|  |  |
| --- | --- |
| **ЖДТУ** | **Міністерство освіти і науки України**  **Житомирський державний технологічний університет** |

Затверджено науково-методичною

радою ЖДТУ

протокол від «\_\_»\_\_\_\_\_\_\_ 20\_\_ р. №\_\_

Методичні рекомендації

для проведення лабораторних робіт

з навчальної дисципліни

**«БАЗИ ДАНИХ»**

для студентів освітнього рівня «БАКАЛАВР»

денної форми навчання

спеціальності 121 «Інженерія програмного забезпечення»

освітньо-професійна програма «Інженерія програмного забезпечення»

факультет інформаційно-комп’ютерних технологій

кафедра інженерії програмного забезпечення

Обговорено на засіданні кафедри програмного забезпечення систем

Протокол № \_\_

від „\_\_” серпня 2017 р.

Розробник: к.т.н., доцент кафедри інженерії програмного забезпечення І.І. Сугоняк

Житомир, 2018

ЗМІСТ

[Лабораторна робота № 1. ПРОЕКТУВАННЯ ДОДАТКУ ДО БД НА ОСНОВІ ADO.NET 3](#_Toc506778872)

[Лабораторна робота № 2. ПРОЕКТУВАННЯ ЗВІТНОСТІ СИСТЕМИ 17](#_Toc506778873)

[Лабораторна робота № 3. ВИКОЛИК ПРОЦЕДУР, ЩО ЗБЕРІГАЮТЬСЯ 21](#_Toc506778874)

[Лабораторна робота № 4. ОРГАНІЗАЦІЯ ДОДАТКІВ ДО БД ІЗ ВИКОРИСТАННЯМ   
WPF 27](#_Toc506778875)

[Лабораторна робота № 5. Доступ до БД за допомогою Entity Framework 6.x 36](#_Toc506778876)

[Лабораторна робота № 6. РЕАЛІЗАЦІЯ БД ТА ІНТЕРФЕСІВ ДЛЯ WEB-ОРІЄНТОВАНИХ СИСТЕМ (PHP+MYSQL) 55](#_Toc506778877)

[Лабораторна робота № 7. РАБОТА З NOSQL MONGDB 62](#_Toc506778878)

# Лабораторна робота № 1. ПРОЕКТУВАННЯ ДОДАТКУ ДО БД НА ОСНОВІ ADO.NET

Навчальна та виховна мета: набуття практичних навичок створення інтерфейсів користувача із використанням технологій та інструментарію ІCP MS VISUAL STUDIO.

Навчально-матеріальне забезпечення: MS VISUAL STUDIO, MS SQL SERVER

Література: [7, c.24-31], [8, c.27-49]

**Завдання на лабораторну роботу**

На зайнятті необхідно:

1. Створити головну форму додатку до БД;
2. Створити інтерфейсні форми доступу до даних таблиць БД, зокрема

**а) прості форми:**

Товар (таблична форма) із можливостями вставки та оновлення даних

Постачальник (форма на основі компонентів – поля вводу), для неї передбачити фільтрацію даних;

**б) складни форми:**

Постачальник-Товар із використанням елементів DataGrid та інших компонентів;

Заказ-замовлені товари, що ієрархічно відображує дані.

Хід роботи

**Завдання 1. Створення головної форми додатку:**

1. Запустити MS SQL Server Manager та відновити підключення до БД, створеної на попередньому заняття. Вікно Manager не закривати, в подальшому його можна використовувати для перегляду точних назв(ідентифікаторів) об‘єктів бази даних.
2. Створити проект WindowsFormApplication та зберегти його.
3. На форму винести компонент MainMenu та організувати його структуру наступним чином:

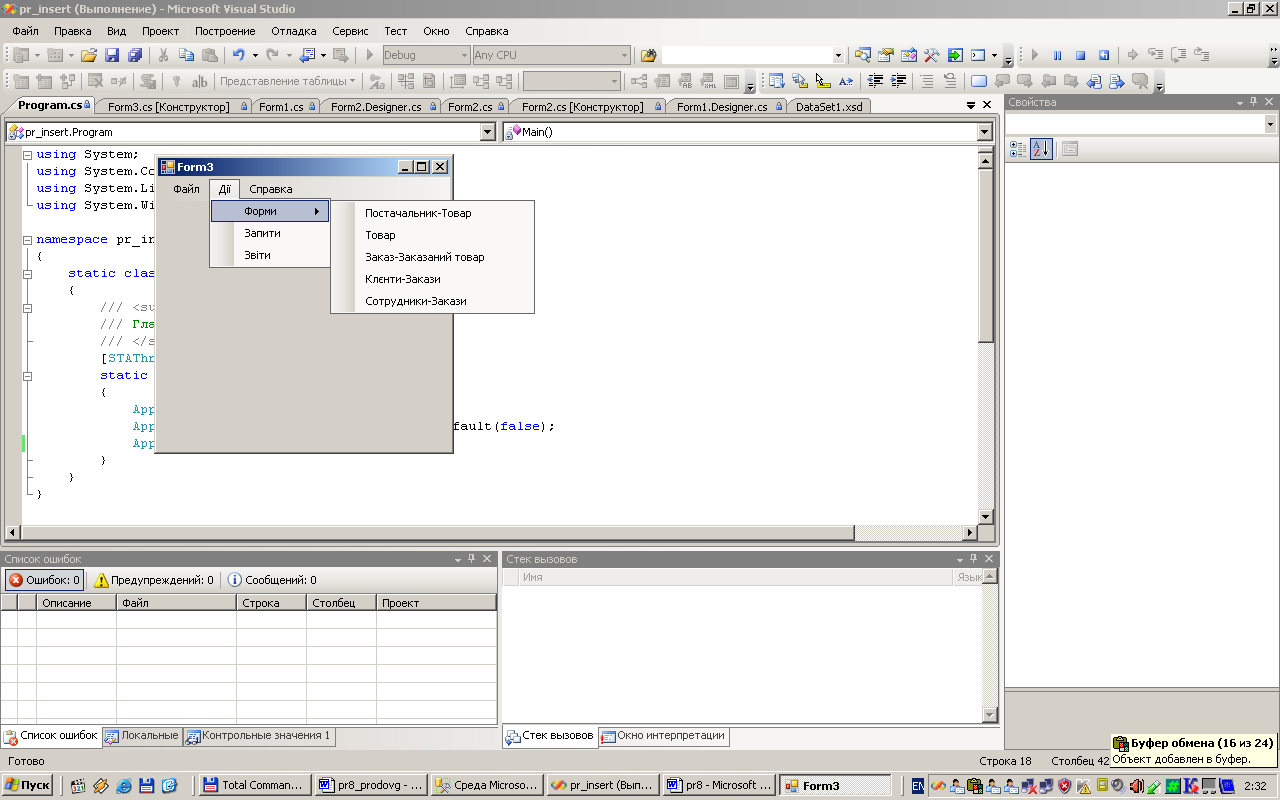


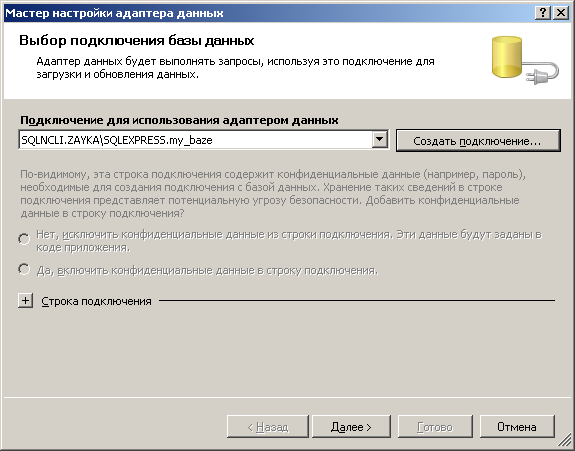
Рис.1

Для запуску інших форм, що будуть створені в проекті для відображення даних в обробник події Click відповідного меню заноситься наступний код (приклад для виклику об‘єкта Form2):

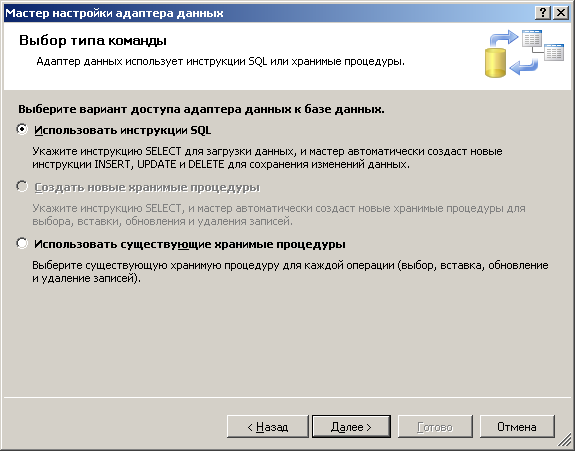


**Завдання 2.1. Створення простої табличної форми**

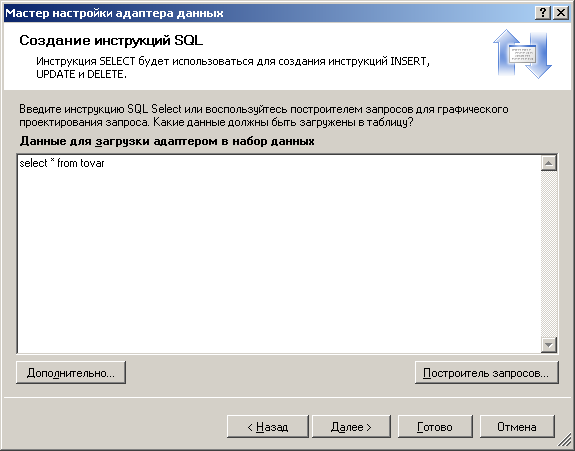
1. Для створення з‘єднання з БД на форму винести компонент SQLDataAdapter(OleDbDataAdapter) (вкладка Date).
2. Створити з‘єднання до БД, що створена на попередньому практичному занятті. З‘єднання створюємо до таблиці „Tovar” використовуючи майстер на лаштування адаптера:



а) перший крок (\* назва сервера обирається за переліком і може відрізнятися від наведеної у вікні)



б) другий крок



в) третій крок

Рис.2

Зверніть увагу! На форму для реалізації з‘єднання автоматично додається OleDBConnection

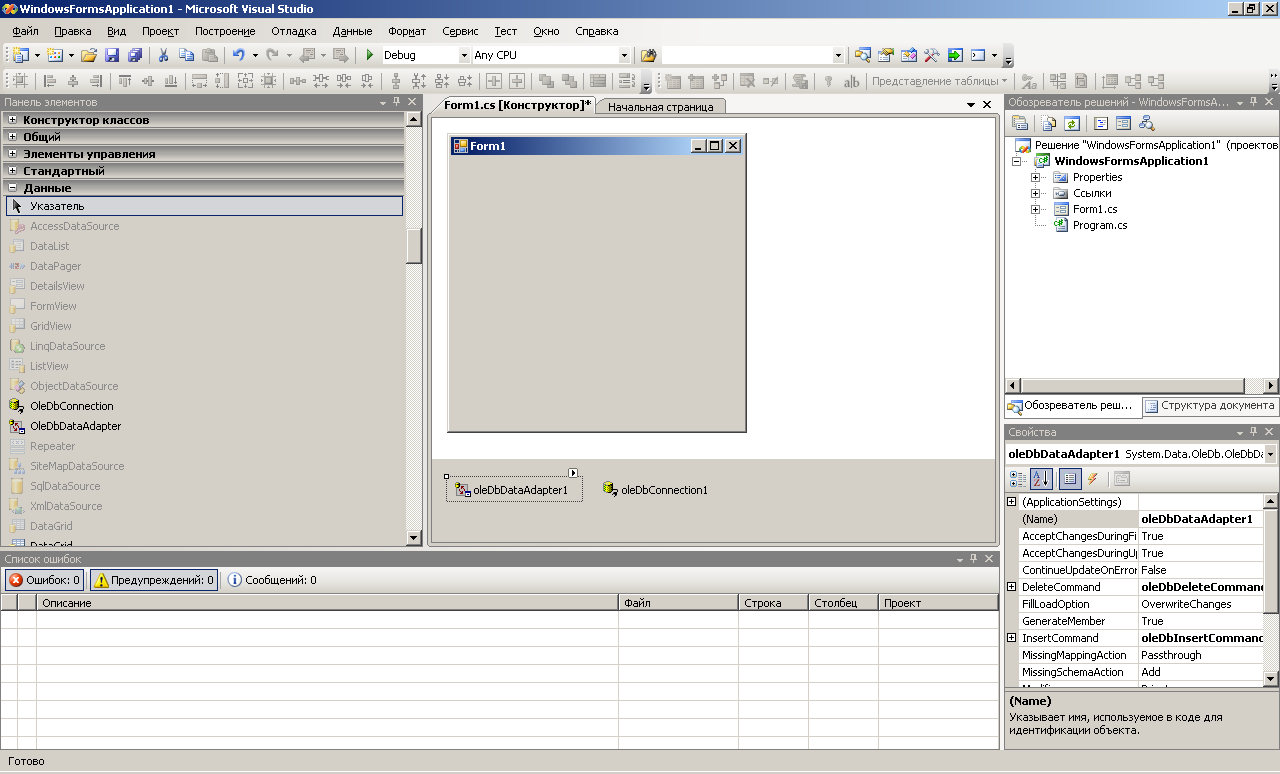


Рис.3

1. На форму необхідно винести наступні компоненти: DataGrid(вкладка Data) та дві кнопки – btnLoad, btnUpdate

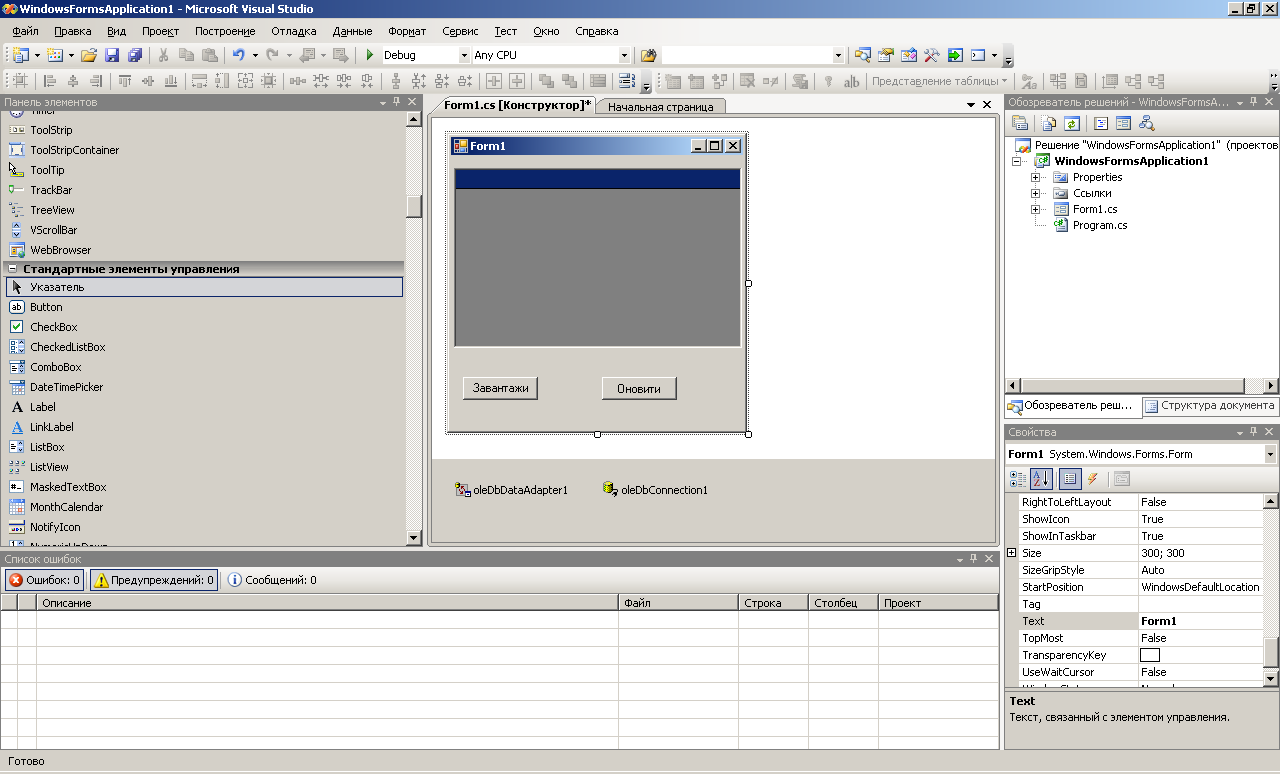


Рис.3

5. Для створення набору даних у пункті меню Data обираємо команду Generate DataSet.

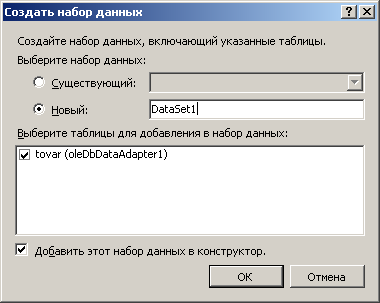


Рис.4

1. Для забезпечення відображення даних необхідно під‘єднати DataGrid до набору даних DataSet11. Для цього необхідно зазначити у властивостях DataGrid1:

DataSource – Dataset11; DataMembers – Tovar;

1. Для забезпечення з‘єднання з базою даних необхідно написати насупний обробник події OnClick для кнопки btnLoad:



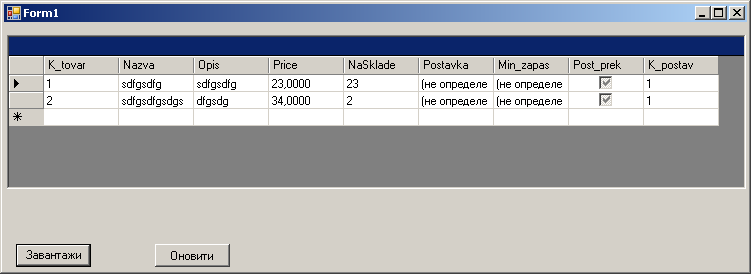


Рис.5

1. Для забезпечення оновлення даних необхідно для написати насупний обробник події OnClick для кнопки btnUpdate:



1. **Написати процедуру відкриття форми відповідним пунктом меню Головного додатку.**
2. Відкомпілювати додаток та перевірити його роботу.

**Завдання 2.2. Створення форми на компонента текстового введення із операціями фільтрафії**

1. Для забезпечення фільтрації даних необхідно модифікувати звичайний запит на вибірку на запит з параметрами (приклад передбачає створення нової форми для відображення даних з таблиці Postachalnik):

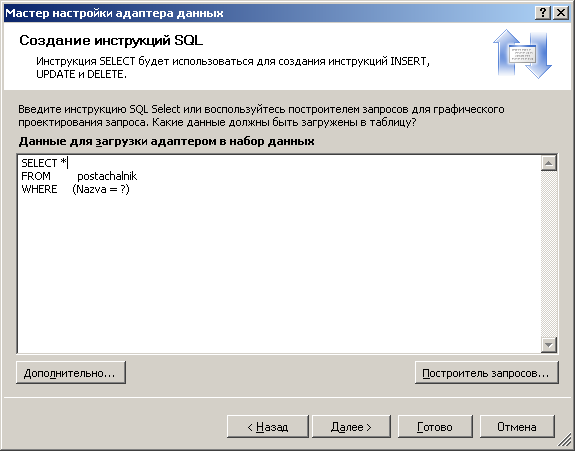


Рис.6

1. Згенерувати набір даних (або використати наявний)

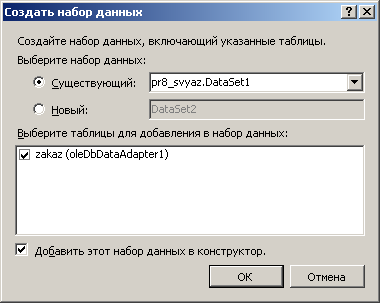


Рис.7

1. Винести на форму наступні елементи управління (назва кнопки – btnShow, назви полів у відповідності до полів БД – txtK\_post, txtNazva, txtAddr тощо).

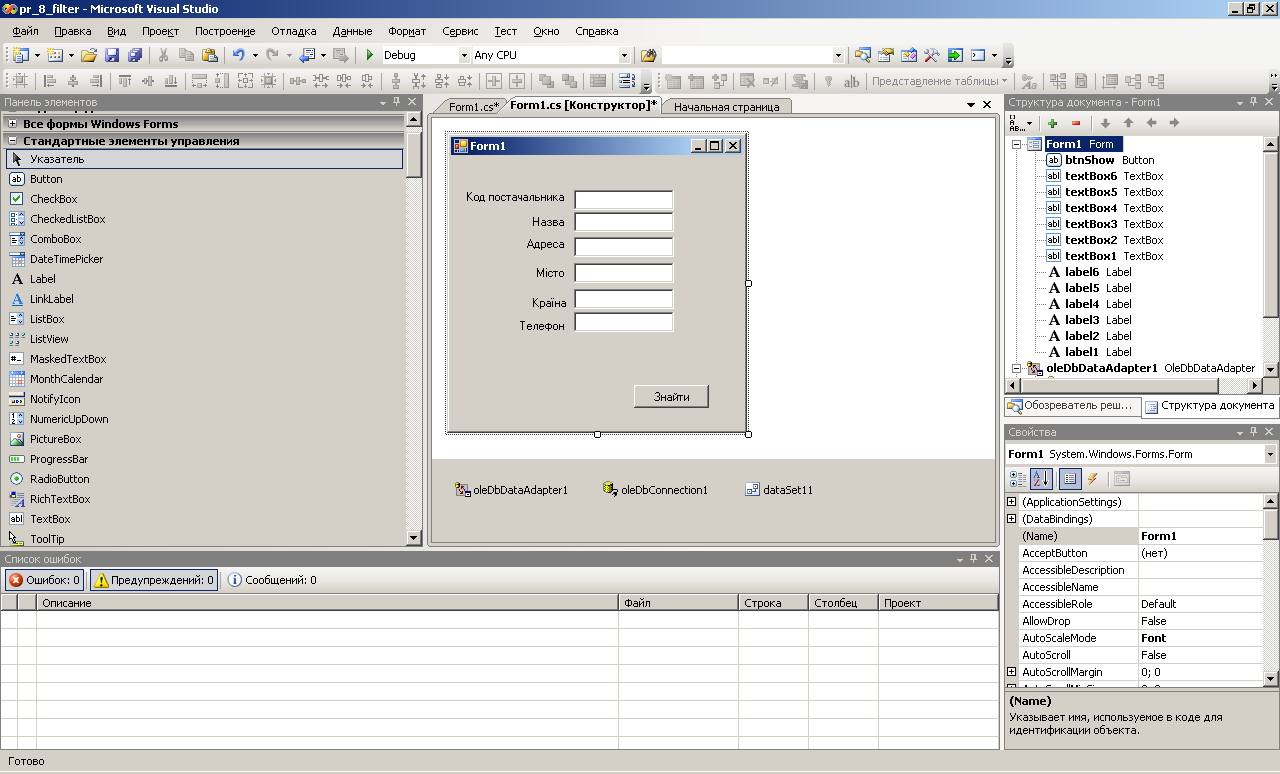


Рис.8

1. Для відображення даних у полях необхідно для кожного поля у властивості Text:

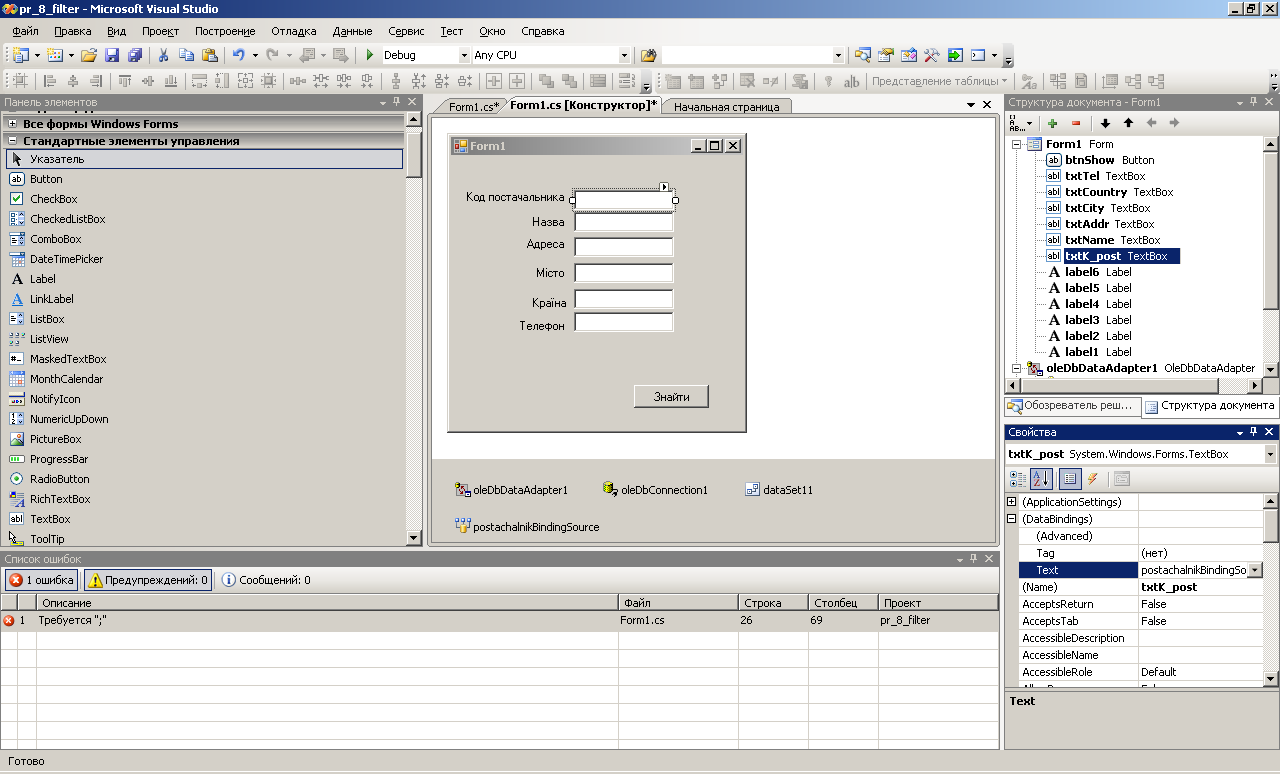


Рис.9

обрати відповідне поле набору даних:

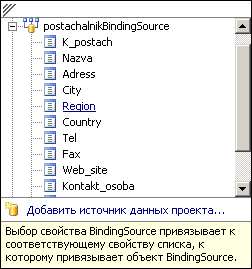


Рис.10

1. Для кнопки передбачити наступний обробник подій:



1. **Написати процедуру відкриття форми відповідним пунктом меню Головного додатку.**
2. Відкомпілювати додаток та перевірити його роботу.

**Завдання 2.3. Складна форма Постачальник-Товар**

1. Для створення складних форм необхідно передбачити опис зв‘язку між таблицями. Для цього необхідно під‘єднати два компоненти OleDBDataAdapter для яких зазначити запити (завдання передбачає створення нової форми):

Select \* From posachalnik;

та

Select \* From Tovar;



Рис.11

згенерувати набір даних (новий або скористатись існуючи без використання DataAdapter):

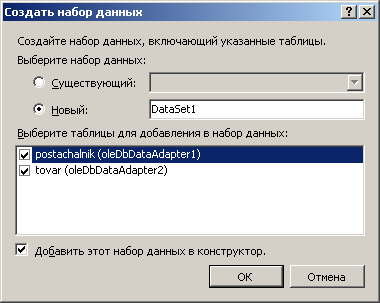


Рис.12

1. Подвійний натиском на елемент DataSet11.xcd у вікні Solution Explorer відкрити конструктор схеми даних та встановити зв‘язок між таблицями. Для цього необхідно перетащити компонент Relation з вкладки DataSet на дочірню таблицю та визначити зв‘язок :

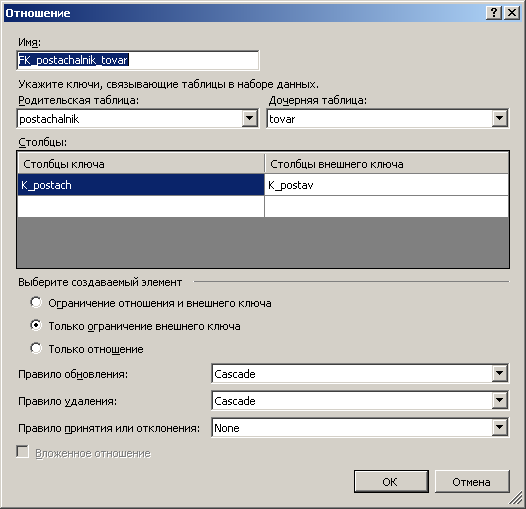


Рис.13

1. Збережіть схему та закрите вікно.
2. На формі розташуйте наступні компоненти відображення даних:

ListBox - для відображення переліку постачальників, та DataGrid – для відображення переліку Товарів, зв‘язаного з постачальниками.

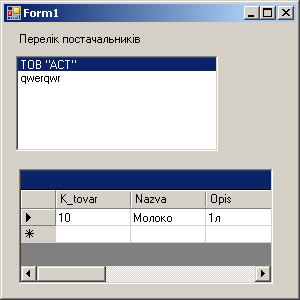


Рис.14

1. Зв‘яжіть компонент ListBox з таблицею Postachlnik:

Для властивості DataSource – обираємо таблицю “postachalnik”,

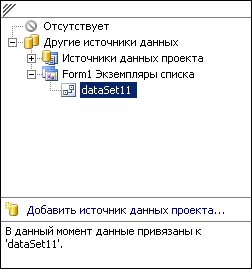


Рис.15

а у властивість DisplayMember – Nazva

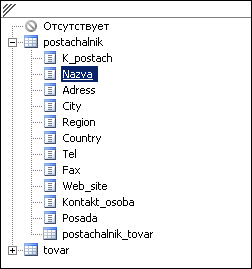


Рис.16

Компоненту DataGrid в свою чергу у властивість DataSource – присвоюємо

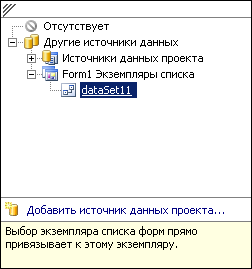


Рис.17

У властивість

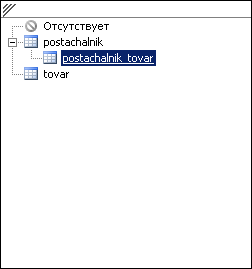


Рис.18

Для заповнення форми необхідно в обробнику полії Load для форми приписати:  


1. **Написати процедуру відкриття форми відповідним пунктом меню Головного додатку.**
2. Відкомпілювати додаток та перевірити його роботу.

**Продовження завдання 2.1. Забезпечення оновлення, знищення та модифікації даних**

1. Модифікувати (Створити) форму Tovar:

‑ додати кнопи – btnInsert; btnDelete;

* додати поля введення textBox – 8 шт., що віповідають полям таблиці Товар, та comboBox для поля К\_postach.

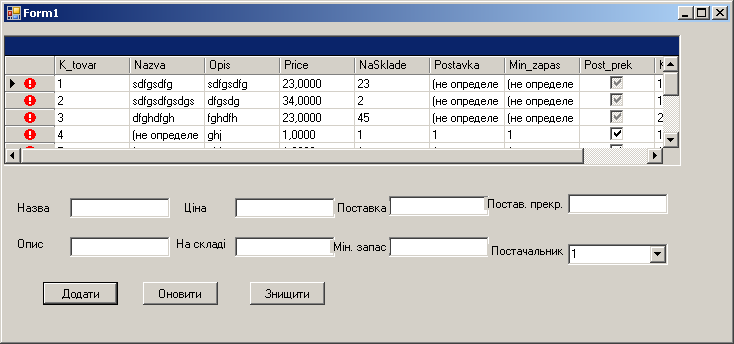


Рис.19

1. Для компонента DataGrid –переглянути, що властивості DataSet та DataSource задані наступним чином:

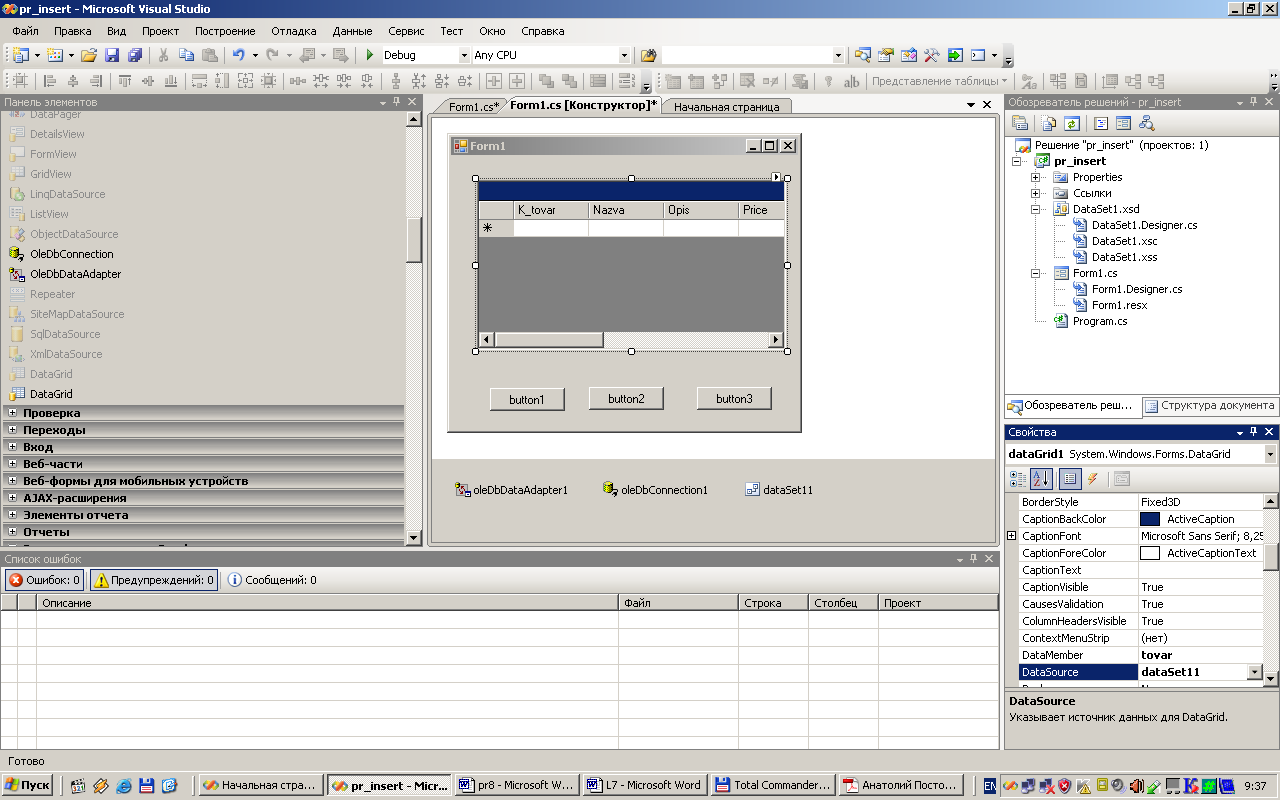


Рис.20

1. У вікні Server Explorer встановити підключення до БД і обрати поля таблиці Tovar (на рис. виділено декілька, можна обирати усі):

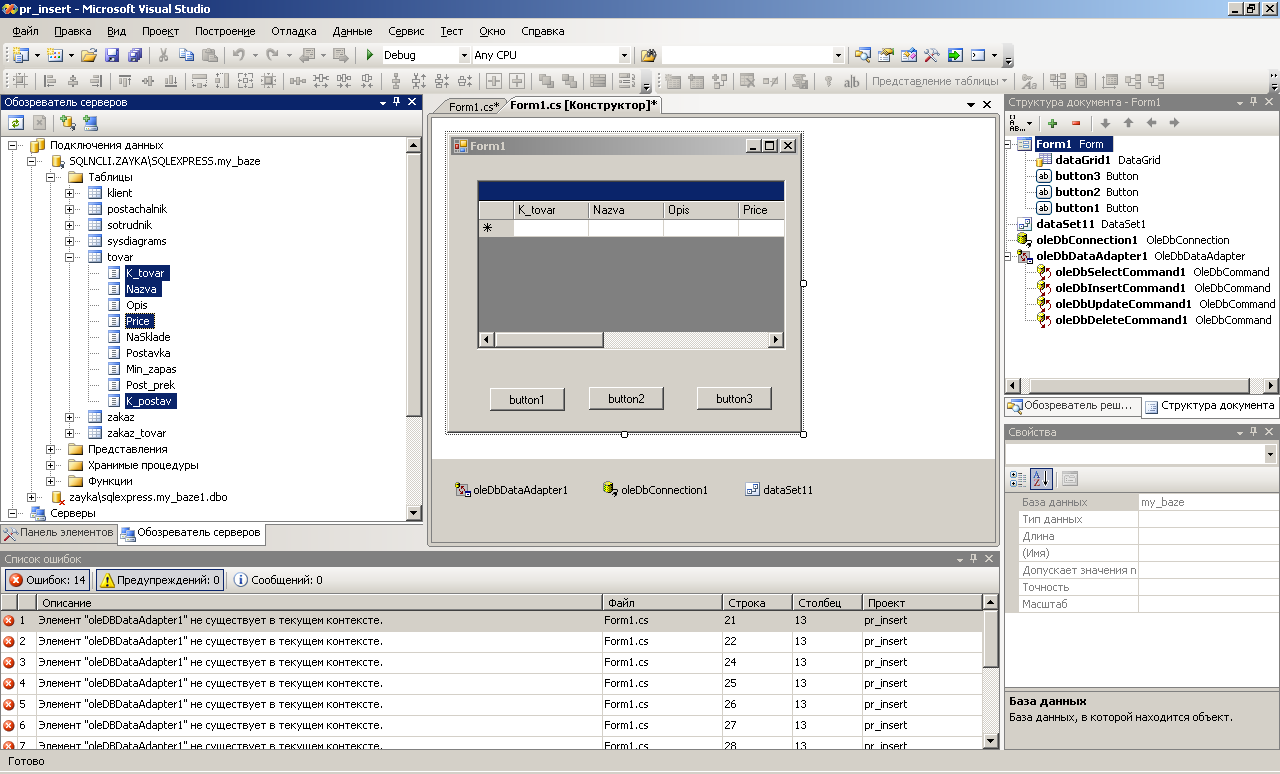


Рис.21

1. Для компонента comboBox1 задати наступні властивості:

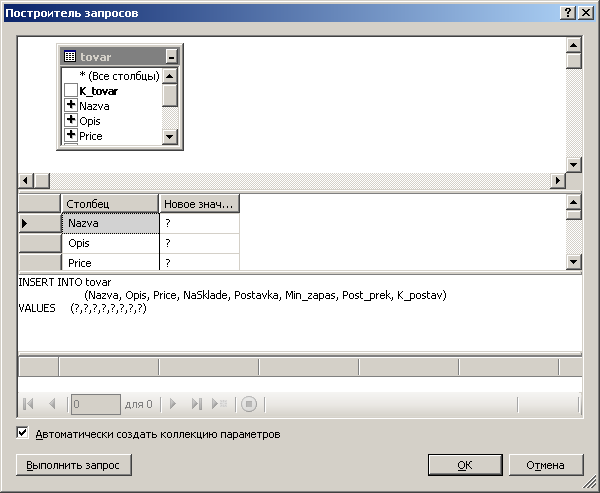
DataSource – DataSet11;

DisplayMember – postachalnik.k\_postа;

Для події Load компонента Form написати обробник з наступним програмним кодом:



1. Для реалізації операцій вставки оберіть у вікні властивостей OleDbAdapter пункт InsertCommand у переліку якого найдіть властивість CommandText, де відкрийте конструктор запитів та перегляньте, що текст запиту наступний:



1. Для забезпечення виконання операції вставки необхідно виконати наступний алгоритм:
2. Зчитати значення параметрів (полів запиту Insert) для відповідного компонента DataAdatter;
3. Під’єднатися до БД;
4. Виконати запит;
5. Оновити дані (Очистити набір даних, заповнити знову).

Дан послідовність дій реалізується у обробнику події OnClick кнопки btnInsert настуним чином:



1. **Написати процедуру відкриття форми відповідним пунктом меню Головного додатку.**
2. Відкомпілювати додаток, та перевірити коректність роботи програми.
3. Для реалізації операції знищення даних оберіть у вікні властивостей OleDbAdapter пункт DeleteCommand у переліку якого найдіть властивість CommandText, де відкрийте конструктор запитів та модифікуйту текст запиту наступний:



1. Для забезпечення виконання операції знищення необхідно виконати наступний алгоритм:
2. Зчитати значення параметрів (поле Nazva запиту Delete) для відповідного компонента DataAdatter;
3. Під’єднатися до БД;
4. Виконати запит;
5. Оновити DataSet (очистити набір даних, заповнити знову).
6. Дан послідовність дій реалізується у обробнику події OnClick кнопки btnDelete настуним чином:



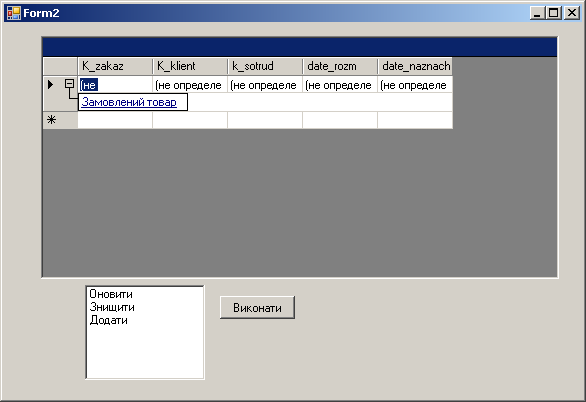
1. Для реалізації операції знищення модифікувати обробник події OnClick кнопки btnUpdate наступним чином:



1. Відкомпілювати додаток, та перевірити коректність роботи програми.

З**авдання 2.4. Складна форма з ієрархічним відображенням даних**

1. Створити нову форму:



1. Розмістити на ній наступні компоненти:

DataGrid, ListBox, Button та DataAdapter – 2 шт.

1. За допомогою адаптерів даних встановити зв‘язок з таблицями Zakaz та Zakaz\_tovar, для цього необхідно записати відповідні запити SQL:

Select \* from zakaz;

Select \* from zakaz\_tovar;

1. Для встановлення зв‘язків між наборами даних та забезпечення ієрархічного відображення даних необхідно у обробнику події Load для форми написати наступний код:



1. **Написати процедуру відкриття форми відповідним пунктом меню Головного додатку.**
2. Відкомпілювати додаток, та перевірити коректність роботи програми.

**Завдання на самостійну роботу**

1. Створити складні форми, що забезпечують доступ та модифікацію до у інших таблицях БД, або прості форми та організувати на них або ієрархічне відображення даних.
2. Передбачити на них можливість введення, знищення та оновлення даних.
3. Організувати відображення всіх форм у межах єдиного проекту.

# Лабораторна робота № 2. ПРОЕКТУВАННЯ ЗВІТНОСТІ СИСТЕМИ

Навчальна та виховна мета: набуття практичних навичок створення інтерфейсів користувача із використанням технологій та інструментарію ІCP MS VISUAL STUDIO.

Навчально-матеріальне забезпечення: MS VISUAL STUDIO, MS SQL SERVER

Література: [7, c.24-31], [8, c.27-49]

**Завдання на лабораторну роботу**

На зайнятті необхідно:

**Завдання 1.** Створити просту форму „Tovar” із можливістю переміщення за записами, оновлення та знищення записів;

**Завдання 2.** Створити складну форму із розрахунковими полями на основі компонента DataGridViewer.

**Завдання 3.** Створити звіти за наведеними запитами:

* перелік товарів на складі.
* перелік замовлень за постачальниками.

**Хід роботи**

1. Створити в проекті нову форму та розмітити на ній наступні компоненти:

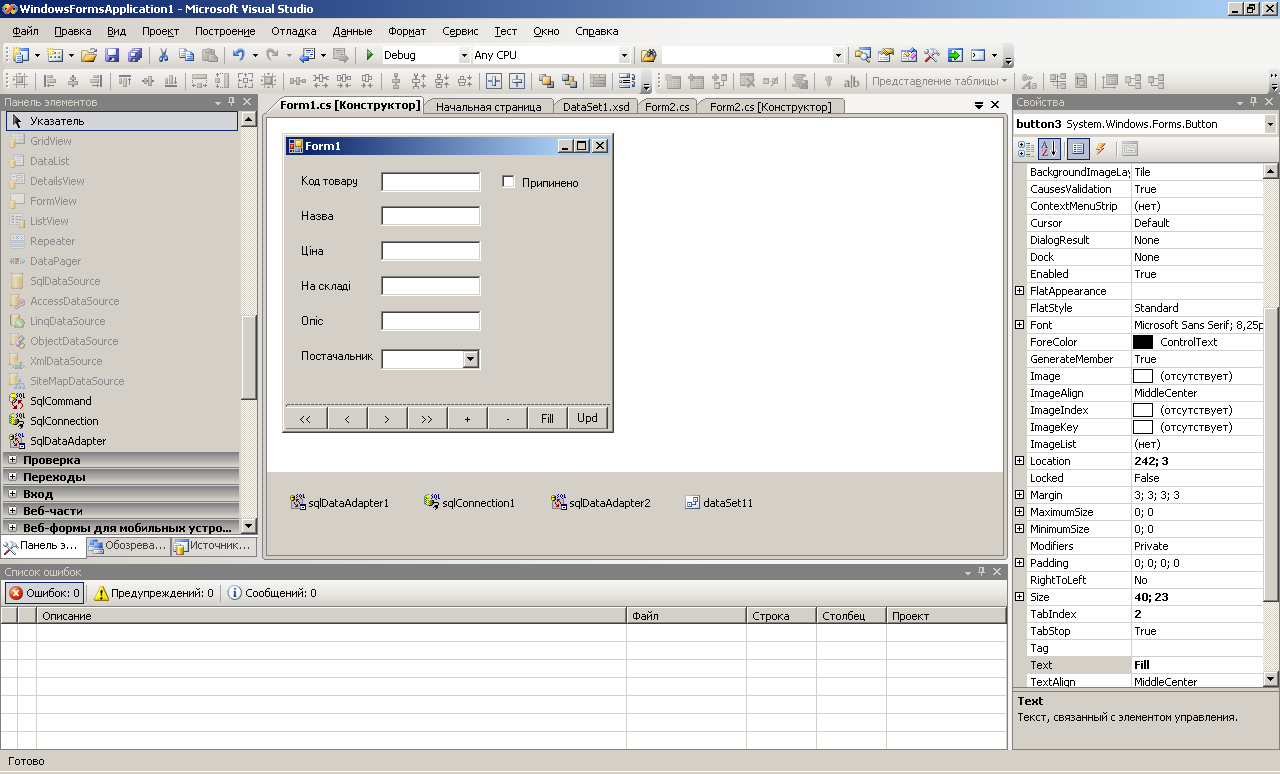


Рис.1

1. Зв‘язати її з джерелом даних таблицею „Tovar”;

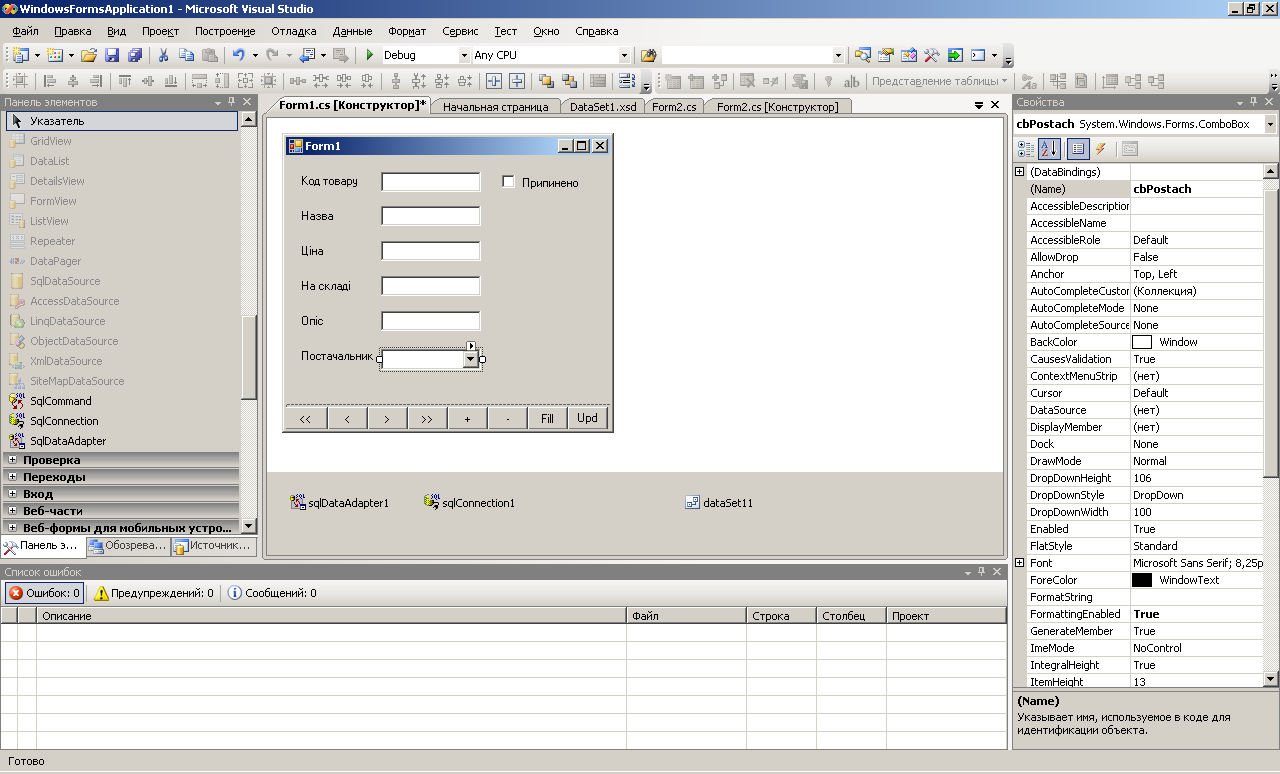


Рис.2

1. Для реалізації подій кнопок додати наступний код:





1. Для того щоб дані введені в поля запа‘ятовувались необхідно прописати наступні процедури для всіх полів введення:



1. Відкомпелювати додаток та перевірити його роботу.
2. Перейти в діагарму даних DataSet11 та створити у таблиці Tovar новее поле Itog. У властивості Expession занотовати наступне:

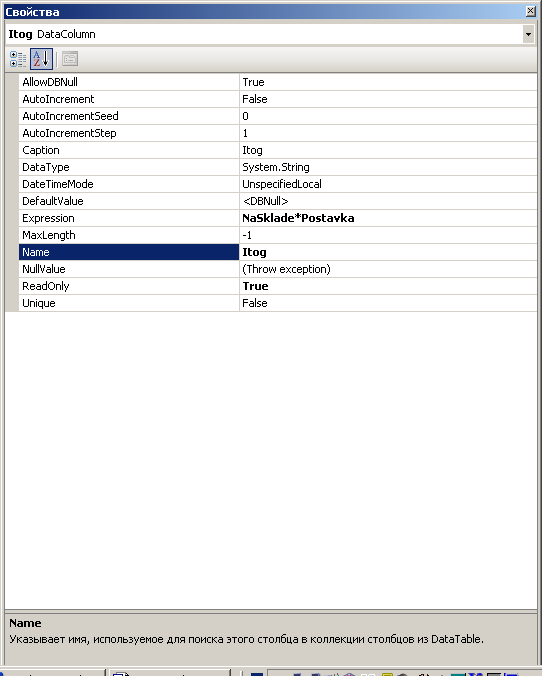


Рис.3

1. Відкомпелювати додаток та перевірити його роботу.
2. Створити нову форму. На неї винести компонент sqlDataAdapter та прописати в ньому наступний запит:

Select Tovar.Nazva, Postachalnik.Nazva, Tovar.Price, Tovar.NaSklade

From Tovar, Postachalnik

Whrere Tovar.k\_postach=postacalnik.k\_postach;

1. Винести на форму компонент MicrosoftReportViewer та згенерувати звіт.

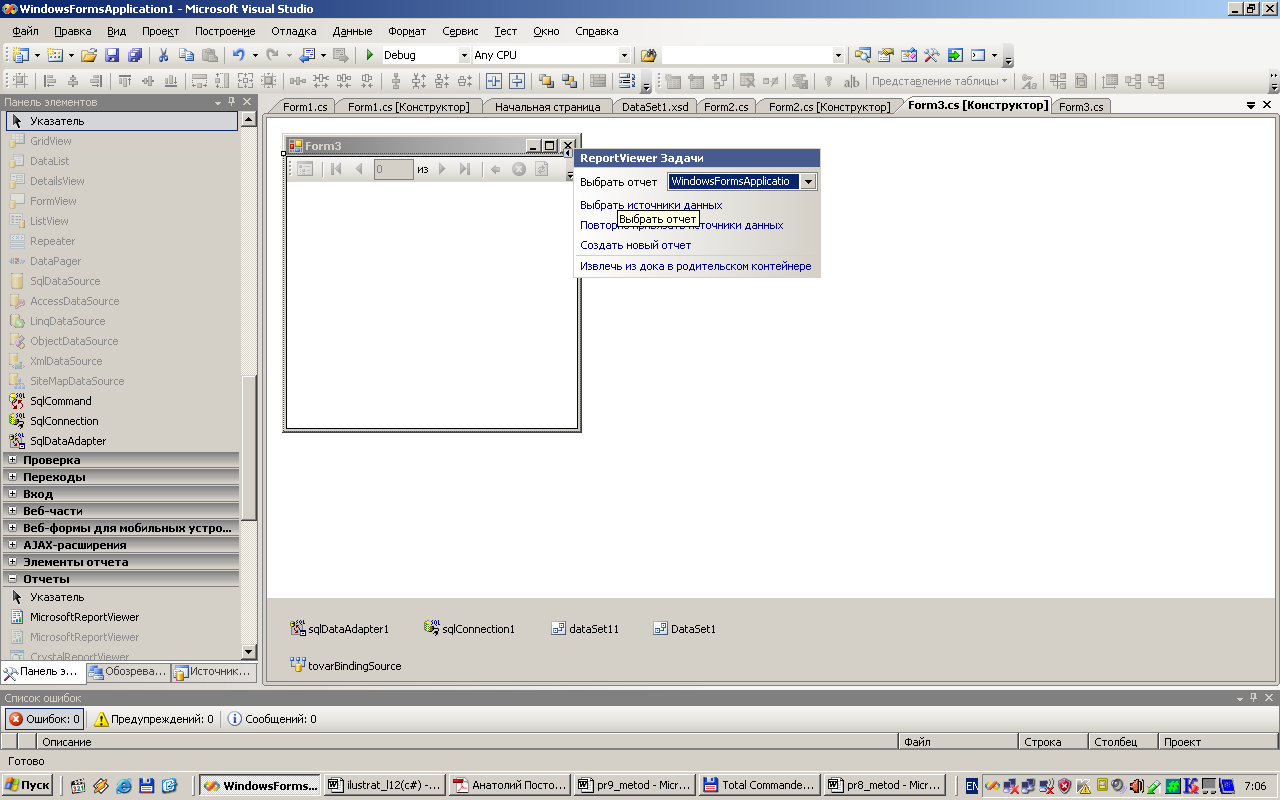


Рис.4

1. Перевірити чи з‘явився він у пункті „Выбрать отчет”. Якщо ні додати його.
2. Відкомпелювати додаток та перевірити його роботу.
3. Аналогічним чином створити звіт за замовленнями постачальників.

**Завдання на самостійну роботу:**

Визначити механізм реалізації полів підстановки на основі компонентів ComboBox та DataGridViewComboBoxColumn.

# Лабораторна робота № 3. ВИКОЛИК ПРОЦЕДУР, ЩО ЗБЕРІГАЮТЬСЯ

Навчальна та виховна мета: набуття практичних навичок створення процедур, що зберігаються та їх виклик із зовнішніх додатків.

Навчально-матеріальне забезпечення: MS VISUAL STUDIO, MS SQL SERVER(MY SQL Server)

Література: [7, c.24-31], [8, c.27-49]

Технічні засоби: ПЕОМ

Питання до обговорення:

1. Синтаксис процедури, що зберігається за стандартом ISO
2. Параметри в процедурах, що зберігаються
3. Розгалуження в процедурах, що зберігаються
4. Синтаксис тригерів в середовище MS SQL Server
5. Синтаксис тригерів в середовище MYSQL Server
6. Події у тригерах.
7. Виклик процедур, що зберігаютсья
8. Ініціація тригерів

1. Вступна частина:

* перевірка присутності студентів
* оголошення теми та мети практичного заняття
* ознайомлення з планом проведення заняття

3. Основна частина:

1. Виклик процедур, що зберігаються, які повертають дані.
2. Виклик процедур, що зберігаються, які не повертають дані.
3. Виклик процедур, що зберігаються, із передачею параметів.
4. Керування транзакціями.

4. Заключна частина:

* відповіді на питання студентів
* підбиття підсумків та ознайомлення із завданням на самопідготовку

Порядок виконаня роботи

1. Застосування процедур, що зберігаються, з параметрами, пов'язане з інтерфейсом - користувач має можливість вводити значення і потім на підставі його отримувати результат. Для демонстрації програмної роботи з процедурами, що зберігаються, створимо нову Windows -форму, головне вікно якого представлене на рис. 1.

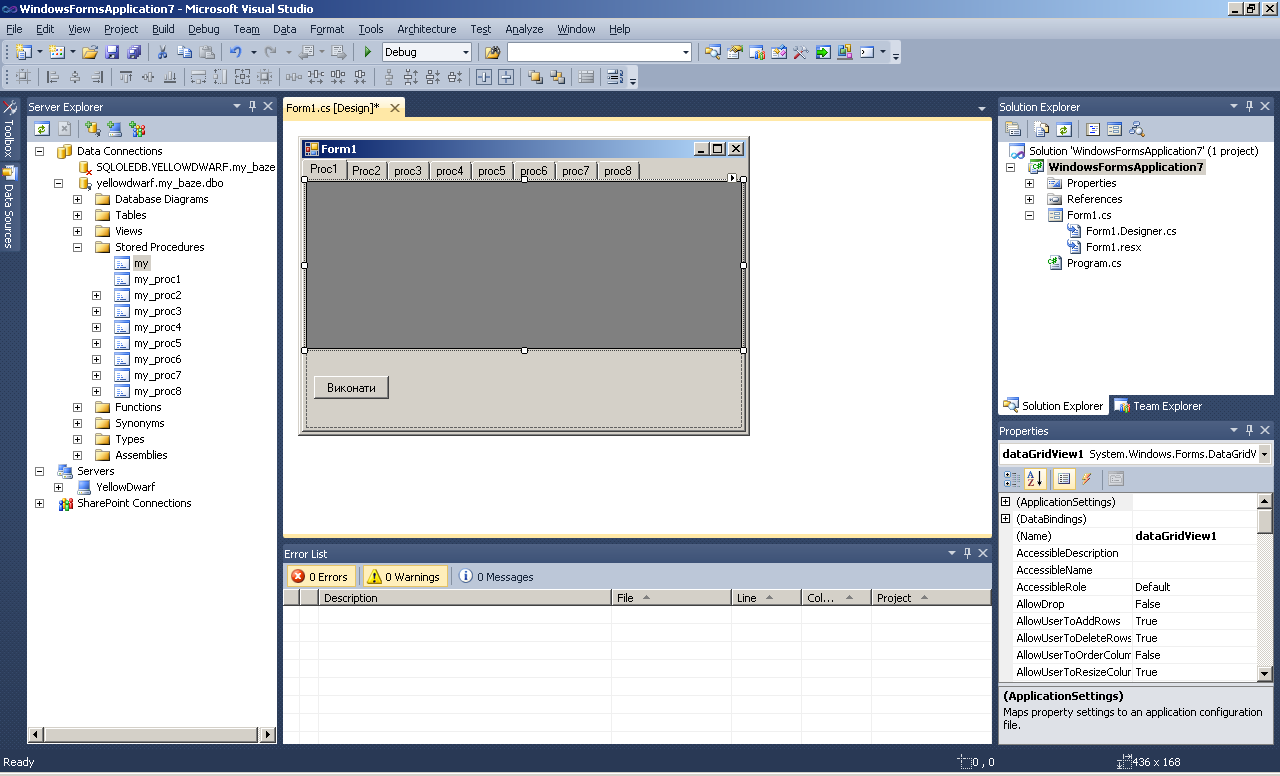


Рис.1

1. На вкладці розмістити компоненти sqlConnection та sqlCommand. Визначити параметри під’єднання компонента sqlConnection1 таким чином:

Data Source=ServName;Initial Catalog=trog\_firm;Integrated Security=True

Де ServName – назва сервера, trog\_firm – назва бази до якої під’єднано форму.

Для обєкту sqlCommand1 визначити параметри об’єкта згідно рис.2.

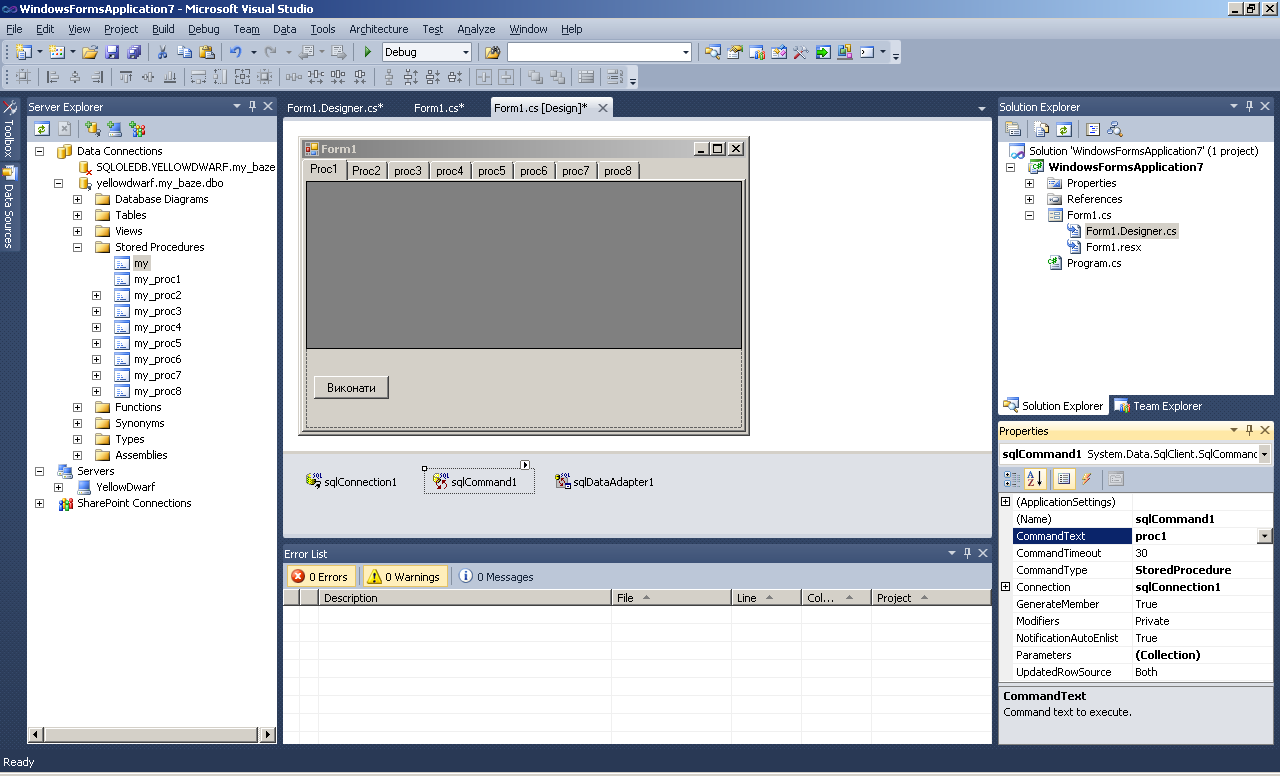


Рис.2

Для відображення даних використати компонент sqlDataAdapter.

В обробнику події компонента Button1->Click визначити такий код:

private void button1\_Click(object sender, EventArgs e)

{

sqlConnection1.Open();

sqlCommand1.ExecuteNonQuery();

sqlDataAdapter1.SelectCommand = sqlCommand1;

DataSet ds = new DataSet();

sqlDataAdapter1.Fill(ds, "my\_proc1");

sqlConnection1.Close();

dataGridView1.DataSource = ds.Tables["my\_proc1"].DefaultView;

}

Відкомпелювати форму та переглянути результат.

Для перегляду результатів процедури 2 – оновлення даних на вкладку винести два компоненти dataGridView та dataAdapter, які налаштувати для перегляду запиту: SELECT K\_tovar, Nazva, Opis, Price, NaSklade, Postavka, Min\_zapas, Post\_prek, K\_postav

FROM tovar

WHERE (k\_postav = 1)

Для виконання proc2 в обробнику події компонента Button2->Click визначити такий код:

sqlCommand1.CommandText = "my\_proc2";

sqlConnection1.Open();

int UspeshnoeIzmenenie=sqlCommand1.ExecuteNonQuery();

dataSet11.Clear();

sqlDataAdapter2.Fill(dataSet11);

if (UspeshnoeIzmenenie !=0) {

MessageBox.Show("Изменения внесены", "Изменение записи");

}

else {

MessageBox.Show("Не удалось внести изменения", "Изменение записи");

}

sqlConnection1.Close();

}

}

Для виконная процедури із параметрами “my\_proc3” передбачити на формі наступні елемени:

TextBox – для введення назви клієнта.

ListBox – для відображення результату

Button – для керування.

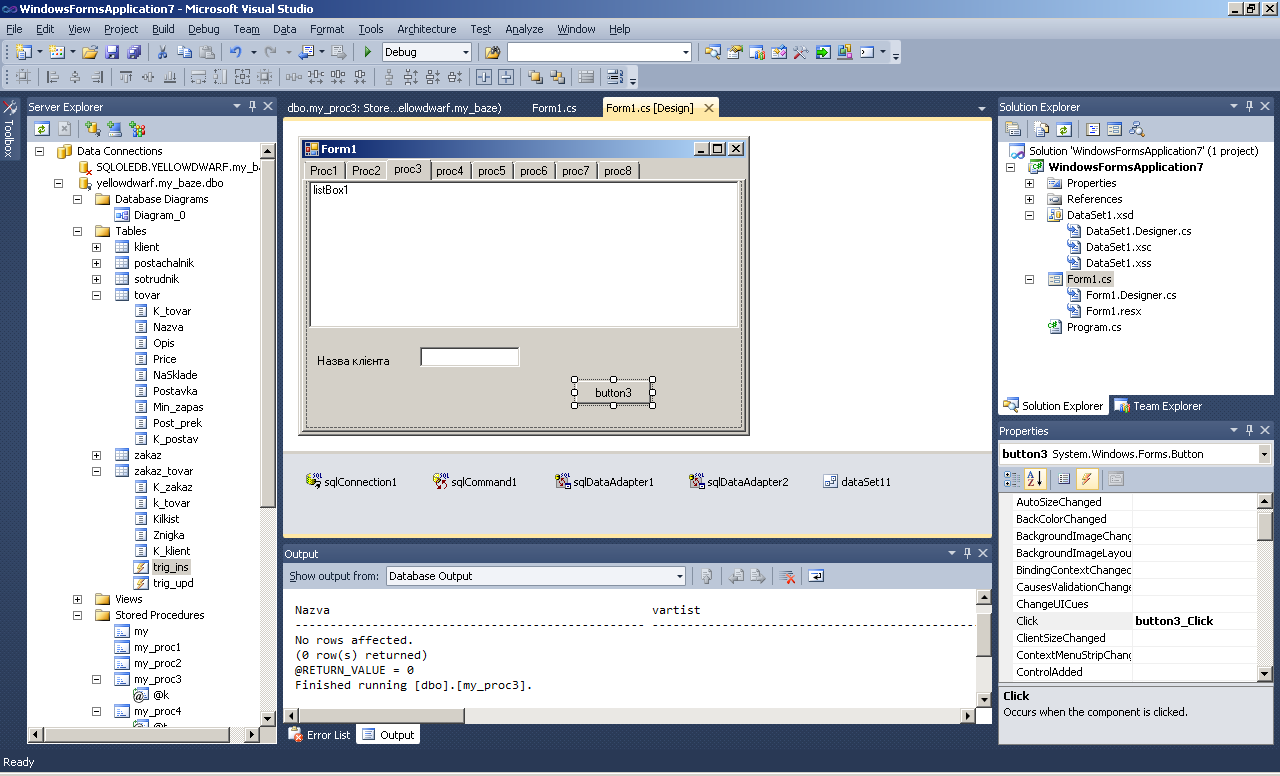


Рис.3

В обробнику події компонента Button3->Click визначити такий код:

sqlCommand1.CommandText = "my\_proc3";

sqlCommand1.Parameters.Add("@k", SqlDbType.VarChar, 20);

sqlCommand1.Parameters["@k"].Value = Convert.ToString(textBox1.Text);

sqlConnection1.Open();

System.Data.SqlClient.SqlDataReader reader=sqlCommand1.ExecuteReader();

while (reader.Read()) {

string Tovars = reader.GetString(0);

float Vartist = reader.GetFloat(1);

string Klient = reader.GetString(1);

listBox1.Items.Add("Товар: "+Tovars+"Вартість: " + Vartist + "Клієнт: " + Klient);

}

reader.Close();

sqlConnection1.Close();

}

}

Аналогічним чином реалізувати інші процедури.

1. Реалізувати процедуру заповнення НД і контролю зміни даних наступним чином:



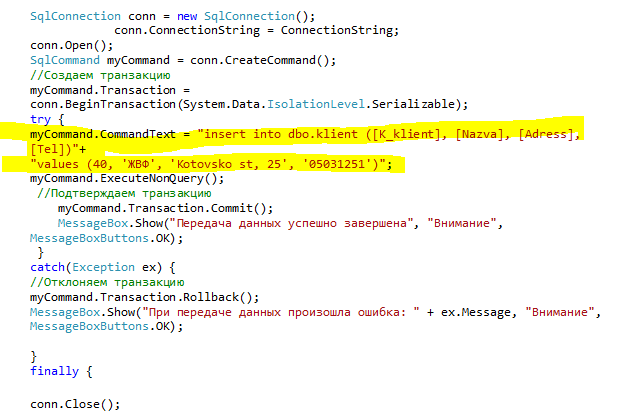




Для події закриття форми реалізувати відстеження змін:



1. Для процедур оновлення даних реалізувати оброку транзакцій за зразком:



Виділений жовтим кольором фрагмент має бути замінено на запит із параметрами.

Завдання на самостійну роботу.

1. Реалізувати виклик 4 процедур, що зберігаються із використанням властивостей DataAdapter.
2. Реалізувати виведення повідомлень тригерів на екран у додатку.

# Лабораторна робота № 4. ОРГАНІЗАЦІЯ ДОДАТКІВ ДО БД ІЗ ВИКОРИСТАННЯМ WPF

Навчальна та виховна мета: набуття практичних навичок створення процедур, що зберігаються та їх виклик із зовнішніх додатків.

Навчально-матеріальне забезпечення: MS VISUAL STUDIO, MS SQL SERVER(MY SQL Server)

Література: [7, c.24-31], [8, c.27-49]

Технічні засоби: ПЕОМ

## Завдання на лабораторну роботу

1.Реалізувати інтерфейс програми із використанням WPF

2. Реалізувати 3 процедури із параметрами для перегляду даних.

3. Реалізувати не менше 3-х процедур на зміну даних.

## Теоретичні відомості

## Хід роботи

1. Застосування процедур, що зберігаються, з параметрами, пов'язане з інтерфейсом - користувач має можливість вводити значення і потім на підставі його отримувати результат. Для демонстрації програмної роботи з процедурами, що зберігаються, створимо новий додаток, головне вікно якого представлене на рис. 1.

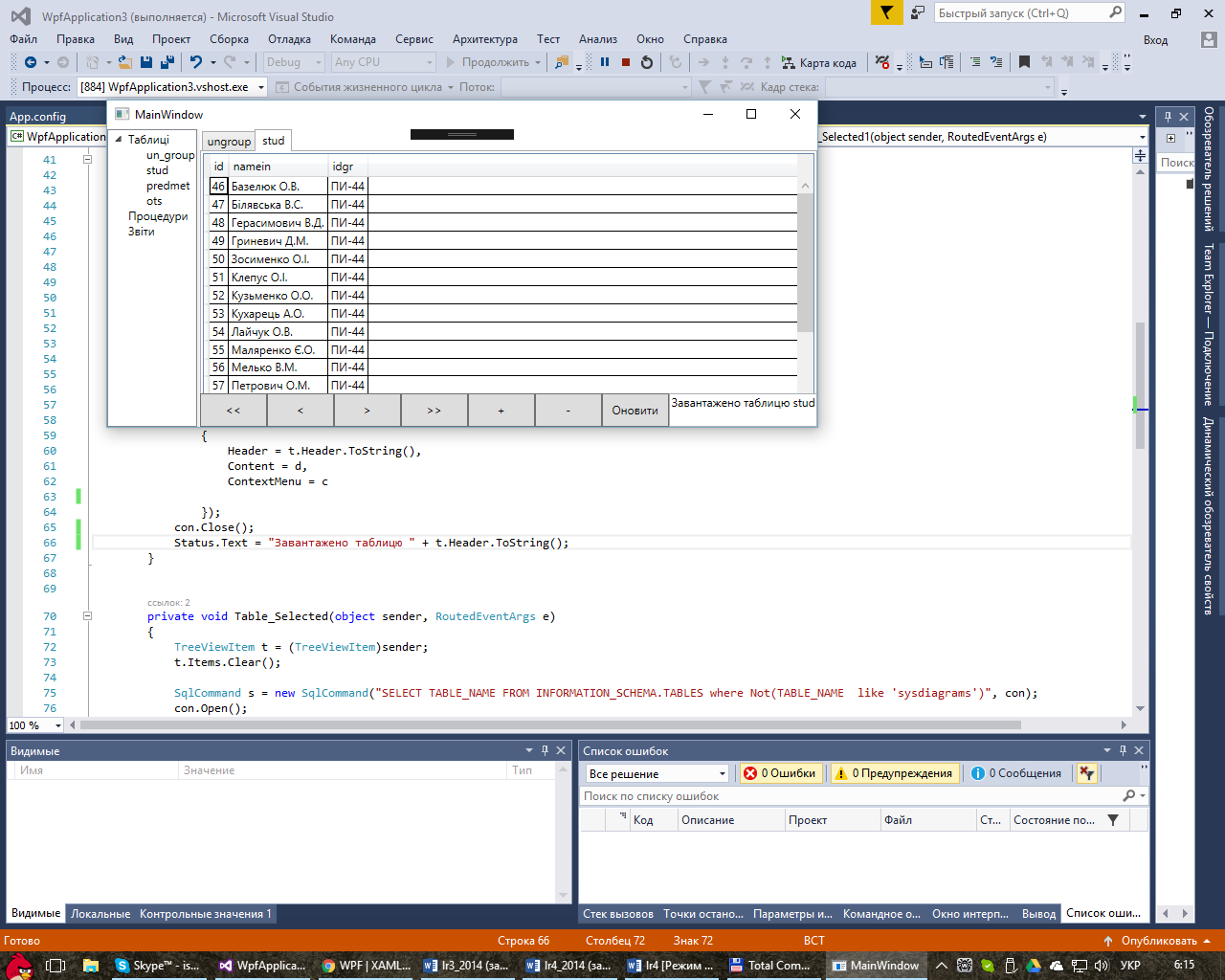


Рис.1

Основними елементами є TreeView – для відображення переліку об’єктів, TabControl – для відображення вмісту(даних) та розташований на елементів DockPanel навігатор, що складається із 7 кнопок керування та TextBox для відображення стану додатку (можливо використовувати поле введення для завдання значень параметрів процедури). Приклад XAML розмітки див. лістинг 2. Зверніть увагу, що реалізації вкладки (TabItem «First») не є обов’язковою, так як всі вкладки створюються програмно.

1. В конфігурації проекту (файл App.config) додати рядок підключення.

Визначити параметри під’єднання компонента sqlConnection1 таким чином:

Data Source=ServName; Initial Catalog=database\_name;Integrated Security=True

де ServName – назва сервера, database\_name – назва бази до якої під’єднано додаток.



Лістинг 1. Вміст файлу App.config



Лістинг 2. Вміст файлу MainMindow.xaml

Для реалізації процедури зчитування переліку таблиць використати об’єкт DataReader наступним чином:



Зверніть увагу, що під’єднання до БД визначається в розділі метода конструктора форми:

public partial class MainWindow : Window

{

SqlConnection con;

DataSet ds;

public MainWindow()

{

InitializeComponent();

con =new SqlConnection(ConfigurationManager.ConnectionStrings["Connect1"].ConnectionString);

ds = new DataSet();

}

Для доступу до об’єктів конфігурації (ConfigurationManager) необхідно додати підключення бібліотеки :

Using System.Configuration та у розділі References(Посилання) відповідну зборку:

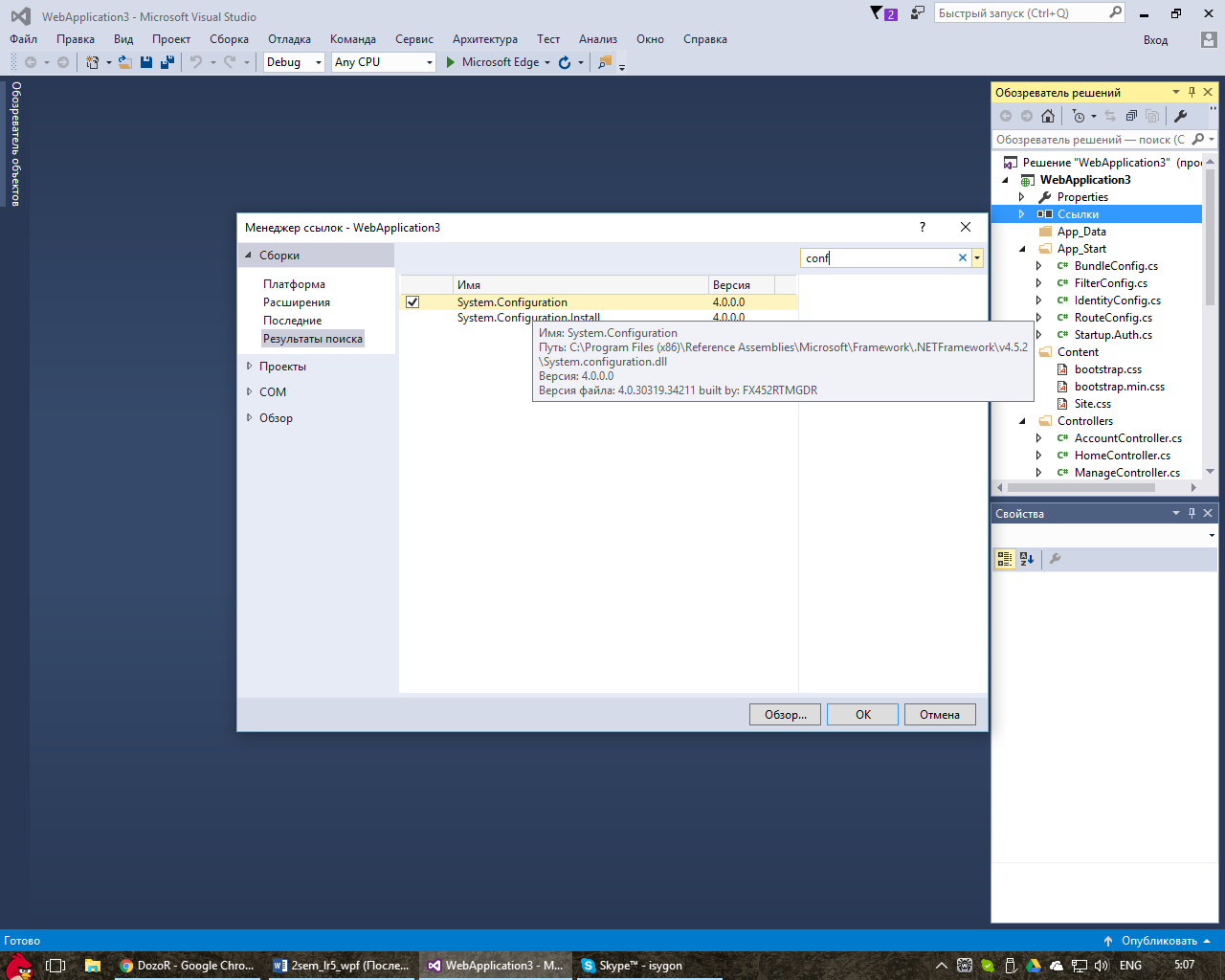


Рис.2

Для відкриття таблиці (метод TableSelected1) можна використати наступну процедуру:

private void Table\_Selected1(object sender, RoutedEventArgs e) {

TreeViewItem t = (TreeViewItem)sender;

SqlCommand s = new SqlCommand("SELECT \* from " + t.Header.ToString() , con);

con.Open();

SqlDataAdapter DA = new SqlDataAdapter(s);

DA.Fill(ds, t.Header.ToString());

DataGrid d = new DataGrid();

d.AutoGenerateColumns = true;

d.ItemsSource = ds.Tables[t.Header.ToString()].DefaultView;

ContextMenu c = new ContextMenu();

MenuItem m = new MenuItem();

m.Header = "Закрити";

m.Click += MenuItem\_Click ;

c.Items.Add(m);

tabs.Items.Add(

new TabItem

{

Header = t.Header.ToString(),

Content = d,

ContextMenu = c

});

con.Close();

Status.Text = "Завантажено таблицю " + t.Header.ToString();

}

Реалізувати процедуру MenuItem\_Click:

private void MenuItem\_Click(object sender, RoutedEventArgs e)

{

tabs.Items.RemoveAt(tabs.SelectedIndex);

}

Для першої процедури створити запит на вибірку всіх даних:

create proc proc1

as select \* from stud

Для виконання процедури використати модифікований текст метода TableSelected1:



Модифікувати процедуру TableSelect, додавши зчитування з сервера переліку процедур:



Для виконная процедури із параметрами (умовна назва “my\_proc3”) передбачити на вкладці елементи TextBox для введення назви клієнта (він є у навігаторі), та кнопку. Компонент можна реалізувати на елементі ListView тоді код виклику процедури буде наступним:

SqlCommand s;

s.CommandType = CommandType.StoredProcedure;

s.CommandText = "my\_proc3";

s.Connection = con;

s.Parameters.Add("@k", SqlDbType.VarChar, 20);

s.Parameters["@k"].Value = Convert.ToString(textBox1.Text);

s.Open();

System.Data.SqlClient.SqlDataReader reader=s.ExecuteReader();

while (reader.Read()) {

string Tovars = reader.GetString(0);

float Vartist = reader.GetFloat(1);

string Klient = reader.GetString(1);

listView1.Items.Add("Товар: "+Tovars+"Вартість: " + Vartist + "Клієнт: " + Klient);

}

reader.Close();

sqlConnection1.Close();

}

}

Створення вкладки на основі компонента ListView відбувається по аналогії із вкладкою з DataGrid.

Аналогічним чином реалізувати виклик не менше 2-х існуючих в БД процедур. Можна використати процедури, створені в попередніх лабораторних работах. Зверність увагу, що обрані процедури мають повертати набори даних.

1. Реалізувати процедуру вставки даних. Для цього на стороні SQL Server створити процедуру, що зберігається:

CREATE proc [dbo].[ins\_stud]

@spets varchar(2),

@kurs int,

@id\_gr varchar(7),

@rez varchar(7) out

as

begin

insert into un\_group

values(@id\_gr, @spets, @kurs)

set @rez = @id\_gr

end

В обробнику події Click кнопки Оновити ввести код,що реалізує налаштування команди InsertCommand адаптера даних та створення колекції параметрів:

SqlCommand s = new SqlCommand("ins\_stud", con);

s.CommandType = CommandType.StoredProcedure;

s.Parameters.Add("@spets", SqlDbType.VarChar, 7, "spets");

s.Parameters.Add("@kurs", SqlDbType.Int, 8,"kurs");

s.Parameters.Add("@id\_gr", SqlDbType.VarChar, 2, "id\_gr");

s.Parameters.Add("@rez", SqlDbType.VarChar, 7);

s.Parameters["@rez"].Direction=ParameterDirection.Output;

DA.InsertCommand = s;

// SqlCommandBuilder comandbuilder = new SqlCommandBuilder(DA);

DA.Update(ds, "un\_group");

DataTable st = ds.Tables["un\_group"];

ds.AcceptChanges();

1. Аналогічним чином реалізувати процедуру редагування даних. Процедура, що зберігається має наступний вигляд:

CREATE proc [dbo].[upd\_stud]

@spets varchar(2),

@kurs int,

@id\_gr varchar(7),

@rez varchar(7) out

as

begin

update un\_group

set spets=@spets, kurs=@kurs

where id\_gr=@id\_gr;

set @rez = @id\_gr

end

Код створення об’єкту SQLCommand наступний:

SqlCommand s = new SqlCommand("upd\_stud", con);

s.CommandType = CommandType.StoredProcedure;

s.Parameters.Add("@spets", SqlDbType.VarChar, 7, "spets");

s.Parameters.Add("@kurs", SqlDbType.Int, 8,"kurs");

s.Parameters.Add("@id\_gr", SqlDbType.VarChar, 2, "id\_gr");

s.Parameters.Add("@rez", SqlDbType.VarChar, 7);

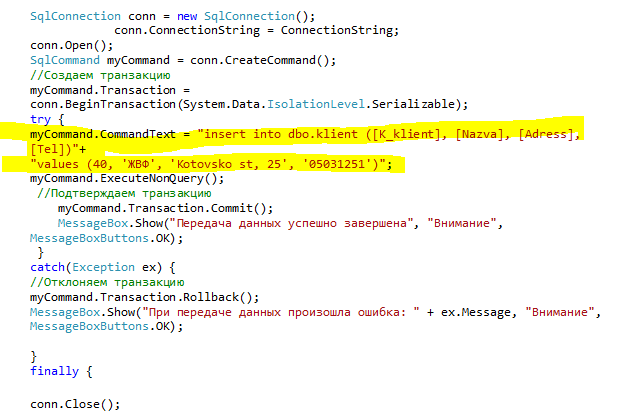
s.Parameters["@rez"].Direction=ParameterDirection.Output;

DA.UpdateCommand = s;

1. Самостійно реалізувати процедуру знищення даних.
2. Для події закриття форми реалізувати відстеження змін:



1. Для процедур оновлення даних можна реалізувати оброку транзакцій за зразком:



Виділений жовтим кольором фрагмент має бути замінено вже створний код процедур модифікації даних, для змінених, доданих або знищених записів.

Для отримання змін в наборі даних можна використати наступний код:

DataSet dch = ds.GetChanges(DataRowState.Added); - копія набору даних, що містить лише додані рядки.

DataSet dch = ds.GetChanges(DataRowState.Modified); - копія набору даних, що містить лише додані рядки.

DataSet dch = ds.GetChanges(DataRowState.Deleted); - копія набору даних, що містить лише додані рядки.

Завдання на самостійну роботу.

1. Реалізувати виведення повідомлень тригерів на екран у додатку.

# Лабораторна робота № 5. ДОСТУП ДО БД ЗА ДОПОМОГОЮ ENTITY FRAMEWORK 6.X

**Завдання на лабораторну роботу**

1. **Створити додаток до БД на основі наявної бази даних із використанням підходу Database First.**
2. **Створити додаток до БД на основі підходу Model First для схеми даних за індивідуальним варіантом.**

**Теоретичні відомості**

Database First був першим підходом, який з'явився в Entity Framework. Даний підхід багато в чому схожий на Model First і підходить для тих випадків, коли розробник вже має готову базу даних.

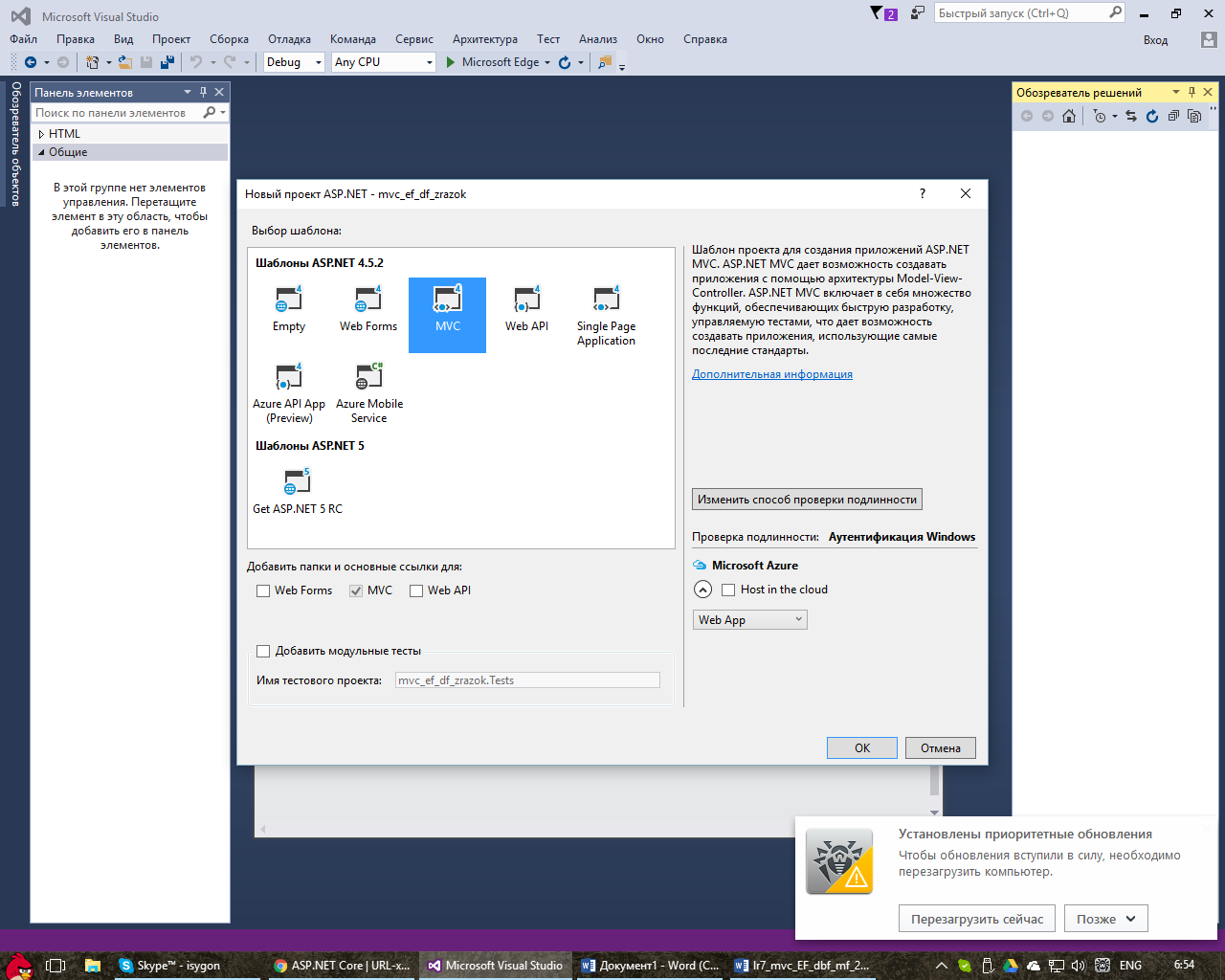
Model First представляє ще один підхід до роботи з Entity Framework. Суть даного підходу полягає в тому, що спочатку робиться модель, а потім по ній створюється база даних.

Щоб Entity Framework міг отримати доступ до бази даних, в системі повинен бути встановлений відповідний провайдер. Так, Visual Studio вже підтримує відповідну інфраструктуру для СУБД MS SQL Server. Для інших СУБД, наприклад, MySQL, Oracle та інших треба встановлювати відповідні провайдери. Список провайдерів для найбільш поширених СУБД можна знайти на сторінці ADO.NET Data Providers.

**ХІД РОБОТИ**

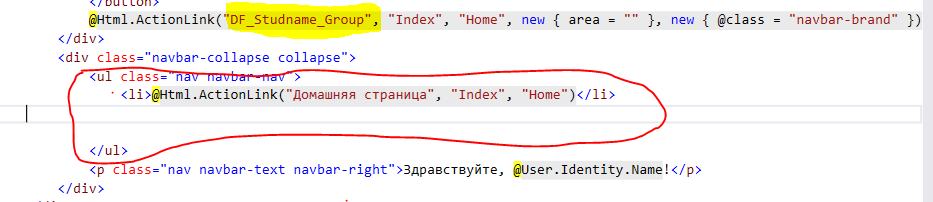
**Database First**

1. Спочтку необхідно створити новий проект типу ASP.Net MVC.



Модифікуємо створений проект наступним чином:

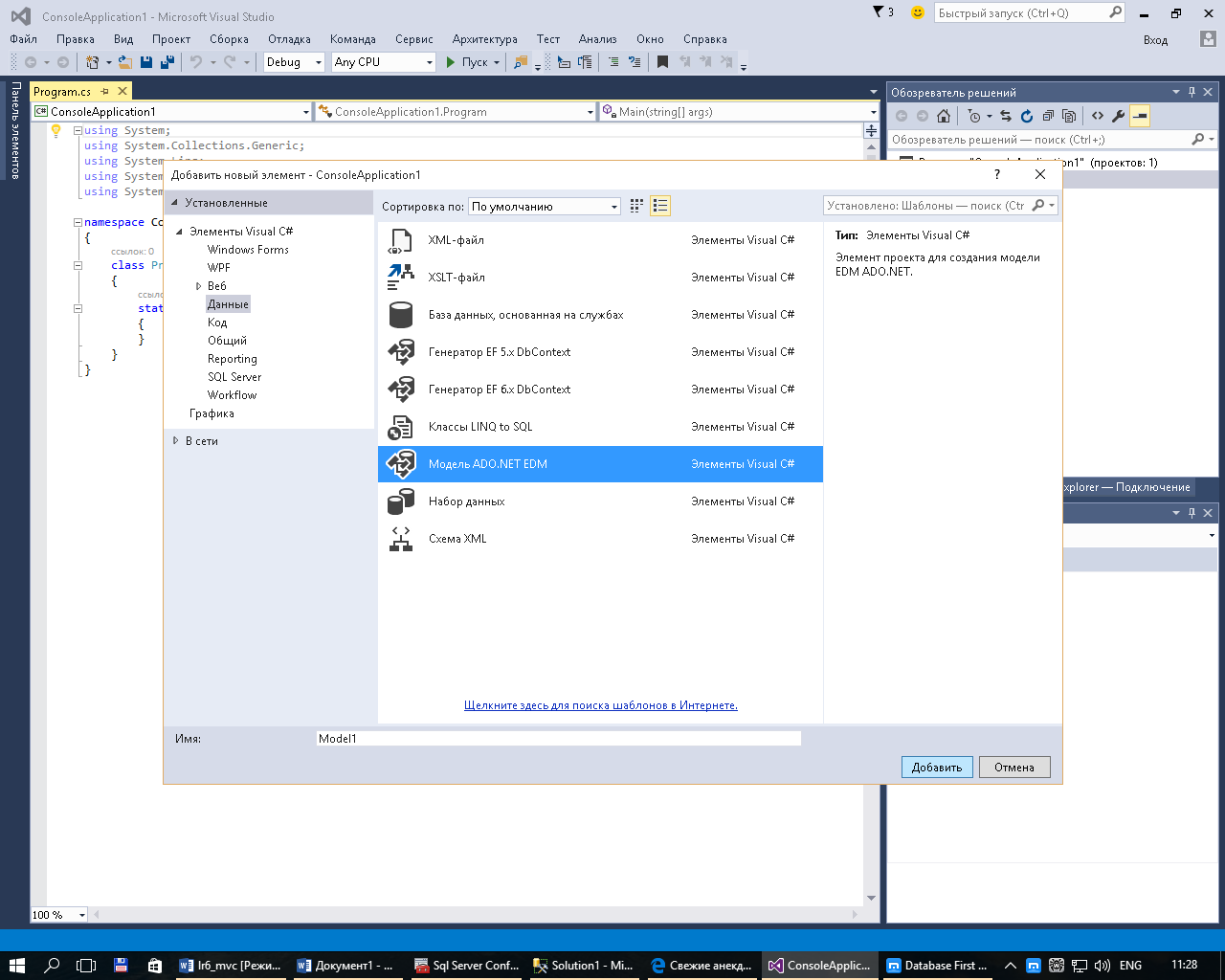
1. Знищимо Project\_readme.html
2. В шаблоні \_Layout.shtm (папка Shared) внесемо наступні зміни:



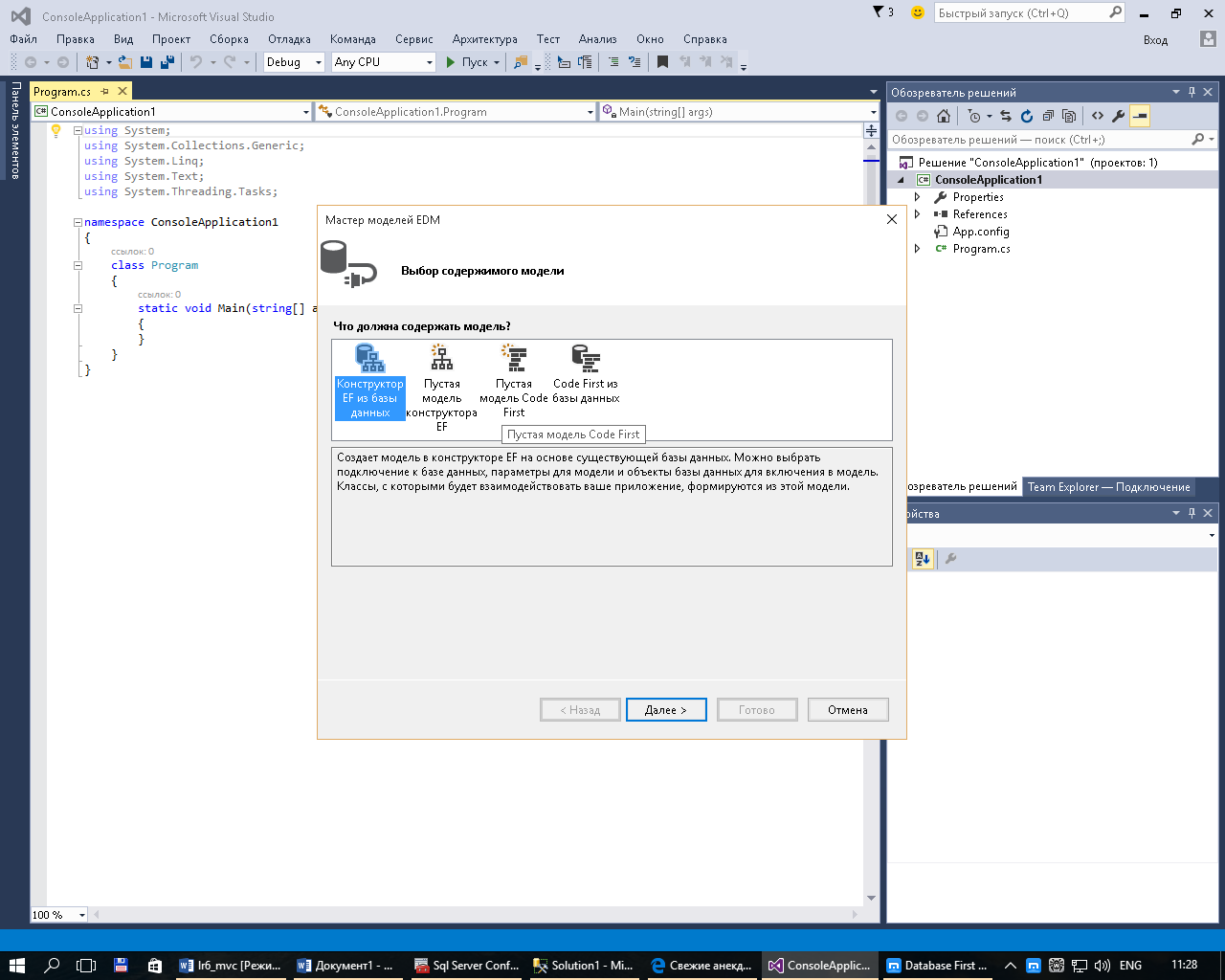
Ім’я проекту змінимо на «DF\_Name\_Group» Name - підставити власне прізвище, та Group - групу. Знищимо зайві посилання.

1. Знищимо з папки View представлення About та Contact.

У Visual Studio в вікні Solution Explorer натиснемо на проект правою кнопкою миші і виберемо в Add -> New Item. Далі у вікні додавання нового елемента виберемо ADO.NET Entity Data Model. Дамо новому компоненту нову назву:

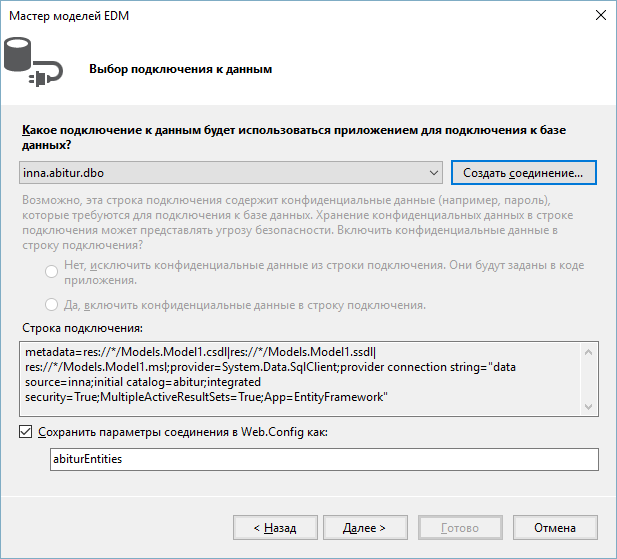


Після цього нам відкриється вікно майстра створення моделі. Якщо ви працюєте з Visual Studio 2013 з пакетом оновлень SP2, SP3, то відкриється наступне вікно майстра моделі:



Тут треба вибрати опцію EF Designer from database. Далі В треба вибрати пункт Generate from database (Конструктор EF з бази даних).

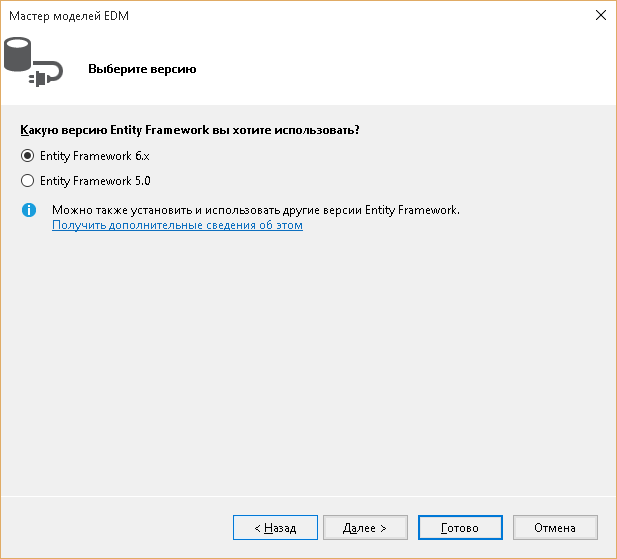
Потім відкриється вікно наступного кроку по створенню моделі, на якому треба буде встановити підключення до бази даних:



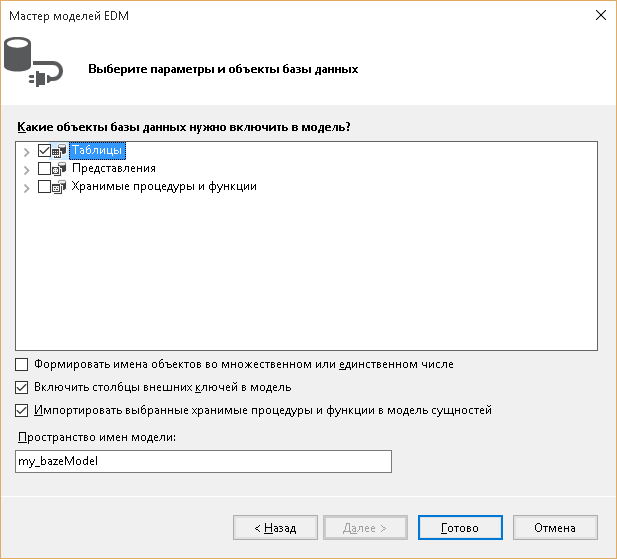
У випадаючому списку виберемо одне з доступних підключень або із використанням кнопки New Connection встановимо нове підключення.

Також внизу вказується назва контексту даних, який буде використовуватися для доступу до даних. За замовчуванням у мене контекст має назву userstoredbEntities. Можна змінити, а можна і залишити.

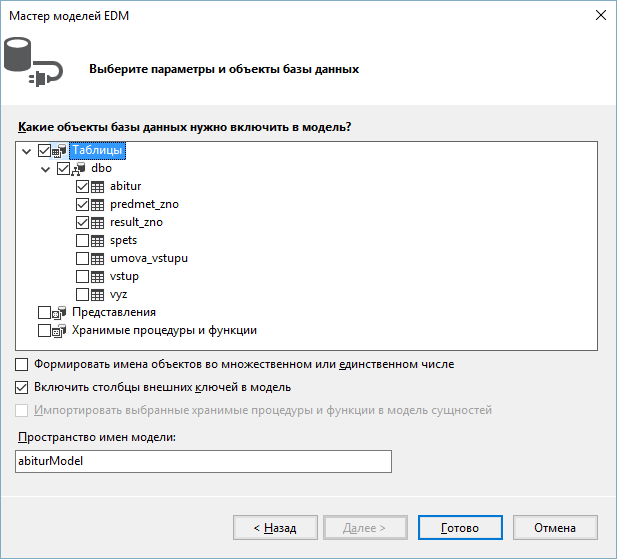
Якщо буде запропоновано вибрати версію Entity Framework доцільно обрати шосту версію:



Далі Visual Studio вилучає всю інформацію про базу даних:

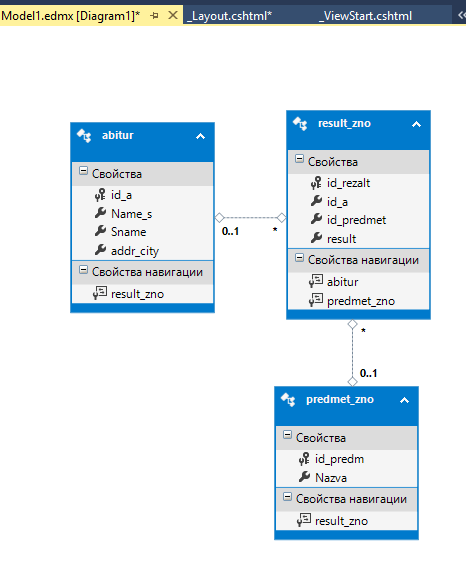


Відкрити вузол Tables. Він відображає всі таблиці, наявні в базі даних. Слід відзначити не менше 3-х таблиць з гілки Tables.

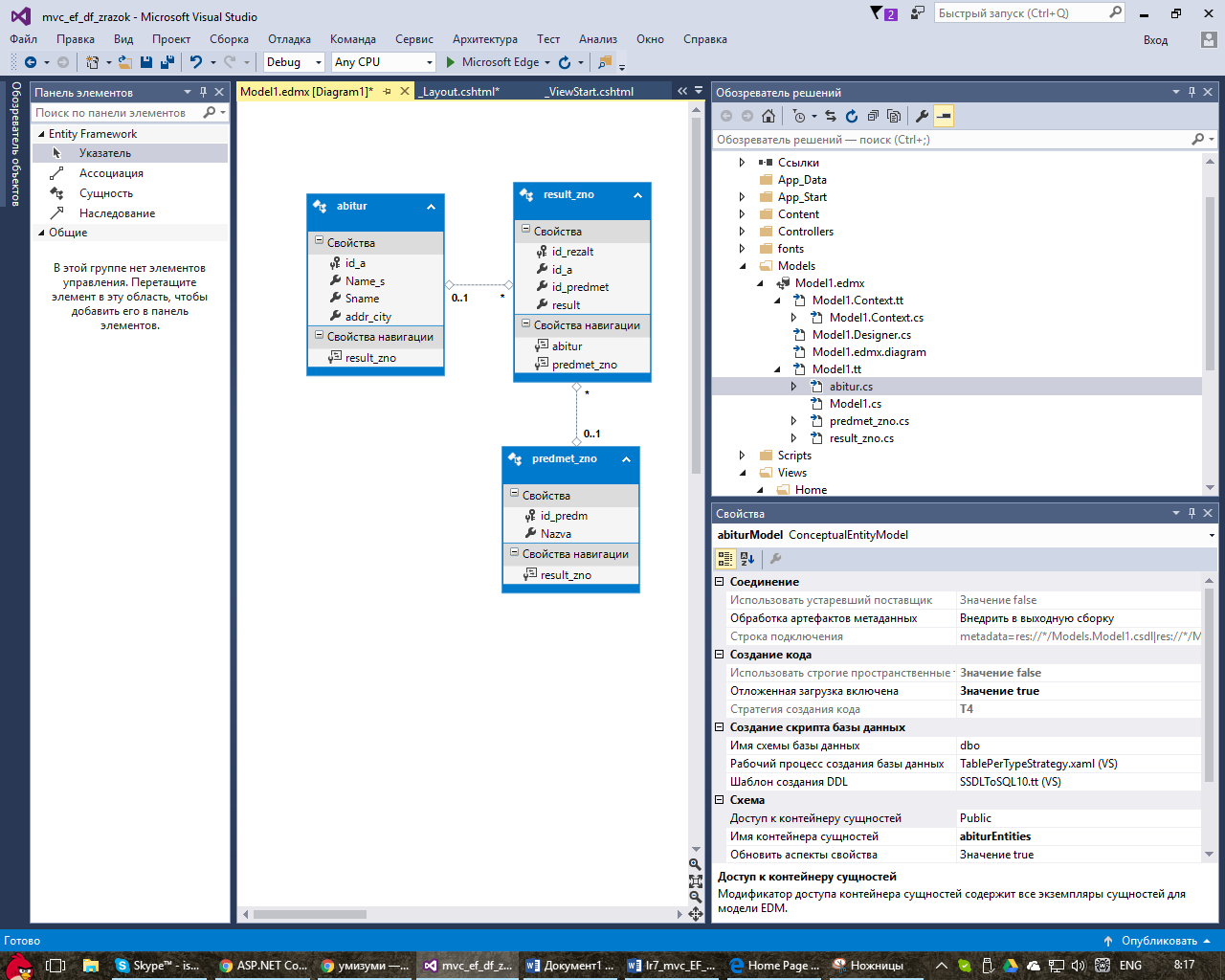


В поле Model Namespace треба вказати ім’я моделі (можна залишити наявне) та натиснути Finish. Після цього Entity Framework згенерує модель по базі даних і додасть її в проект.

Visiual Studio відобразить нам схему моделі:



Якщо виділити сутність у вікні властивостей стають доступними її властивості:



Властивість Name у вікні властивостей вказує на клас, яким буде представлена дана сутність. А властивість Entity Set Name вказує на ім'я набору об'єктів (тобто властивість DbSet контексту даних). Переглянемо вміст згенерованих файлів:

* Model1.context.cs:
* Abitur.cs;
* Predmet,cs;
* Result\_Zno.cs;

*!!!В звіті з лабораторної роботи навести лістинг файлів класів моделі і контексту моделі із коментарями для всіх рядків коду.*

Для тестування із використанням консольних додатків код main() має дуже простий вигляд:

:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | using (my\_bazeEntities db = new abiturEntities())  {  var abitur = db.abitur;  foreach (abitur a in abitur)  Console.WriteLine("{0}.{1} - {2}", a.K\_abitur, a.Name\_s, a.Addr\_City);    } |

Для відображення даних додатку відкриємо у папці Controllers файл HomeController.cs і відредагуємо його вміст наступним чином:

1. Додамо піключення ресурсів:

using System.Data.Entity;

using mvc\_ef\_df\_zrazok.Models;

mvc\_ef\_df\_zrazok – назва проекту ASP.Net MVC, що використовується у зразку (вставити, власну назву проекту)

У клас HomeController : Controller додамо наступну властивість:

abiturEntities db = new abiturEntities();

У подію public ActionResult Index() додамо наступний код:

{

IEnumerable<abitur> Abiturs = db.abitur;

ViewBag.Abiturs = Abiturs;

return View(); }

abiturEntities – власна зава контексту бази даних в моделі за зразком. Замінити на назву, що визначена у Model1.context.cs.

Для відображення даних відредагуємо представлення Index ( з папки Home):

<div class="jumbotron">

<p />

<table>

<tr>

<td>Код абітурієнта</td>

<td>Прізвище</td>

<td>Ім'я</td>

<td>Адреса</td>

</tr>

@foreach (var r in ViewBag.Abiturs)

{

<tr>

<td>@r.id\_a</td>

<td>@r.Name\_s</td>

<td>@r.Sname</td>

<td>@r.addr\_city</td>

<td><a href="/Home/Edit/@r.id\_a">Edit</a></td>

<td><a href="/Home/Delete/@r.id\_a">Delete</a></td>

</tr>

}

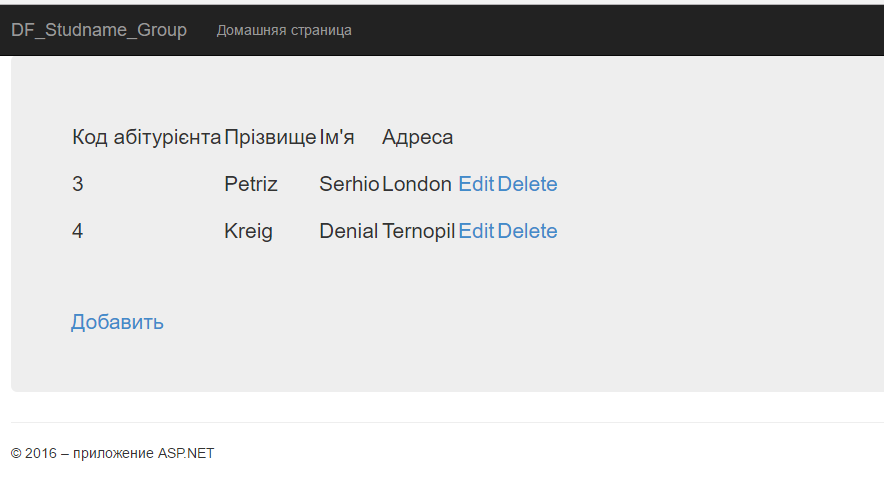
</table>

<br />

<a href="/Home/Insert">Добавить</a>

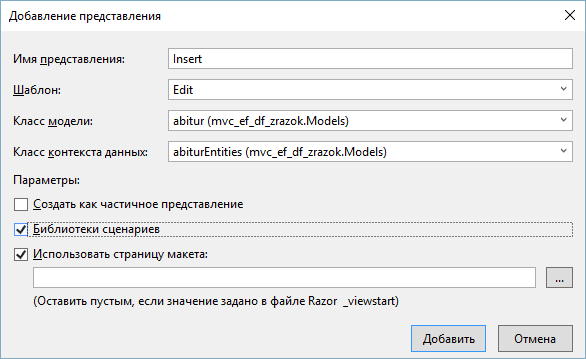
</div>

Результат буде мати приблизно такий вигляд:



Для реалізації операцій модифікації даних створимо додаткове представлення Edit(Insert) та Delete.

Для створення представлення Insert використаємо шаблон:



Далі в HomeControllers внесемо вдповідний код для відображення форми:

[HttpGet]

public ActionResult Insert()

{

return View();

}

Для вставки даних:

[HttpPost]

public string Insert(abitur a) {

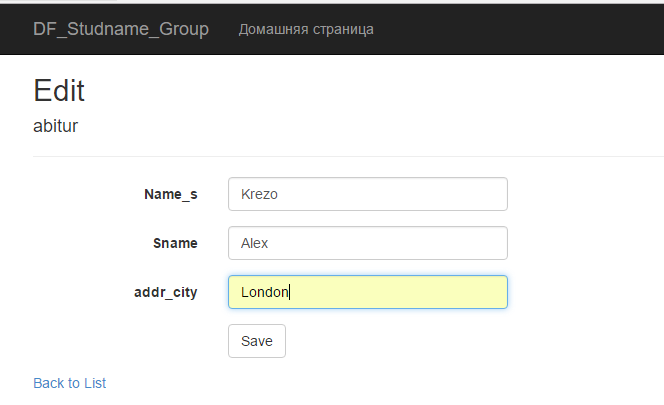
db.abitur.Add(a);

db.SaveChanges();

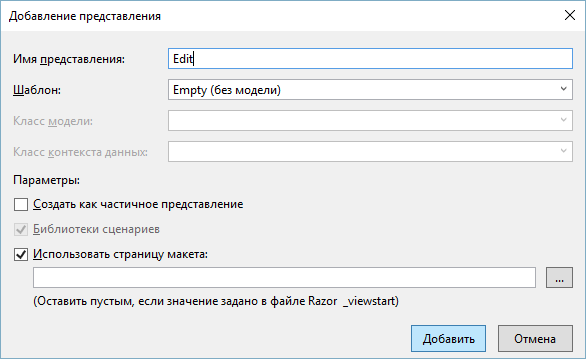
return "Зміни внесено";

}

Результат буде мати такий вигляд:



Для редагування створимо представлення Edit на основі шаблону Empty:



Макет представлення наступний:

odel mvc\_ef\_df\_zrazok.Models.abitur

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

@{

ViewBag.Title = "Edit";

}

<html>

<head>

<h2>Edit</h2>

</head>

<body>

<div>

@using (Html.BeginForm("Edit", "Home", FormMethod.Post))

{

// <input type="hidden" value="@ViewBag.id\_a" name="idgr" />

@Html.Hidden("id\_a", @ViewBag.id\_a as string);

<br />

@Html.LabelFor(m => m.Name\_s, "Прізвище");

<br />

<input type="text" value="@ViewBag.Name\_s" name="Name\_s">

<br />

@Html.LabelFor(m => m.Sname, "I'мя");

<br />

<input type="text" value="@ViewBag.Sname" name="SName">

<br />

@Html.LabelFor(m => m.addr\_city, "Місто");

<br />

<input type="text" value="@ViewBag.addr\_city" name="addr\_city">

<br />

<br />

<p><input type="submit" value="Зберегти" /></p>

@Html.ActionLink("Back", "Index")

}

</div>

</body>

</html>

Обробник події метода Get:

[HttpGet]

public ActionResult Edit(int id)

{

abitur a = db.abitur.Find(id);

ViewBag.id\_a = a.id\_a;

ViewBag.Name\_s = a.Name\_s;

ViewBag.Sname = a.Sname;

ViewBag.addr\_city = a.addr\_city;

return View();

}

В коді дописуємо обробник метода Post:

public string Edit(abitur a)

{

abitur ins\_A = db.abitur.Find(a.id\_a);

ins\_A.Name\_s = a.Name\_s;

ins\_A.Sname = a.Sname;

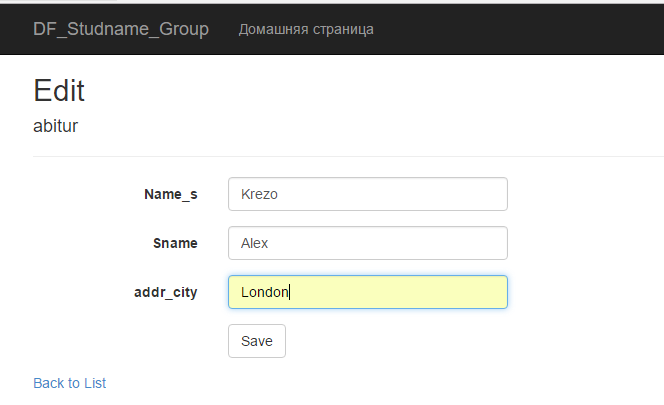
ins\_A.addr\_city = a.addr\_city;

db.SaveChanges();

return "Зміни внесено";

}

Результат наведено на рисунку:



Для знищення записів реалізуємо метод Delete. Додамо обробник події метода Get, що має наступний вид:

[HttpGet]

public ActionResult Delete(int id)

{

abitur a = db.abitur.Find(id);

if (FormMethod.Get == 0)

{

ViewBag.abitur = a;

return View("Delete");

}

}

В свою чергу метод Post забезпечує саме знищення даних після підтвердження:

[HttpPost]

public ActionResult Delete(abitur a)

{

db.abitur.Remove(db.abitur.Find(a.id\_a));

db.SaveChanges();

IEnumerable<abitur> Abiturs = db.abitur;

ViewBag.abiturs = Abiturs;

return View("Index");

}

Макет представлення реалізуємо наступним чином:

@{

ViewBag.Title = "Delete";

}

<h2>Delete</h2>

<div>

@using (Html.BeginForm("Delete", "Home", FormMethod.Post))

{

var a = ViewBag.abitur;

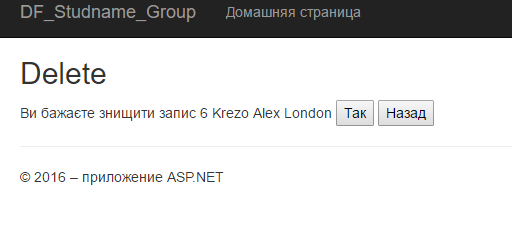
@Html.Raw("Ви бажаєте знищити запис" + @a.id\_a + " " + @a.Name\_s + " " + a.Sname + " " + a.addr\_city);

<input type="hidden" value="@a.id\_a" name="id\_a" />

<input type="submit" value="Так" />

<input type="button" onclick="history.back();" value="Назад" />

}</div>

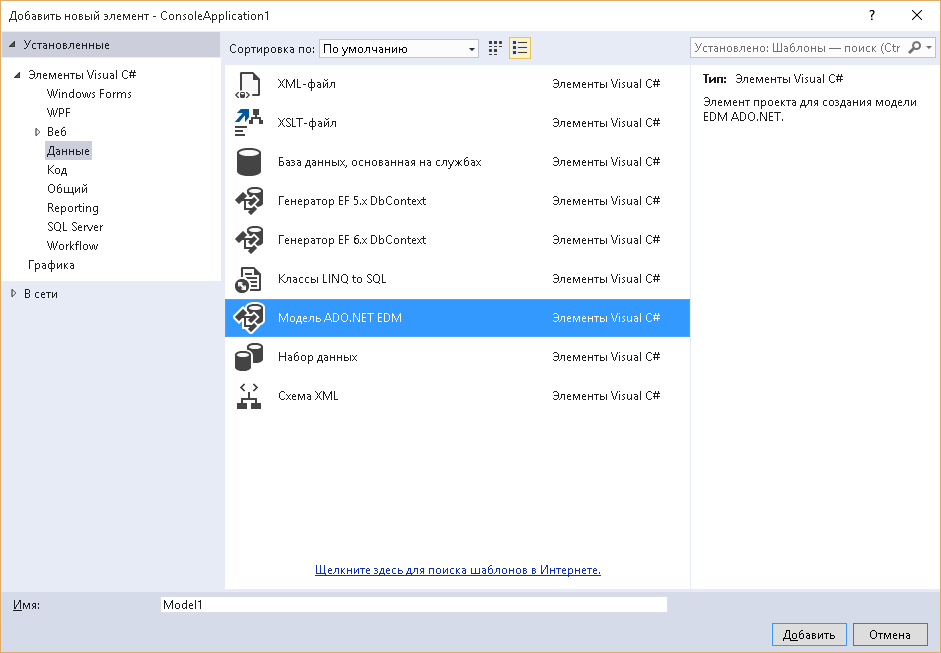


Далі необхідно самостійно реалізувати інтерфейси для управління даними у інших двох таблицях. По можливості врахувати зв’язки між таблицями.

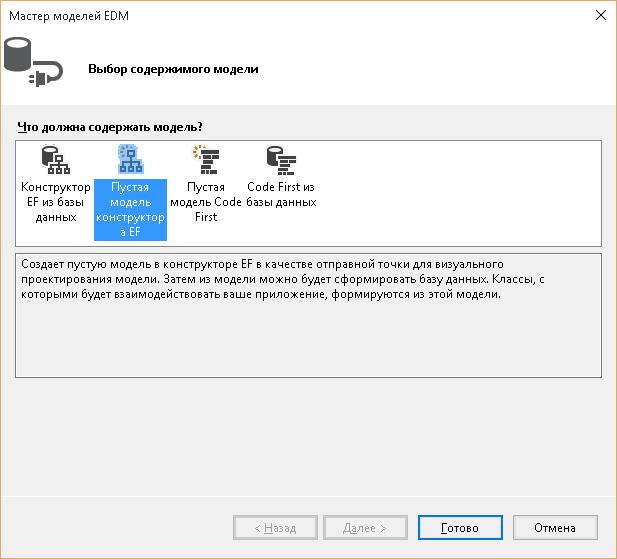
**Model First**

Створимо новий додаток-клієнт для БД з використанням підходу Model First.

Для виконання завдання необхідно створити новый проект по типу ASP .Net MVC. Далі необхідно додати в проект новий елемент ADO.NET Entity Data Model:



Оскільки модель буде описувати замовлення, то назвемо її Zakaz. Натиснемо OK і нам відкриється майстер створення моделі. Якщо у нас Visual Studio з пакетами оновлення SP2, SP3, то майстер створення моделі виглядає наступним чином:

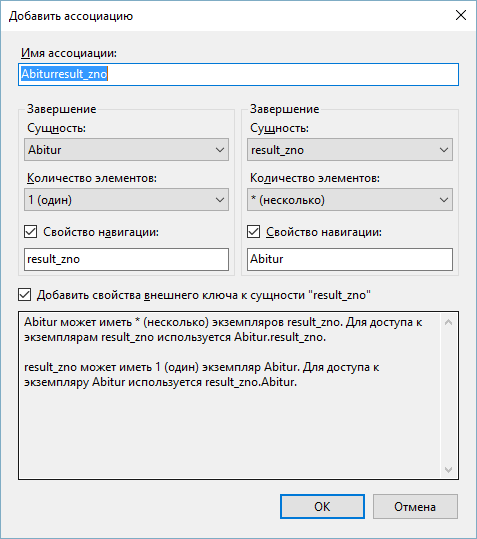


Вікно нам пропонує чотири варіанти створення моделі, з яких нам треба вибрати Empty EF Designer Model. Натиснемо кнопку Finish, і перед нами відкриється порожнє вікно створення моделі.

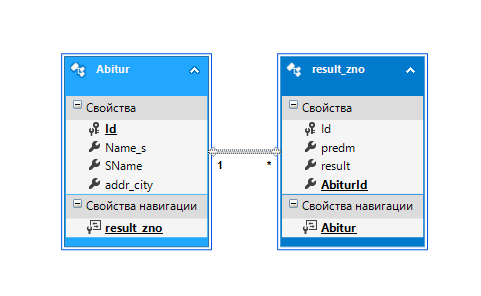
Перетягнемо на це поле з панелі Toolbox (Панель Інструментів) в лівій частині елемент Entity. Тепер у нас на поле створення моделі є невелика схема майбутньої моделі, в якій зараз за замовчуванням вказано лише одне поле - Id. Для сутності необхідно змінити назву.

Далі слід додати властивості (Add New -> Scalar Property) та визначити їх тип. Аналогічним чином створюємо ще дві таблиці.

Після цього потрібно буде додати зв'язок між цими таблицями. Це можна зробити за допомогою елемента Association з панелі Toolbox або клацнувши правою кнопкою миші по таблиці і вибравши в контекстному меню Add New -> Association. Давайте використаємо другий підхід. При цьому відкриється модальне вікно Add Association:

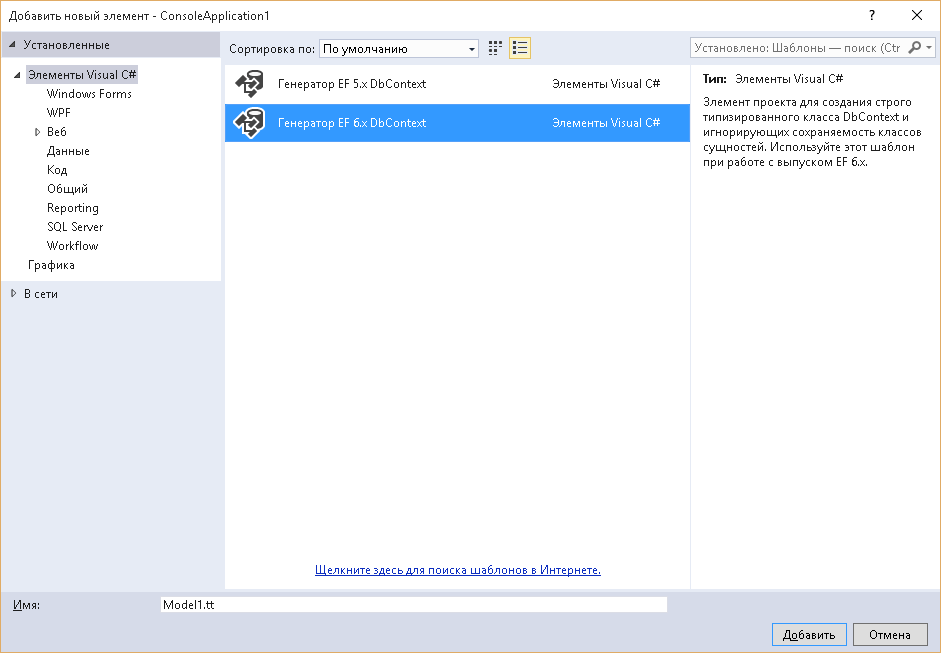


Таким чином, побудовано наступну схему:

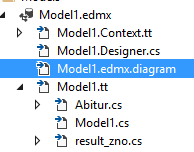


Після створення діаграми моделі слід виконати перебудову проекту за допомогою опції Rebuild.

Тепер потрібно згенерувати код і базу даних. Спочатку сгенерируем код моделі. Для цього натиснемо на діаграму моделі правою кнопкою миші і виберемо пункт Add Code Generation Item:



Слід обрати Генератор 6.х DbContext. Після цього в структурі проекту можна побачити вузол Model1.tt, який класи моделі у файлах \*.cs (назви файлів співпадають із сутностями моделі).



Також тут ми можемо знайти файл контексту даних Model1.Context.cs, який виглядає у прикладі наступним чином:

namespace mf\_sugonyak\_2016.Models

{

using System;

using System.Data.Entity;

using System.Data.Entity.Infrastructure;

public partial class Model1Container : DbContext

{

public Model1Container()

: base("name=Model1Container")

{

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

throw new UnintentionalCodeFirstException();

}

public virtual DbSet<Abitur> AbiturSet { get; set; }

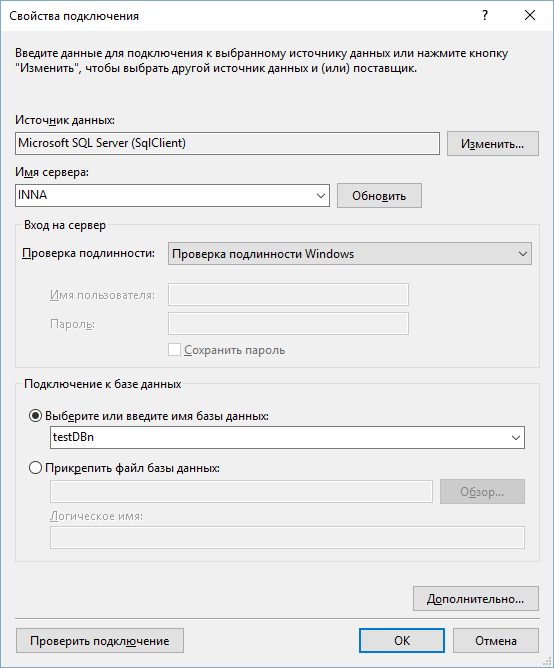
public virtual DbSet<result\_zno> result\_znoSet { get; set; }

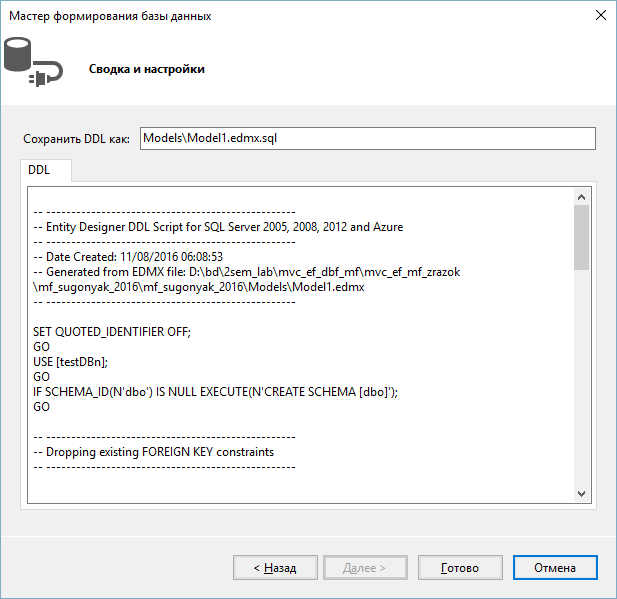
}

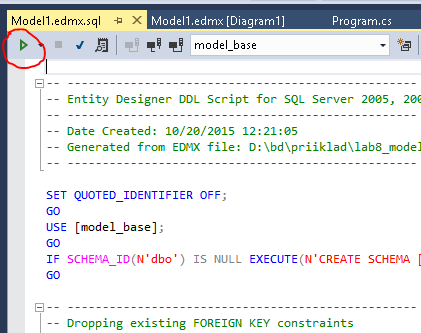
}

Необхідно згенерувати відповідну базу даних. Для цього в контекстному меню слід обрати Generate Database from Model (Згенерувати базу даних по моделі).

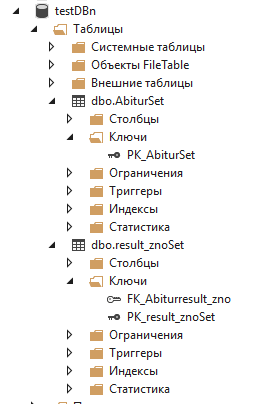
За допомогою опції New Connection (Нове підключення) слід налаштувати підключення і створити базу даних. Тут нам треба ввести ім'я сервера та назву БД.



Visual Studio встановить в якості підключення моделі щойно створену базу даних. Після цього буде згенеровано скрипт бази даних: 

Натиснемо Finish (Готово). Файл скрипта автоматично відкриється в Visual Studio. І в завершенні нам треба буде запустити цей скрипт. Для цього натиснемо в верхньому лівому кутку на зелену кнопку Execute (Виконати): 

Після цього в нижньому вікні Visual Studio нам повідомляє про успішне (або неуспішне) створення бази даних.



Для тестування можна створити консольний додаток з мінімальним кодом:

static void Main(string[] args)

{

using (Model1Container db = new Model1Container())

{

// добавление элементов

db.abitur.Add(new abitur { Name\_s = "Tom", Sname = 'Jons' });

db.Klient.Add(new Klient { Name\_s = "John", Same = 'Smoth' });

db.SaveChanges();

// получение элементов

var Abiturs = db.abitur;

foreach (abitur k in Abiturs)

Console.WriteLine("{0}.{1} - {2}", k.Id, k.SName, k.Name\_s);

}

Console.Read();

та створити додаток ASP.Net MVC для управління даними аналогічно із попереднім завданням.

**Вимоги до звіту:**

Звіт з лабораторної роботи містить:

1. Тема та мета роботи
2. Лістинг файлів моделі \*.сs згенерований за підходу DBFirst.
3. Лістинг Home Controller.cs та інших контролерів (якщо вони наявні)
4. Лістинг фрагментів коду представлень (тільки якщо код створений власноруч)
5. Скріншоти, представлень робочого додатку
6. Діаграму моделі, Діаграму БД + скріпт згенерованої за підходом ModelFirst.
7. Лістинг файлів моделі \*.сs
8. Скріншоти, представлень робочого додатку

# Лабораторна робота № 6. РЕАЛІЗАЦІЯ БД ТА ІНТЕРФЕСІВ ДЛЯ WEB-ОРІЄНТОВАНИХ СИСТЕМ (PHP+MYSQL)

**Завдання на лабораторну роботу**

1. **Реалізувати форму доступу до даних за допомогою API pdo (перегляд, вставка, знищення даних.)**
2. **Реалізувати основні запити до бази даних.**

**Запити згідно індивідуального варіанту.**

1. **Реалізувати виклики процедур, що зберігаються**

**Вимоги до звітності.**

1. **Лістинг процедур, що зберігаються.**
2. **Лістинг та screenshots форми.**

Теоретичні відомості

[Процедури, що зберігаються в MySql](http://ruseller.com/lessons.php?id=1189)

PDO Соединение с БД

//пример соединения с MySQL при помощи PDO

$db = new PDO('mysql:host=localhost;dbname=test', $user, $pass);

Как только будет установлено успешное соединение с конкретным сервером базы данных, то будет возвращен объект PDO. Этот объект позволяет выполнять самые разнообразные задачи баз данных.

Если есть какие-либо ошибки подключения, то сработает механизм исключений – PDOException. Нужно всегда оборачивать операции PDO в блок try/catch. Вы можете поймать исключение, если вы хотите обрабатывать ошибки, или вы можете оставить его для глобального обработчика исключений (exception), которые вы создали с помощью set\_exception\_handler (). В PDO есть специальные функции для ошибок: errorCode() – вернет номер ошибки, errorInfo() – вернет массив с номером ошибки и описанием. Они нужны так как по умолчанию режим работы с ошибками стоит ERRMODE\_SILENT. В этом случае чтобы увидеть эти ошибки придется их вызвать:

echo $conn->errorCode();

echo $conn->errorInfo();

Чтобы этого не делать, в режиме разработке проще сразу выставить нужный режим работы с ошибками: ATTR\_ERRMODE и ERRMODE\_EXCEPTION. Можно также прописать кодировку работы с базой. В итоге у нас получится такой вот код подключения:

try {

$db = new PDO("mysql:host=$host;dbname=$dbname", $user, $password);

$db->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

$db->exec("set names utf8");

}

catch(PDOException $e) {

echo $e->getMessage();

}

После успешного подключения к базе данных, экземпляр класса PDO возвращается в сценарий. $db содержит дескриптор базы данных. Соединение остается активным в течение всей жизни объекта PDO. Чтобы закрыть соединение, вам нужно уничтожить объект с гарантией, что все остальные ссылки на него будут удалены. Сделать это можно путем присвоения переменной, которая содержит объект, значения NULL. Если вы не сделаете этого явно, PHP будет автоматически закрывать соединение после завершения работы скрипта.

//Закрытие соединения

$db = null;

Многие веб-приложения выиграют от создания постоянных подключений к серверам баз данных. Постоянные соединения не закрываются в конце скрипта, а кэшируются и повторно используются, когда другой сценарий запрашивает соединение, используя те же учетные данные. Постоянное соединение позволяет уменьшить ресурсы на создание нового соединения каждый раз, когда сценарий должен обратиться к базе данных, что приводит к более быстрой работе веб-приложения.

//Постоянное соединение

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass, array(

PDO::ATTR\_PERSISTENT => true

));

Теперь, когда вы увидели, как открывать и закрывать соединение, давайте рассмотрим другие примеры работы с PDO. В данном случае я собираюсь показать вам, как выполнять запросы к конкретной базе данных. Запросы можно делать 3 функциями: exec(), query() и prepare+execute.

Exec()

Первый – exec вернет только кол-во задействованных строк или FALSE при ошибке и используется там, где не возвращаются данные, например при удалении:

//использование метода exec()

try{

$db = new PDO('mysql:host=localhost;dbname=test','user','password');

$delrows=$db->exec('DELETE FROM users WHERE id>20');

echo 'Количество удаленных строк: '.$delrows;

}

catch(PDOException $e){

echo 'Error : '.$e->getMessage();

exit();

}

//Еще

$db->exec('DELETE FROM folks WHERE 1');

//или

$db->exec("SET time\_zone = '-8:00'");

//или

$db->exec("CREATE TABLE `test`(

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(20) NOT NULL DEFAULT '',

email VARCHAR(50) NOT NULL DEFAULT '')");

//или

$db->exec("SET CHARACTER SET utf8");

Query()

Второй – query() вернет результат в объекте PDOStatement. Также возвращает результат или FALSE при ошибке. Ему можно доверить простые запросы. Можно использовать query() с условием (я правда не знаю зачем кому то это может понадобиться), но тогда все равно придется экранировать данные методом PDO::quote

//Простые запросы

$db->query("SET CHARACTER SET utf8");

$db->query("SELECT \* FROM users");

//Можно вычислить количество строк

$stmt = $db->query('SELECT \* FROM table');

$row\_count = $stmt->rowCount();

echo $row\_count.' rows selected';

//Еще вариант с количеством

$stmt = $db->query('SELECT \* from users');

$rows = $stmt->fetchAll();

$count = count($rows);

foreach($rows as $row)

{

print\_r($row);

}

//Запрос с условием и экранированием

$conn->query('SELECT \* FROM table WHERE id = ' . $conn->quote($id));

**lastInsertId()** возвращает идентификатор последней вставленной строки базы данных.

//метод lastInsertId() возвращает id последней записи

$db->query("INSERT INTO users SET name='Vasya',address='Here',email='vasya@test.com'");

$insertId=$db->lastInsertId();

Подготовленные выражения – prepared statments.

Третий способ**-**prepare+execute - **Подготовленные выражения**, они же подготовленные инструкции они же плейсхолдеры они же **prepared statments**или связываемые переменные, позволяют определить выражение один раз, а затем многократно его выполнять с разными параметрами. Также они позволяют отделить переменные от запроса, что делает код безопаснее и повышает скорость выполнения. В вашем коде больше не надо будет пытаться очистить передаваемые данные. Мы сделаем это только один раз перед выполнением запроса к БД. Для этого используем функцию **Prepare()**; В качестве параметра она принимает SQL запрос, но в нем, вместо переменных используются метки, в виде знака вопроса ‘?’ или номеров ‘:1′, или переменой, имя которой начинается с двоеточия ‘:’. Если вы остановились на знаках вопроса (:цифрах), то вам надо в функцию execute передать массив значений, в соответствующей последовательности. Если ваш выбор именованные переменные, то надо назначить каждой переменной значение через одну из двух функций: либо bindValue(), которая присваивает псевдо-переменной значение, либо bindParam(), которая связывает псевдо-переменную с настоящей переменной. Третьим параметром можно указать тип переменной, например $db->bindParam(‘:id’,$id, PDO::PARAM\_INT).

//Не именованные метки

try {

$stmt = $db->prepare("INSERT INTO test (label,color) VALUES (?,?)");

$stmt -> execute(array('perfect','green'));

}

catch(PDOException $e){

echo 'Error : '.$e->getMessage();

exit();

}

//stmt - это "дескриптор состояния"

//Именованные метки

$stmt = $db->prepare("INSERT INTO test (label,color) VALUES (:label,:color)");

$stmt -> execute(array('label'=>'perfect', 'color'=>'green'));

//Еще вариант

$stmt = $db->prepare("INSERT INTO users (firstname, lastname, email) VALUES (:firstname, :lastname, :email)");

$stmt->bindParam(':firstname', $firstname);

$stmt->bindParam(':lastname', $lastname);

$stmt->bindParam(':email', $email);

$firstname = "John";

$lastname = "Smith";

$email = "john@test.com";

$stmt->execute();

Еще раз напомню, что если вы не используете подготовленные выражения но все-таки хотите обезопасить передаваемые данные, то это можно сделать при помощи функции PDO:quote.

PDO SELECT данных

Для выборки данных используются методы fetch() или fetchAll(). Перед вызовом функции нужно указать PDO как Вы будете доставать данные из базы. PDO::FETCH\_ASSOC вернет строки в виде ассоциативного массива с именами полей в качестве ключей. PDO::FETCH\_NUM вернет строки в виде числового массива. По умолчанию выборка происходит с PDO::FETCH\_BOTH, который дублирует данные как с численными так и с ассоциативными ключами, поэтому рекомендуется указать один способ, чтобы не иметь дублирующих массивов:

$stmt = $db->query('SELECT \* from users');

//Установка fetch mode

$stmt->setFetchMode(PDO::FETCH\_ASSOC);

while($row = $stmt->fetch())

{

echo "<p>" . $row['firstname'] . "&nbsp;" . $row['lastname'] . "</p>";

echo "<p>" . $row['email'] . "</p><br />";

}

Либо можно указать метод выборки в самом методе ->fetch()

$stmt = $db->query('SELECT \* FROM table');

while($row = $stmt->fetch(PDO::FETCH\_ASSOC)) {

echo $row['field1'].' '.$row['field2']; //и т.д...

}

//Внимание, не работающий пример с LIKE!

$stmt = $db->prepare("SELECT field FROM table WHERE field LIKE %?%");

$stmt->bindParam(1, $search, PDO::PARAM\_STR);

$stmt->execute();

//Нужно так:

$stmt = $db->prepare("SELECT field FROM table WHERE field LIKE ?");

$stmt->bindValue(1, "%$search%", PDO::PARAM\_STR);

$stmt->execute();

//Еще пример

$stmt = $db->prepare("SELECT \* FROM table WHERE id=? AND name=?");

$stmt->bindValue(1, $id, PDO::PARAM\_INT);

$stmt->bindValue(2, $name, PDO::PARAM\_STR);

$stmt->execute();

$rows = $stmt->fetchAll(PDO::FETCH\_ASSOC);

//Еще вариант

$stmt = $db->prepare('SELECT \* from users');

$stmt -> execute();

while($row = $stmt->fetch()) {

print\_r($row);

}

PDO UPDATE данных

Происходит по существу также как и INSERT и SELECT (в данном случае опять же будем использовать именованные метки (placeholders)):

$stmt = $db->prepare("UPDATE users set email = :email where lastname=:lastname");

$stmt->bindParam(':lastname', $lastname);

$stmt->bindParam(':email', $email);

$lastname = "Smith";

$email = "newmail@test.com";

$stmt->execute();

**DELETE** Удаление происходит простейшим образом:

$db->exec("DELETE FROM users");

Конечно вы можете также использовать именованные параметры (placeholders) при удалении.

Если у вас есть свои часто используемые заготовки при работе с расширением PHP PDO, буду

PDO Хранимые процедуры

Если драйвер базы данных поддерживает, то запрос может также связывать IN и OUT параметры. Параметры OUT обычно используются, чтобы получить данные из хранимых процедур. Выходные параметры OUT чуть более сложнее, в использовании чем входные параметры IN. Разработчик должен знать, насколько большое значение может принять данный параметр. Если значение, оказывается, больше чем указано, произойдёт ошибка.

Вызов хранимой процедуры с параметром OUT:

$stmt = $dbh->prepare( 'CALL sp\_returns\_string(?)' );

// Здесь четвёртым параметром указывается макс. величина, которую

// может принять параметр:

$stmt->bindParam(1, $return\_value, PDO::PARAM\_STR, 4000);

// Выполнение хранимой процедуры:

$stmt->execute();

echo "procedure returned $return\_value\n";

Разработчики могут также определить параметры, которые являются INOUT. Синтаксис подобен, привязке IN и OUT параметров. В следующем примере строку «привет» мы передадим в хранимую процедуру параметром INOUT, и когда он же вернётся, значение «привет» заменится возвращаемым значением процедуры:

Вызов хранимой процедуры с параметром INOUT:

$stmt = $dbh->prepare("CALL sp\_takes\_string\_returns\_string(?)");

$value = "hello";

$stmt->bindParam(1, $value, PDO::PARAM\_STR|PDO::PARAM\_INPUT\_OUTPUT, 4000);

// Выполнение хранимой процедуры:

$stmt->execute();

print "procedure returned".$value;

Не правильное использование плейсхолдера:

//Так нельзя:

$stmt = $dbh->prepare("SELECT \* FROM REGISTRY where name LIKE '%?%' ");

$stmt->execute(array($\_GET['name']));

// Так правильно:

$stmt = $dbh->prepare('SELECT \* FROM REGISTRY where name LIKE ?');

$stmt->execute(array('%$\_GET[name]%'));

Index.php

Insert.php



# Лабораторна робота № 7. РАБОТА З NOSQL MONGDB

**Задание**

1. **Реализовать БД за індивідуальним варіантом**
2. **Встановити між колекціями зв’язки із використанням вкладених документів та DBRef**
3. **Реалізувати основні операції над даними**
4. **Реалізувати 4 запити на вибірку даних (з одним параметром.ю з декылькома параметрами.ю з використанням спеціальних операторів, з використанням операторів сортування, ліміту та зсуву)**

[**http://jsman.ru/mongo-book/**](http://jsman.ru/mongo-book/)**+ презентация**

**ХІД РОБОТИ**

**Шаг 0 — Установка**

Давайте скачаем её с офф. сайта [тут](http://www.mongodb.org/downloads).

**Важно!**  
Тем у кого Windows Server 2008 R2 или Windows 7, необходимо установить hotfix, чтобы устранить проблему с памятью.  
hotfix можно скачать [тут](http://support.microsoft.com/kb/2731284).

После того как вы скачали и установили Hotfix вам нужно перезагрузить ПК для того, чтобы обновление вступили в силу.

Скачанный сервер MongoDB установите в директорию С:\mongodb

Так же важно знать, что MongoDB не зависит от этой папки и вы можете установить его куда угодно в любую папку.

**Шаг 1 — Конфигурирование MongoDB**

MongoDB требует наличие папки для хранения своих файлов.

Давайте создадим две папки в C:/mongodb/**data**и C:/mongodb/**log**

После этого создайте и откройте на редактирование файл C:/mongodb/**mongo.config**:

[?](http://devcolibri.com/2091)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ##store data  dbpath=C:\mongodb\data    ##Log File  logpath=C:\mongodb\log\mongo.log    ##log read and write operations  diaglog=3 |

**Шаг 2 — Старт MongoDB**

В консоли **cmd** выполним следующую команду для старта **mongodb**:

[?](http://devcolibri.com/2091)

|  |  |
| --- | --- |
| 1 | mongod.exe --config c:\mongodb\mongo.config |

В результате вы должны увидеть в консоли следующее:

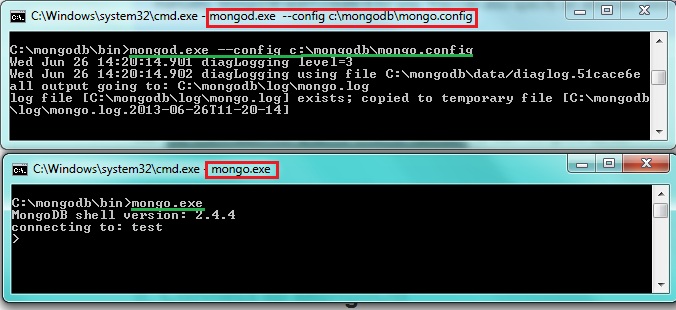
http://devcolibri.com/cp/wp-content/uploads/2013/06/2091_2.jpg

Не закрывая эту консоль откройте новую и выполните следующую команду:

[?](http://devcolibri.com/2091)

|  |  |
| --- | --- |
| 1 | c:/mongodb/bin/mongo.exe |

У вас должно получится следующее:



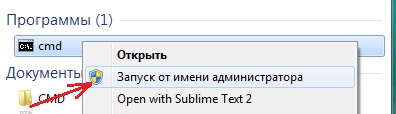
Но этот способ плох тем, что у вас постоянно должна быть открыта первая консоль которая запустила **MongoDB**.

**Шаг 3 — Добавляем MongoDB в Windows Service**

Как это исправить? Легко, для этого нужно добавить запуск mongodb в сервис.

Давайте добавим **MongoDB** в **Windows Service**.

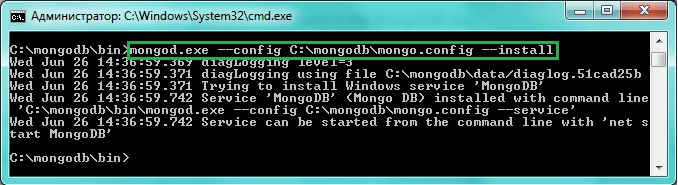
Для этого запустите cmd консоль от имени Администратора:



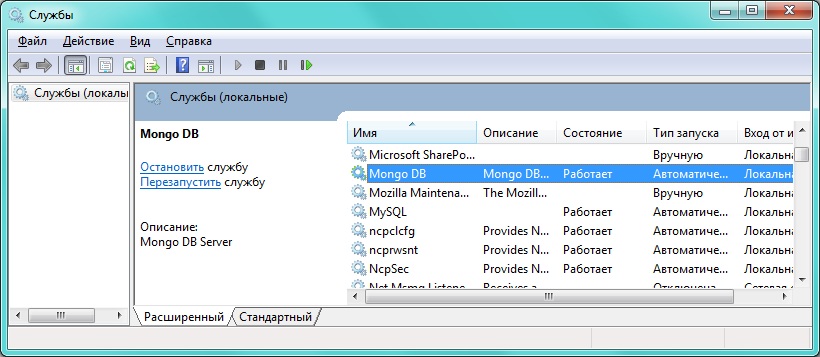
Дальше выполняем следующую команду:

[?](http://devcolibri.com/2091)

|  |  |
| --- | --- |
| 1 | mongod.exe --config C:\mongodb\mongo.config --install |



После этого вы можете зайти в **Панель Управления**(Control Panel) -> **Службы**(Services) и там вы увидите добавленный нами сервис.



Что он дает? Он запускает сервер автоматически при загрузки ОС.

Все, теперь вы можете работать с БД MongoDB.

**Ресурс:** [http://docs.mongodb.org/](http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/)

### Установка драйверов MongoDB

Конечно, мы можем работать и через консоль mongo, добавляя и отображая объекты в бд. Но нам также было бы неплохо, если бы mongoDB взаимодействовала бы с нашими приложениями, написанными на PHP, C++, C# и других языках программирования. И для этой цели нам потребуются специальные драйверы.

На офсайте на странице <http://docs.mongodb.org/ecosystem/drivers/> можно найти драйвера для таких языков программирования, как PHP, C++, C#, Java, Python, Perl, Ruby, Scala и др.

Далее уже, рассматривая взаимодействие отдельных языков программирования с MongoDB, мы подробнее рассмотрим установку и драйвера и всю необходимую конфигурацию для определенных языков программирования.

Рекомендації до виконання роботи

Запустите сервер mongod и консоль mongo, если ещё не запустили. Консоль работает на JavaScript. Есть несколько глобальных команд, например help или exit. Команды, которые вы запускаете применительно к текущей базе данных исполняются у объекта db, например db.help() или db.stats() . Команды, которые вы запускаете применительно к конкретной коллекции, исполняются у объектаdb.ИМЯ\_КОЛЛЕКЦИИ, например db.unicorns.help() или db.unicorns.count() .

Введите db.help() и получите список команд, которые можно выполнить у объекта db.

Заметка на полях. Поскольку консоль интерпретирует JavaScript, если вы попытаетесь выполнить метод без скобок, то в ответ получите тело метода, но он не выполнится. Не удивляйтесь, увидев function (...){, если случайно сделаете так. Например, если введёте db.help (без скобок), вы увидите внутреннее представление метода help.

Сперва для выбора базы данных воспользуемся глобальным методом use — введите use learn. Неважно, что база данных пока ещё не существует. В момент создания первой коллекции создастся база данных learn. Теперь, когда вы внутри базы данных, можно вызывать у неё команды, например db.getCollectionNames(). В ответ увидите пустой массив ([ ]). Поскольку коллекции бесструктурны *(в оригинале «schema-less». Здесь и далее — прим. перев.)*, мы не обязаны создавать их явно. Мы просто можем вставить документ в новую коллекцию. Чтобы это сделать, используйте команду insert, передав ей вставляемый документ:

db.unicorns.insert({name: 'Aurora', gender: 'f', weight: 450})

Данная строка выполняет метод insert («вставить») в коллекцию unicorns, передавая ему единственный аргумент. MongoDB у себя внутри использует бинарный сериализированный JSON формат. Снаружи это означает, что мы широко используем JSON, как, например, в случае с нашими параметрами. Если теперь выполнить db.getCollectionNames(), мы увидим две коллекции: unicorns и system.indexes.system.indexes создаётся в каждой базе данных и содержит в себе информацию об индексах этой базы.

Теперь у коллекции unicorns можно вызвать метод find, который вернёт список документов:

db.unicorns.find()

Заметьте, что кроме данных, которые мы задавали, появилось дополнительное поле \_id . Каждый документ должен иметь уникальное поле\_id. Можете генерировать его сами или позволить MongoDB самой сгенерировать для вас ObjectId. В большинстве случаев вы скорее всего возложите эту задачу на MongoDB. По умолчанию \_id — индексируемое поле, вследствие чего и создается коллекция system.indexes. Давайте взглянем на system.indexes:

db.system.indexes.find()

Вы увидите имя индекса, базы данных и коллекции, для которой индекс был создан, а также полей, которые включены в него.

Вернёмся к обсуждению бесструктурных коллекций. Давайте вставим кардинально отличный от предыдущего документ в unicorns, вот такой:

db.unicorns.insert({name: 'Leto', gender: 'm', home: 'Arrakeen', worm: false})

И снова воспользуемся find для просмотра списка документов. Теперь, узнав чуть больше, мы можем обсудить это интересное поведение MongoDB, но, надеюсь, вы уже начинаете понимать, почему традиционная терминология здесь не совсем применима.

**Осваиваем селекторы**

В дополнение к изученным ранее шести концепциям, есть ещё один немаловажный практический аспект MongoDB, который следует освоить, прежде чем переходить к более сложным темам: это — селекторы запросов. Селектор запросов MongoDB аналогичен предложению whereSQL-запроса. Как таковой он используется для поиска, подсчёта, обновления и удаления документов из коллекций. Селектор — это JSON-объект, в простейшем случае это может быть даже {}, что означает выборку всех документов (аналогичным образом работает null). Если нам нужно выбрать всех единорогов (англ. «unicorns») женского рода, можно воспользоваться селектором {gender:'f'}.

Прежде, чем мы глубоко погрузимся в селекторы, давайте сначала создадим немного данных, с которыми будем экспериментировать. Сперва давайте удалим всё, что до этого вставляли в коллекцию unicorns с помощью команды: db.unicorns.remove() (поскольку мы не передали селектора, произойдёт удаление всех документов). Теперь давайте произведём следующие вставки, чтобы получить данные для дальнейших экспериментов (можете скопировать и вставить это в консоль):

db.unicorns.insert({name: 'Horny', dob: new Date(1992,2,13,7,47), loves: ['carrot','papaya'], weight: 600, gender: 'm', vampires: 63});

db.unicorns.insert({name: 'Aurora', dob: new Date(1991, 0, 24, 13, 0), loves: ['carrot', 'grape'], weight: 450, gender: 'f', vampires: 43});

db.unicorns.insert({name: 'Unicrom', dob: new Date(1973, 1, 9, 22, 10), loves: ['energon', 'redbull'], weight: 984, gender: 'm', vampires: 182});

db.unicorns.insert({name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44), loves: ['apple'], weight: 575, gender: 'm', vampires: 99});

db.unicorns.insert({name: 'Solnara', dob: new Date(1985, 6, 4, 2, 1), loves:['apple', 'carrot', 'chocolate'], weight:550, gender:'f', vampires:80});

db.unicorns.insert({name:'Ayna', dob: new Date(1998, 2, 7, 8, 30), loves: ['strawberry', 'lemon'], weight: 733, gender: 'f', vampires: 40});

db.unicorns.insert({name:'Kenny', dob: new Date(1997, 6, 1, 10, 42), loves: ['grape', 'lemon'], weight: 690, gender: 'm', vampires: 39});

db.unicorns.insert({name: 'Raleigh', dob: new Date(2005, 4, 3, 0, 57), loves: ['apple', 'sugar'], weight: 421, gender: 'm', vampires: 2});

db.unicorns.insert({name: 'Leia', dob: new Date(2001, 9, 8, 14, 53), loves: ['apple', 'watermelon'], weight: 601, gender: 'f', vampires: 33});

db.unicorns.insert({name: 'Pilot', dob: new Date(1997, 2, 1, 5, 3), loves: ['apple', 'watermelon'], weight: 650, gender: 'm', vampires: 54});

db.unicorns.insert({name: 'Nimue', dob: new Date(1999, 11, 20, 16, 15), loves: ['grape', 'carrot'], weight: 540, gender: 'f'});

db.unicorns.insert({name: 'Dunx', dob: new Date(1976, 6, 18, 18, 18), loves: ['grape', 'watermelon'], weight: 704, gender: 'm', vampires: 165});

Теперь, когда данные созданы, можно приступать к освоению селекторов. {поле: значение} используется для поиска всех документов, у которых поле равно значение. {поле1: значение1, поле2: значение2} работает как логическое И. Специальные операторы $lt,$lte, $gt, $gte и $ne используются для выражения операций «меньше», «меньше или равно», «больше», «больше или равно», и «не равно». Например, чтобы получить всех самцов единорога, весящих более 700 фунтов, мы можем написать:

db.unicorns.find({gender: 'm', weight: {$gt: 700}})

//или (что не полностью эквивалентно, но приведено здесь в демонстрационных целях)

db.unicorns.find({gender: {$ne: 'f'}, weight: {$gte: 701}})

Оператор $exists используется для проверки наличия или отсутствия поля, например:

db.unicorns.find({vampires: {$exists: false}})

Вернёт единственный документ. Если нужно ИЛИ вместо И, мы можем использовать оператор $or и присвоить ему массив значений, например:

db.unicorns.find({gender: 'f', $or: [{loves: 'apple'}, {loves: 'orange'}, {weight: {$lt: 500}}]})

Вышеуказанный запрос вернёт всех самок единорогов, которые или любят яблоки, или любят апельсины, или весят менее 500 фунтов.

В нашем последнем примере произошло кое-что интересное. Вы заметили — поле loves это массив. MongoDB поддерживает массивы как объекты первого класса. Это потрясающе удобная возможность. Начав это использовать, вы удивитесь, как вы раньше жили без этого. Самое интересное это та простота, с которой делается выборка по значению массива: {loves: 'watermelon'} вернёт нам все документы, у которыхwatermelon является одним из значений поля loves.

Это ещё не все операторы. Самый гибкий оператор — $where, позволяющий нам передавать JavaScript для его выполнения на сервере. Это описано в разделе [Сложные запросы](http://ru.wiki.mongodb.org/display/DOCS/Advanced+Queries) на сайте MongoDB. Мы изучили основы, которые нам нужны для начала работы. Это также то, что вы будете использовать большую часть времени.

Мы видели, как эти селекторы могут быть использованы с командой find. Они также могут быть использованы с командой remove, которую мы кратко рассмотрели, командой count, на которую мы пока не взглянули, но которую вы скорее всего изучите, и командой update, с которой в дальнейшем мы проведём большую часть времени.

ObjectId, сгенерированный MongoDB для поля \_id, подставляется в селектор следующим образом:

db.unicorns.find({\_id: ObjectId("TheObjectId")})

### Обновление данных: замена и $set

В простейшей форме, update принимает 2 аргумента: селектор (where) для выборки и то, чем обновить соответствующее поле. Чтобы Roooooodles прибавил в весе, используем следующий запрос:

db.unicorns.update({name: 'Roooooodles'}, {weight: 590})

(Если в ходе экспериментов вы удалили данные из ранее созданной коллекции unicorns, сделайте всем документам remove, и вставьте их заново с помощью кода из главы 1)

В реальной жизни, конечно, следует обновлять документы, выбирая их по \_id, однако, поскольку я не знаю какой \_id MongoDB сгенерировала для вас, будем выбирать по имени — name. Теперь, давайте взглянем на обновленную запись:

db.unicorns.find({name: 'Roooooodles'})

Вот и первый сюрприз, который нам преподнёс update. Документ не найден, поскольку второй параметр используется для **полной замены**оригинала. Иными словами, update нашёл документ по имени и заменил его целиком на новый документ (свой второй параметр). Вот в чём отличие от SQL-команды UPDATE. Иногда это идеальный вариант, который может использоваться для некоторых действительно динамических обновлений. Однако, если вам нужно всего лишь изменить пару полей, лучше всего использовать модификатор $set:

db.unicorns.update({weight: 590}, {$set: {name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44), loves: ['apple'], gender: 'm', vampires: 99}})

Это восстановит утерянные ранее поля. Поле weight не перезапишется, поскольку мы его не передали в запрос. Теперь, если выполнить:

db.unicorns.find({name: 'Roooooodles'})

мы получим ожидаемый результат. Таким образом, в первом примере правильно было бы обновить weight следующим образом:

db.unicorns.update({name: 'Roooooodles'}, {$set: {weight: 590}})

### Модификаторы обновления

Кроме $set можно использовать и другие модификаторы для разных изящных вещей. Все эти модификаторы обновления действуют над полями — так что ваш документ не окажется перезаписан целиком. Например, модификатор $inc служит для того, чтобы изменить поле на положительную (увеличить) или отрицательную (уменьшить) величину. Например, если единорог Pilot был ошибочно награждён за убийство пары лишних вампиров, мы можем исправить эту ошибку следующим образом:

db.unicorns.update({name: 'Pilot'}, {$inc: {vampires: -2}})

Если Aurora внезапно пристрастилась к сладостям, мы можем добавить соответствующее значение к ее полю loves с помощью модификатора $push:

db.unicorns.update({name: 'Aurora'}, {$push: {loves: 'sugar'}})

Информацию об остальных модификаторах можно найти в разделе [Обновление](http://ru.wiki.mongodb.org/display/DOCS/Updating) на сайте MongoDB.

### Обновление/вставка

Один из приятных сюрпризов операции обновления — это возможность обновления/вставки (*upsert*от*update*— обновить и*insert*— вставить) Обновление/вставка обновляет документ, если он найден, или создаёт новый — если не найден. Обновление/вставка — полезная вещь в некоторых случаях; когда столкнётесь с подобным, сразу поймёте. Чтобы разрешить вставку при обновлении, установите третий параметр в true.

Пример из жизни — счётчик посещений для веб-сайта. Если мы хотим в реальном времени видеть количество посещений страницы, мы должны посмотреть, существует ли запись, и — в зависимости от результата — выполнить update либо insert. Если опустить (или установить в false) третий параметр, следующий пример не сработает:

db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}});

db.hits.find();

Однако, если разрешить вставку при обновлении, результаты будут иными:

db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, true);

db.hits.find();

Поскольку документы с полем page, равным unicorns, не существуют, то будет создан новый документ. Если выполнить это вторично, существующий документ будет обновлён, и поле hits увеличится до 2.

db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, true);

db.hits.find();

### Множественные обновления

Последний сюрприз метода update — это, то что он по умолчанию обновляет лишь один документ. До сих пор это было логично в случае с уже рассмотренными примерами. Однако, если выполнить что-нибудь вроде:

db.unicorns.update({}, {$set: {vaccinated: true }});

db.unicorns.find({vaccinated: true});

, то вы очевидно будете ожидать, что все единороги будут привиты (*vaccinated*). Чтобы это сработало, нужно установить четвертый параметр в true:

db.unicorns.update({}, {$set: {vaccinated: true }}, false, true);

db.unicorns.find({vaccinated: true});

### Выбор полей

Прежде чем переходить к курсорам, следует знать, что find принимает второй необязательный параметр. Это — список полей, которые мы хотим получить. Например, мы можем получить все имена единорогов следующим запросом:

db.unicorns.find(null, {name: 1});

Поле \_id по умолчанию возвращается всегда. Мы можем явным способом исключить его, указав {name:1, \_id: 0}.

За исключением поля \_id, нельзя смешивать включения и исключения полей. Задумавшись, можно понять, зачем так сделано. Можно или хотеть включить или хотеть наоборот — исключить определённые поля явным образом.

### Сортировка

Я уже несколько раз упомянул, что find возвращает курсор, который исполняется отложенно — по мере необходимости. Однако, вы уже без сомнения могли видеть, что find исполняется мгновенно. Такое поведение характерно только для консоли. Можно пронаблюдать за истинным поведением курсоров, взглянув на любой из методов, который мы можем присоединить к find. Первым из них будет sort. Синтаксисsort примерно такой же, как у выбора полей, который мы видели в предыдущем разделе. Мы указываем поля, по которым надо сортировать, используя 1 для сортировки по возрастанию и -1 для сортировки по убыванию. Например:

//сортируем по весу — от тяжёлых к лёгким единорогам

db.unicorns.find().sort({weight: -1})

//по имени единорога, затем по числу убитых вампиров:

db.unicorns.find().sort({name: 1, vampires: -1})

Подобно реляционной базе данных, MongoDB может использовать индексы для сортировки. Детальнее мы рассмотрим индексы несколько позже. Однако следует знать, что без индекса MongoDB ограничивает размер сортируемых данных. Если вы попытаетесь отсортировать большой объем данных, не используя индекс, вы получите ошибку. Некоторые считают это ограничением. Хотя я думаю, что и другим базам данных не мешало бы запрещать выполнение неоптимальных запросов. (Я не стану превращать каждый недостаток MongoDB в её достоинство, однако я сталкивался с большим числом неоптимальных баз данных, которым очень не хватало подобного режима строгой проверки.)

### Разбиение на страницы

Разбиение на страницы может быть осуществлено с помощью методов limit и skip. Чтобы получить второго и третьего по весу единорога, можно выполнить:

db.unicorns.find().sort({weight: -1}).limit(2).skip(1)

Используя limit вместе с sort можно избежать проблем с сортировкой по неиндексированным полям.

### Count

Консоль позволяет выполнить count прямо над коллекцией:

db.unicorns.count({vampires: {$gt: 50}})

На практике же count — это метод курсора, консоль просто обеспечивает удобное сокращение. С драйверами, не поддерживающим подобного сокращения, нужно писать что-то вроде этого (конечно, и в консоли тоже так можно):

db.unicorns.find({vampires: {$gt: 50}}).count()

## Моделирование данных

Давайте сменим тему и поговорим о более абстрактных концепциях MongoDB. Довольно просто объяснять новые термины и новый синтаксис. Гораздо сложнее говорить о моделировании в терминах новой парадигмы. Смысл в том, что большинство из нас привыкли пробовать любую новую технологию, моделируя реальные задачи. Мы поговорим об этом, но в конечном счете вы должны попрактиковаться и изучить реальный код.

Когда речь заходит о моделировании данных, то документ-ориентированные базы данных не настолько сильно отличаются от реляционных, как другие NoSQL-решения. Существующие различия не столь велики, однако это не уменьшает их важности.

### Отсутствие JOIN-ов

Первое и самое фундаментальное различие, с которым вам надо свыкнуться, это отсутствие у MongoDB аналога конструкции JOIN. Неизвестно почему именно MongoDB не поддерживает JOIN-синтаксиса, однако точно можно сказать, что JOIN-ы не масштабируемы. Это значит, что когда вы начнёте разделять данные горизонтально, вам всё равно придётся выполнять JOIN-ы на клиенте (которым является сервер приложений). Независимо от причин, факт остаётся фактом: данные реляционны по своей природе, но MongoDB не поддерживает JOIN-ов.

Мы должны делать JOIN-ы вручную, в коде своего приложения. По существу, мы должны делать второй запрос, чтобы найти связанные данные. Создание данных тут не сильно отличается от создания внешних ключей в реляционных базах. Теперь давайте от единорогов (unicorns) перейдём к сотрудникам (employees). Первым делом создадим сотрудника (я явным образом привожу здесь \_id, чтобы наши примеры выполнялись как задумано)

db.employees.insert({\_id: ObjectId("4d85c7039ab0fd70a117d730"), name: 'Leto'})

Теперь добавим пару сотрудников и сделаем Leto их менеджером:

db.employees.insert({\_id: ObjectId("4d85c7039ab0fd70a117d731"), name: 'Duncan', manager: ObjectId("4d85c7039ab0fd70a117d730")});

db.employees.insert({\_id: ObjectId("4d85c7039ab0fd70a117d732"), name: 'Moneo', manager: ObjectId("4d85c7039ab0fd70a117d730")});

(стоит повторить, что \_id может быть любым уникальным значением. Поскольку в жизни вы скорее всего станете использовать ObjectId, мы также здесь используем его.)

Чтобы найти всех сотрудников, принадлежащих Leto, выполним просто:

db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})

Никакой магии. В худших случаях отсутствие JOIN-ов чаще всего потребует дополнительного запроса (как правило индексированного).

#### Массивы и вложенные документы

Но тот факт, что у MongoDB нет JOIN-ов ещё не означает, что у неё не припасено пару козырей в рукаве. Помните, как мы вкратце поведали ранее о поддержке в MongoDB массивов, как объектов первого класса? Оказывается, что она чертовски удобна, когда требуется смоделировать отношения «один-ко-многим» или «многие-ко-многим». Например, если у сотрудника есть несколько менеджеров, мы просто можем сохранить их в виде массива:

db.employees.insert({\_id: ObjectId("4d85c7039ab0fd70a117d733"), name: 'Siona', manager: [ObjectId("4d85c7039ab0fd70a117d730"), ObjectId("4d85c7039ab0fd70a117d732")] })

А самое интересное, что в одних документах manager можно сделать скалярным значением, а в других — массивом. А наш предыдущий запрос find сработает в обоих случаях:

db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})

Вскоре вы убедитесь, что массивы значений намного удобнее в использовании, нежели таблицы связи «многие-ко-многим».

Кроме массивов MongoDB также поддерживает вложенные документы. Попробуйте вставить документ со вложенным документом, например:

db.employees.insert({\_id: ObjectId("4d85c7039ab0fd70a117d734"), name: 'Ghanima', family: {mother: 'Chani', father: 'Paul', brother: ObjectId("4d85c7039ab0fd70a117d730")}})

Вложенные документы можно запрашивать с помощью точечной нотации:

db.employees.find({'family.mother': 'Chani'})

Мы кратко обсудим, где могут использоваться вложенные документы, и как их следует применять.

#### DBRef

MongoDB поддерживает понятие под названием DBRef, которое является соглашением, принятым во многих драйверах. Когда драйвер видитDBRef, он может автоматически получить связанный документ. DBRef включает в себя коллекцию и \_id документа, на который он ссылается. Это означает следующее — документы из одной и той же коллекции могут ссылаться на другие документы из различных коллекций. То есть документ 1 может ссылаться на документ из коллекции managers, в то же время документ 2 может ссылаться на документ из коллекции employees.

#### Денормализация

Ещё одна альтернатива использованию JOIN-ов — денормализация. Исторически денормализация использовалась для оптимизации производительности, или когда с данных (например, журнала аудита) необходимо было иметь возможность делать снимок. Однако с быстрым ростом NoSQL решений, многие из которых лишены JOIN-ов, денормализация стала в порядке вещей. Это не означает, что нужно дублировать всё подряд в любых документах. Можно остерегаться дублирования данных, а можно соответствующим образом продумать архитектуру своей базы.

К примеру, мы разрабатываем форум. Традиционный путь ассоциировать пользователя с его постом — это колонка userid в таблице posts. с такой моделью нельзя отобразить список постов без дополнительного извлечения данных (JOIN) из таблицы пользователей. Возможное решение — хранить имя пользователя (name) вместе с userid для каждого поста. Можно также вставлять небольшой встроенный документ, например, user: {id: ObjectId('Something'), name: 'Leto'}. Да, если позволить пользователям изменять своё имя, нам придётся обновлять каждый документ (пост) — это один лишний запрос.

Не всем легко приспособиться к такому подходу. Во многих случаях даже не имеет смысла этого делать. Всё же не бойтесь экспериментировать с таким подходом. Иногда это бывает полезным — чуть ли не единственным правильным — решением.

#### Что выбрать?

Также полезной стратегией в случаях отношения «один-ко-многим» или «многие-ко-многим» является массив идентификаторов. Бытует мнение, что DBRef используется не так часто, но конечно вы можете поэкспериментировать с ним. Обычно начинающие разработчики не уверены что подойдёт им лучше — вложенные документы или DBRef.

Во-первых, следует помнить, что одиночный документ ограничен в размере до 4 мегабайт. Факт ограничения (пусть и такого щедрого) размера документа даёт понимание о том, как их следует использовать. Теперь понятно, что большинство разработчиков склоняются к использованию заданных вручную ссылок. Вложенные документы используются часто, но для небольших объёмов данных, если их желательно всегда извлекать вместе с родительским документом. Примером из жизни может быть документ accounts, сохраняемый с каждым пользователем, например:

db.users.insert({name: 'leto', email: 'leto@dune.gov', account: {allowed\_gholas: 5, spice\_ration: 10}})

Это не означает, что можно недооценивать мощь вложенных документов, либо отбрасывать их, как мелкую, второстепенную утилиту. Намного проще живётся, когда структура ваших данных напрямую отображает структуру ваших объектов. Особенно ценным является то, что MongoDB позволяет запрашивать и индексировать поля вложенных документов.

### Мало или много коллекций

Учитывая то, что коллекции не привязывают нас к конкретной схеме, вполне возможно обойтись одной коллекцией, имеющей документы разной структуры. Построенные на MongoDB системы, с которыми мне приходилось сталкиваться, как правило, были похожи на реляционные базы данных. Другими словами, то, что являлось бы таблицей в реляционной базе данных, скорее всего реализуется, как коллекция в MongoDB (таблицы-связки «многие-ко-многим» являются важным исключением).

Дело принимает интересный оборот, если воспользоваться вложенными документами. Пример, который первым делом приходит на ум, это блог. Допустим, есть коллекция posts и коллекция comments, и каждый пост должен иметь вложенный массив комментариев. Если оставить в стороне ограничение 4Мб («Гамлет» на английском едва дотягивает до 200 килобайт, насколько же должен быть популярным ваш блог?), большинство разработчиков предпочитают разделять сущности. Так понятнее и яснее.

Нет какого бы то ни было строгого правила (ну, кроме 4MB). Поэкспериментируйте с различными подходами, и вам станет ясно, что будет правильнее, а что — нет.