new SqlDataAdapter(sql, conn);

ada.Fill(ds);

return ds;

}

}

Поскольку в программе можно было изменять количество записей на странице, постраничный вывод применяется только тогда, когда размер страницы больше 0 (для варианта «Все записи» переменной pagesize задается значение -1).

Такой же запрос можно использовать и для работы с UserControl, если изменить конечную часть и возвращаемое значение на **List<\*нужный тип\*>**.

### Код формы:

Во-первых, на форме надо создать **глобальные переменные** для каждой из переменных функций. В нашем случае это будут:

private int \_page = 1;

private int \_pageSize = 10;

private string \_filter = "";

private string \_sort = "";

private string \_searchString = "";

private string \_searchCategory = "";

В функции **Render()** источник для DataGrid теперь будет выглядеть так:

dgClients.DataSource = DataWork.GetClientEntries(\_page, \_pageSize, \_filter, \_sort, \_searchString, \_searchCategory).Tables[0];

Во всех остальных функциях мы присваиваем значения глобальным переменным и вызываем Render(). Примеры:

### TextBox:

private void txtSearch\_TextChanged(object sender, EventArgs e)

{

\_searchString = txtSearch.Text;

Render();

}

### ComboBox:

private void cmbSearchCategory\_SelectedIndexChanged(object sender, EventArgs e)

{

\_searchCategory = cmbSearchCategory.SelectedItem.ToString();

Render();

}

### RadioButton:

private void rbFemale\_CheckedChanged(object sender, EventArgs e)

{

\_filter = "ж";

Render();

}

### Кнопки пролистывания страниц:

private void btnPrev\_Click(object sender, EventArgs e)

{

if ((\_page - 2) \* \_pageSize + 1 <= 0)

return;

\_page--;

Render();

}

private void btnNext\_Click(object sender, EventArgs e)

{

if ((\_page + 1) \* \_pageSize > \_entryNumber)

return;

\_page++;

Render();

}

Значения получены, по большей части, экспериментальным путем. Суть в том, чтобы не увеличивать/уменьшать номер страницы за пределы их количества.

## Как выбирать строку из DataGrid

Во-первых, найдите свойства **MultiSelect** и **SelectionMode**, проставьте их **False** и **FullRowSelect** соответственно. Теперь будет выбираться ряд **целиком**, причем всегда **один**.

Теперь для свойства **CellContentClick** можно задать подобную функцию.

private void dgClients\_CellContentClick(object sender, DataGridViewCellEventArgs e)

{

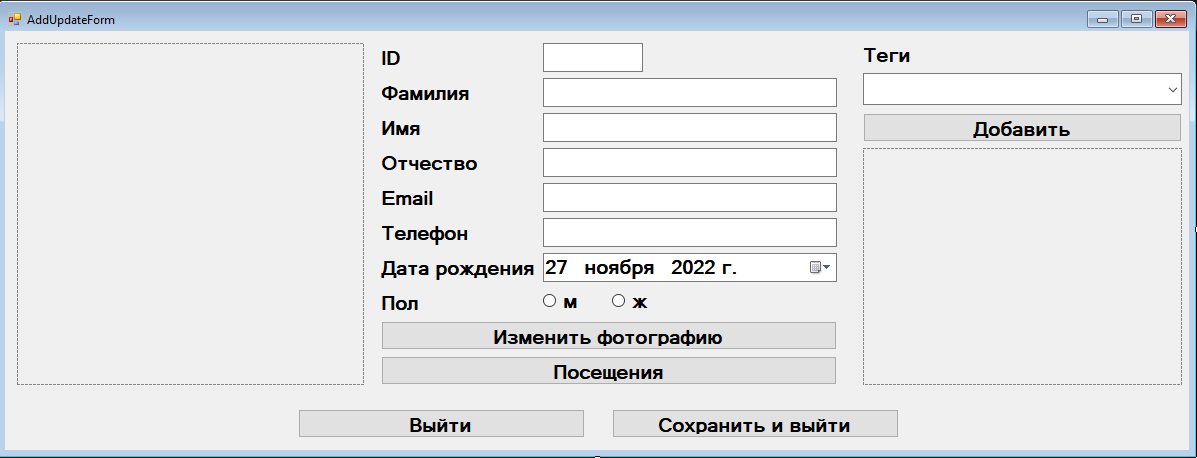
\_selectedClientID = int.Parse(dgClients.SelectedRows[0].Cells[0].Value.ToString());

}

**\_selectedClientID** – глобальная переменная, в которой хранится **текущий** выбранный клиент. Поскольку это **число**, используется **int.Parse()**, в который передается **строковое** значение **первой клетки первого** (в данном случае единственного) **выбранного ряда в датагриде**. Выглядит очень непонятно, но IDE сама заставит написать что-то подобное, так что она вам в помощь.

## Создание формы добавления/редактирования

В большинстве случав они будут выглядеть **идентично**, но иметь немножко разный функционал. Чтобы не засорять проект лишними формами, добавление и редактирование можно делать **на одной**. Постарайтесь дать как можно более общие надписи кнопкам, если это возможно.



Функция для добавления данных о клиенте:

public static void AddClient(Client client)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "insert into Client(LastName, FirstName, Patronymic, Birthday, Phone, Email, RegistrationDate, Gender, PhotoPath)" +

" values(@LastName, @FirstName, @Patronymic, @Birthday, @Phone, @Email, @RegistrationDate, @Gender, @PhotoPath)";

SqlCommand cmd = new SqlCommand(sql,conn);

cmd.Parameters.AddWithValue("LastName", client.LastName);

cmd.Parameters.AddWithValue("FirstName", client.FirstName);

cmd.Parameters.AddWithValue("Patronymic", client.Patronymic);

cmd.Parameters.AddWithValue("Birthday", client.Birthdate);

cmd.Parameters.AddWithValue("Phone", client.Phone);

cmd.Parameters.AddWithValue("Email", client.Email);

cmd.Parameters.AddWithValue("RegistrationDate", client.RegistrationDate);

cmd.Parameters.AddWithValue("Gender", client.Gender);

cmd.Parameters.AddWithValue("PhotoPath", client.PhotoPath);

try

{

cmd.ExecuteNonQuery();

}

catch (Exception e)

{

throw e;

}

}

}

Функция для обновления данных клиента:

public static void UpdateClient(Client client)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "update Client " +

"set LastName = @LastName, FirstName = @FirstName, Patronymic = @Patronymic, " +

" Birthday = @Birthday, Phone = @Phone, Email = @Email, RegistrationDate = @RegistrationDate, Gender = @Gender, PhotoPath = @PhotoPath " +

"where ID = @ID";

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.AddWithValue("ID", client.ID);

cmd.Parameters.AddWithValue("LastName", client.LastName);

cmd.Parameters.AddWithValue("FirstName", client.FirstName);

cmd.Parameters.AddWithValue("Patronymic", client.Patronymic);

cmd.Parameters.AddWithValue("Birthday", client.Birthdate);

cmd.Parameters.AddWithValue("Phone", client.Phone);

cmd.Parameters.AddWithValue("Email", client.Email);

cmd.Parameters.AddWithValue("RegistrationDate", client.RegistrationDate);

cmd.Parameters.AddWithValue("Gender", client.Gender);

cmd.Parameters.AddWithValue("PhotoPath", client.PhotoPath);

try

{

cmd.ExecuteNonQuery();

}

catch (Exception e)

{

throw e;

}

}

}

Да, функции тоже похожи, но у них используется разный запрос. В первом случае **insert into**, который добавляет строку в таблицу, во втором **update**, который изменяет нужную строку/строки.

В обоих случаях создается команда, которая потом исполняется вне очереди: **ExecuteNonQuery()**.

### Код формы:

В коде придется достаточно много писать. Для начала нужно создать глобальную переменную **mode**:

private int mode;

Теперь изменим **конструктор** формы: ей нужно **принимать** на вход либо **клиента**, либо **ID клиента**, либо числовое значение, которое и станет **режимом** работы, тут на ваше усмотрение. В моем случае передается ID:

public AddUpdateForm(int clientID)

{

InitializeComponent();

if (clientID == 0)

{

\_client = new Client();

label1.Visible = false;

txtID.Visible = false;

mode = 0;

}

else

{

try

{

\_client = DataWork.GetClientByID(clientID);

txtID.Text = \_client.ID.ToString();

txtLastName.Text = \_client.LastName;

txtFirstName.Text = \_client.FirstName;

txtPatronymic.Text = \_client.Patronymic;

dtBirthdate.Value = \_client.Birthdate;

txtPhone.Text = \_client.Phone;

txtEmail.Text = \_client.Email;

photoPath = \_client.PhotoPath;

if (\_client.Gender == "м")

{

rbMale.Checked = true;

}

else

{

rbFemale.Checked = true;

}

ImgUpload(ImageFolderPath + Path.GetFileName(\_client.PhotoPath));

mode = 1;

}

catch (Exception e)

{

throw e;

}

}

}

Если ID **не пришел**, форма открывается в режиме **добавления.** По условию нужно скрыть строку, в которой показывается ID, поэтому **скрываются** соответствующие **Textbox и Label**. **Mode = 0**.

Если ID **пришел**, то поля формы **заполняются** данными нужного клиента, а **mode** становится равно **1** (редактирование).

Далее различия встретятся только при нажатии кнопки **«Сохранить и выйти»:**

private void btnSaveAndExit\_Click(object sender, EventArgs e)

{

switch (mode)

{

case 0:

{

try

{

ReadData();

DataWork.AddClient(\_client);

this.Close();

}

catch (Exception exception)

{

MessageBox.Show(exception.ToString());

}

break;

}

case 1:

{

try

{

ReadData();

DataWork.UpdateClient(\_client);

this.Close();

}

catch (Exception exception)

{

MessageBox.Show(exception.ToString());

}

break;

}

}

}

Здесь, в зависимости **от режима** работы, запускается **AddClient** или **UpdateClient**.

**ReadData** – функция, которая читает все данные в полях и записывает в глобальную переменную клиента.

## Генерация рандомного ID

Предположим, ID – это число в диапазоне от 1 до 10000. Я гарантирую, что есть способы лучше, например делать все напрямую в T-SQL, но это самый простой для понимания и реализации способ.

Для начала, ID в базе данных должен быть не identity, чтобы ему можно было присваивать значения.

Нам нужно обеспечить, чтобы вставляемые значения были уникальными. Для этого понадобится функция, которая будет проверять, уникальный ли ID ей передали:

public bool IsUnique(int id)

{

using (var sqlConn = new SqlConnection(ConnectionString))

{

var sql = $"select \* from Users where Users.ID = '{id}'";

var sqlComm = new SqlCommand(sql, sqlConn);

var rd = sqlComm.ExecuteReader();

return rd.HasRows;

}

}

Эта функция работает также, как и проверка роли.

Теперь нужно изменить функцию AddUser, чтобы в ней создавать ID:

public void AddUser()

{

…

Random rnd = new Random();

int id = rnd.Next(10000);

while (!IsUnique(id))

{

id = rnd.Next(10000);

}

…

}

Здесь ID генерируется до тех пор, пока не будет уникальным, а далее все продолжается как обычно.

## Удаление данных

### Функция:

public static void DeleteClient(int ID)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "delete from Client where ID = @ID";

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.AddWithValue("ID", ID);

try

{

cmd.ExecuteNonQuery();

}

catch (Exception e)

{

throw e;

}

}

}

Как и в добавлении/редактировании, здесь используется **ExecuteNonQuery()**.

Если есть какие-то **условия**, по которым **нельзя удалять** данные, можно добавить дополнительную функцию:

public static bool CanDeleteClient(int ID)

{

using (SqlConnection conn = new SqlConnection(ConnStr))

{

conn.Open();

string sql = "select \* from ClientService where ClientID = @ID";

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.AddWithValue("ID", ID);

SqlDataReader reader = cmd.ExecuteReader();

bool canDelete = !reader.Read();

return canDelete;

}

}

Функция **логическая**, принцип тот же, как и у любой другой логической функции в этом документе, но возвращается **противоположное значение**, поскольку, по условию, нельзя было удалять пользователей, у которых были записи о оказанных им сервисах.

### Код в форме:

private void btnDelete\_Click(object sender, EventArgs e)

{

if (\_selectedClientID == 0)

{

MessageBox.Show("Выберите клиента");

return;

}

DialogResult dialogResult = MessageBox.Show("Are you sure?", "Delete client data", MessageBoxButtons.YesNo);

if (dialogResult == DialogResult.Yes)

{

if (!DataWork.CanDeleteClient(\_selectedClientID))

{

MessageBox.Show("Невозможно удалить клиента");

return;

}

DataWork.DeleteClient(\_selectedClientID);

}

}

Если **не выбран** ни один клиент, функция показывает сообщение и завершается.

Если клиент выбран, появляется окно **MessageBox**, которое спрашивает, действительно ли пользователь хочет удалить клиента. Результат записывается в **DialogResult**.

Если результат **«Да»**, функция проверяет, **можно ли удалить** этого клиента. Если **нет** – показывает сообщение и выходит.

Если клиента можно удалять, функция это делает и завершается.

## Динамическая работа с изображениями

На самом деле, можно использовать жесткие пути к картинкам, особенно, учитывая, что их теперь проще получить, когда они лежат в проекте. Однако, изображения в проекте позволяют и полностью забыть о статических адресах, делая проект более подвижным.

Здесь приведены примеры функций для формы, на которой и происходит работа с изображениями, однако, если таких много (и их нельзя сделать одной формой с несколькими режимами), при желании, их можно переписать как более общие функции. Для начала, нам понадобится подключить библиотеку **System.IO**, которая позволяет работать с системой:

using System.IO;

Теперь нам понадобятся две переменные: одна будет хранить путь до папки с изображениями, вторая будет путем до текущего изображения:

private string ImageFolderPath =

Path.GetFullPath(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, @"..\..\IMAGES\Клиенты\"));

private string photoPath;

**AppDomain.CurrentDomain.BaseDirectory** – путь до папки **Debug**, в которой лежит файл .exe. Поскольку изображения были загружены в другое место, нам нужно перейти к ним. Для этого мы пользуемся функцией **Path.Combine()**, и добавляем путь до изображений. Если папка с изображениями находится в Debug, нужно добавить только название папки. Наш случай сложнее: папка находится выше на две директории. Поэтому нужно использовать что-то похожее на **@"..\..\IMAGES\\*название папки с изображениями(если она есть внутри)\*\"**. “..\” – это как раз способ подняться на одну папку выше. **Path.GetFullPath(**) – получает полный путь до того, что указано в скобках, начиная с диска. Таким образом, мы всегда получаем путь до папки с изображениями, где бы не находился проект.

Во время загрузки не забудьте присвоить photopath значение из БД.

Функция для отображения изображений: (Можно обойтись и без неё, но ее наличие немного повышает читаемость кода, к тому же, отображение происходит в двух местах: при загрузке формы и при выборе нового изображения)

private void ImgUpload(string path)

{

if (!String.IsNullOrEmpty(path))

{

Image image = Image.FromFile(path);

imgClient.Image = image;

}

}

Проверка на пустоту строки нужна, чтобы не появлялась ошибка во время **Image.FromFile()**. Если по заданию нужно загрузить плейсхолдер вместо отсутствующего изображения, это можно сделать через **else** или **try..catch**. Для отображения изображения при загрузке формы эта функция используется так:

ImgUpload(ImageFolderPath + Path.GetFileName(\_client.PhotoPath));

**Path.GetFileName()** – обрезает путь до файла до только его имени с расширением. Используется на всякий случай, если в БД кроме имени файла дана ещё и папка.

Как открывать проводник для поиска изображения:  
 private void btnChangeImg\_Click(object sender, EventArgs e)

{

using (var openPhotoDialog = new OpenFileDialog())

{

if (openPhotoDialog.ShowDialog() == DialogResult.OK)

{

ImgUpload(openPhotoDialog.FileName);

photoPath = openPhotoDialog.FileName;

}

}

}

Эта функция находится на событии OnClick кнопки «Выбрать изображение». OpenFileDialog() открывает проводник. Если во время выбора файла выбирается файл, он отображается и путь до него сохраняется в photopath.

Функция для сохранения изображения в папку. Используется только если нужно сохранить изменения, например, в функции ReadData из примера формы добавления/редактирования. Обратите внимание, что эта же функция сохраняет photopath в базе через клиента.

private void ImgSave()

{

string name = Path.GetFileName(photoPath);

if (name != \_client.PhotoPath && photoPath != \_client.PhotoPath)

{

\_client.PhotoPath = name;

File.Copy(photoPath, ImageFolderPath + name, true);

}

}

Name берется для того, чтобы сохранить исключительно имя файла в БД. Можно проверять исключительно соответствие name и того, что лежит в БД, но, не забываем о возможности того, что там может храниться и название папки, поэтому проверяется ещё и полностью photopath, который не изменится, если не загружалось новое изображение. Если ни то, ни другое не совпало с файлом в БД, значение переписывается и файл копируется в папку при помощи **File.Copy(\*откуда копировать\*, \*куда копировать\*, \*переписывать ли, если будет совпадение\*)**. Таким образом, в изображениях проекта могут быть не только импортированные, но и любые новые, которые были добавлены в ходе работы программы. Будьте осторожны с файлами, которые загружаете для теста, не повторяйте моих ошибок.

## Обращение к реальному времени (доброе утро/день/вечер из второго пробника)

Здесь оформлено в отдельную функцию, но это не обязательно.

public static string GetTimed(DateTime dateTime)

{

if ((dateTime.Hour >= 9) && (dateTime.Hour < 11))

{

return "Доброе утро!";

}

else if ((dateTime.Hour >= 11) && (dateTime.Hour < 18))

{

return "Добрый день!";

}

else

{

return "Добрый вечер!";

}

}

У типа **DateTime** есть свойство **Hour**, которое является числом. Есть и другие свойства для **минут, секунд** и **миллисекунд**.

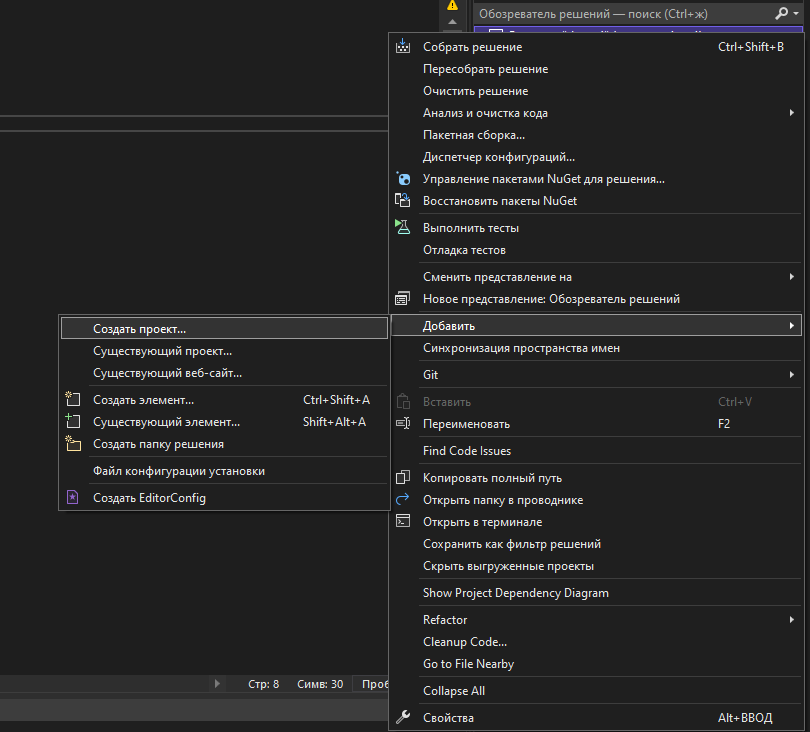
# Unit-тесты

## Создание проекта с тестами:

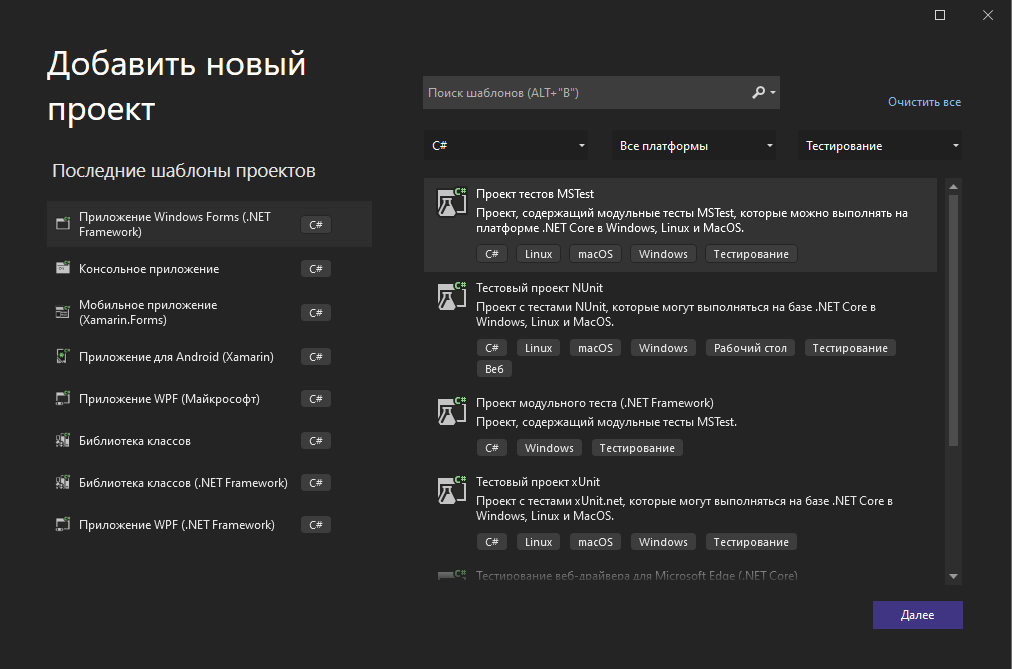
Да, всё правильно, проекта. Поскольку тесты функционируют не совсем так, как остальная программа, им нужен свой проект, но он все еще должен быть в решении.

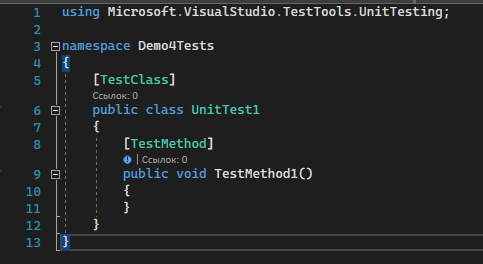
### Метод 1 (более правильный):

Для начала надо нажать ПКМ по решению, найти там «Добавить» -> «Создать проект…»



Теперь появится обычное меню создания проекта. Чтобы долго не искать, выберите в правом списке «тестирование». Выбирайте Core или Framework версию, в зависимости от проекта. Далее пойдут обычные шаги создания, главное, дайте адекватное имя.

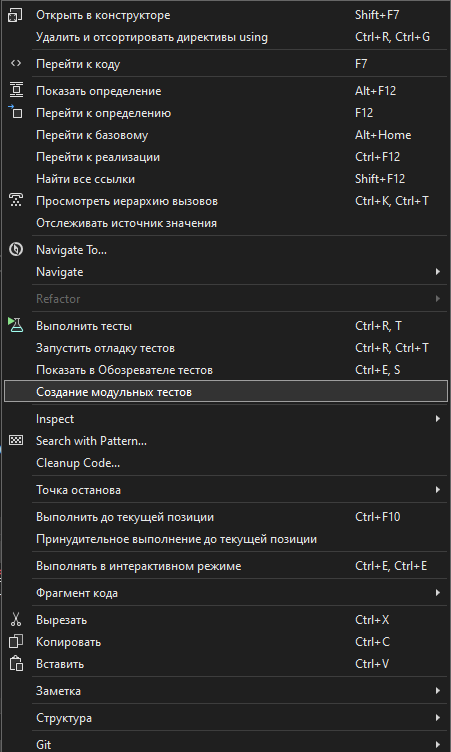




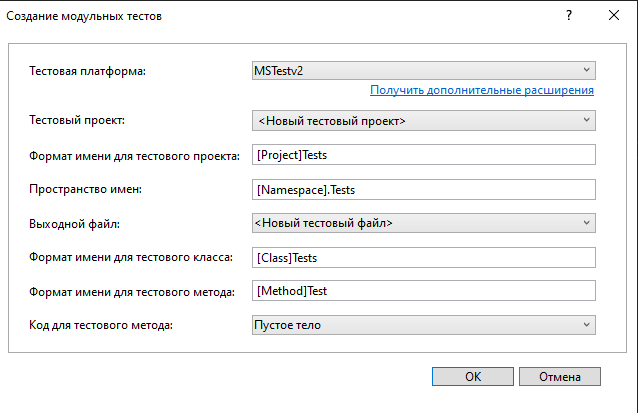
И ура, вот ваш тест. Добавлять новые можно в созданном тест-проекте. Однако остается одна проблема: он не связан с остальным проектом. Исправить это можно, добавив ссылку на проект.

### Метод 2 (более быстрый):

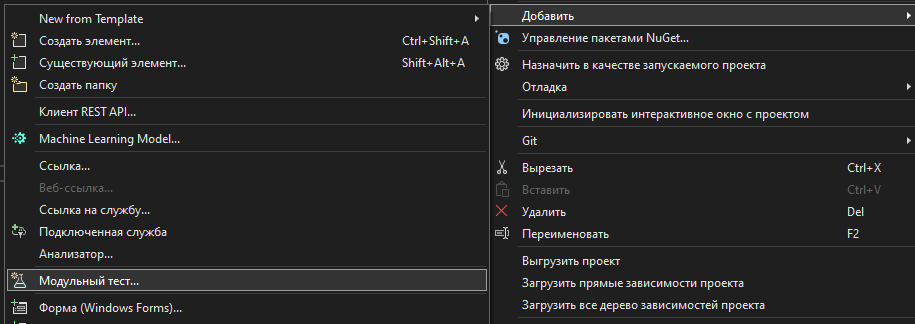
Если вас, как и любого другого нормального человека на демо, интересует скорость, то есть способ проще. Зайдите на код любой формы, нажмите ПКМ, найдите «Создание модульных тестов»



В следующем окне будет много страшного и непонятного, но суть в том, что это можно игнорировать. Просто создайте проект с тестами.



Создали проект? Отлично. Добавьте новый файл и снесите тот, что сгенерировался. Готово, у вас есть тест-проект.



## Примеры тестов:

Я так и не знаю, нужно им **5 тестов** или 5 методов (скорее всего первое). В тестах **основная часть** – это **Assert**. Именно он говорит, **правильно** ли прошел тест (что не всегда означает выполнение функции). **Assert** имеет много функций, но в основном понадобятся следующие:

### Ожидается определенное значение(Assert.AreEqual(expected, result);)

Тут немножко глупо и не очень очевидно, но, к сожалению, проверить целиком весь экземпляр класса **нельзя**. Сравнивать можно только относительно **базовые вещи: числа, строки** и т.п. Поэтому если нужно проверить, читает ли нужную строку функция, можно либо довериться какому-нибудь **одному значению**, либо **проверять все**. К счастью, для второго варианта можно просто **подписывать** ещё **assert**`ы, не создавая новые методы. Но когда время утекает, думаю, будет не до этого.

[TestMethod]

public void Test\_ValidLogin\_ValidPassword()

{

string login = "velizhanina";

string password = "123";

Assert.AreEqual(3, DataWork.GetUser(login, password).ID);

}

### Ожидается экземпляр определенного типа(Assert.IsInstanceOfType(result, type);)

Если проверять каждое поле или даже отдельные поля нет смысла, можно проверить, относится ли полученный результат к **определенному типу**:

[TestMethod]

public void GetExecList\_TypeTest()

{

User user = new User();

user.ID = 1;

Assert.IsInstanceOfType(DataWork.GetExecTasks(user, ""), typeof(DataSet));

}

Здесь обязательно использовать **typeof**, потому что иначе он просто отказывается работать.

### Ожидается не null(Assert.IsNotNull(result);)

Если вообще не имеет смысл проверять, что пришло, а важно проверить, вернулось ли **хоть что-нибудь**, можно проверять, пришло ли значение **null**:

[TestMethod]

public void GetExecList\_TypeTest()

{

User user = new User();

user.ID = 1;

Assert.IsNotNull(DataWork.GetExecTasks(user, ""));

}

Если правильным результатом **является null**, есть **Assert.IsNull()**. Работает абсолютно так же.

### Ожидается true(Assert.IsTrue(result);) или ожидается false(Assert.IsFalse(result);)

Эти тесты нужны для логических функций, например, проверки роли:

[TestMethod]

public void IsManager\_True()

{

User user = new User();

user.ID = 12;

Assert.IsTrue(DataWork.IsManager(user));

}

[TestMethod]

public void IsManager\_False()

{

User user = new User();

user.ID = 1;

Assert.IsFalse(DataWork.IsManager(user));

}

### Ожидание ошибки (Assert.ThrowsException<ExceptionType>(function);)

Вот тот самый случай, когда метод не обязан сработать. Вернее, **обязан не сработать**. Например, используя примеры из данного документа, если передать функции логина неправильные данные, она возвращает исключение. Тест на такой случай может выглядеть так:

[TestMethod]

public void Test\_UnvalidLogin\_UnvalidPassword()

{

string login = "dfwth";

string password = "15fd";

Assert.ThrowsException<Exception>(() => DataWork.GetUser(login, password));

}

В этом случае **assert** требует **лямбда-выражение (=>)**, но просто примите его как данность. Если вы задаете какое-то уникальное исключение, вместо **<Exception>** напишите в скобках **это уникальное исключение**. Например, в функции, делящей одно число на другое, вы можете ожидать **DivideByZeroException**.

# Загрузка на гит с помощью команд

Первое, что нужно сделать – зайти **с консоли в папку** с проектом. Это можно сделать либо открыв консоль в папке (написать cmd в строке с путем), либо открыть консоль и перейти в нужную папку с помощью cd.

Когда вы будете в папке решения, создайте репозиторий: **git init**

Чтобы **добавить** файл в коммит, нужно написать **git add ИМЯ\_ФАЙЛА**. Можно таким же образом добавлять и папки: **git add КАТАЛОГ**.

**Проверить**, что сейчас лежит в коммите, можно с помощью **git status**.

Если передумали добавлять файл в коммит, его можно **удалить: git reset HEAD <file>**

Чтобы оформить изменения в **коммит**, нужно воспользоваться **git commit -m “Осмысленное сообщение”**. В сообщении запишите изменения, которые появились с момента прошлого коммита.

Чтобы поместить коммит **в репозиторий**, используется **git push ИМЯ\_РЕПОЗИТОРИЯ ИМЯ\_ВЕТКИ**. Обычно репозиторий один (origin), и ветка одна (master), поэтому: **git push origin master.**