

Sprawozdanie z Entity Framework

Magdalena Nowak czwartek 14.40 B

Część z zajęć:

1. Stworzyłam projekt zgodnie z wytycznymi, poniżej zamieszczono kod, który pozwala wygenerować bazę. Od razu zawiera encję potrzebną w realizacji dalszej części projektu – Customers.

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new ProdContext())
        {
            var wasCreated = db.Database.CreateIfNotExists();
            db.Database.Connection.Open();
        }
    }
}

public class Category
{
    [Key]
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public virtual List<Product> Products { get; set; }
}

public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public int UnitsInStock { get; set; }
    public int CategoryId { get; set; }
    public decimal UnitPrice { get; set; }
}

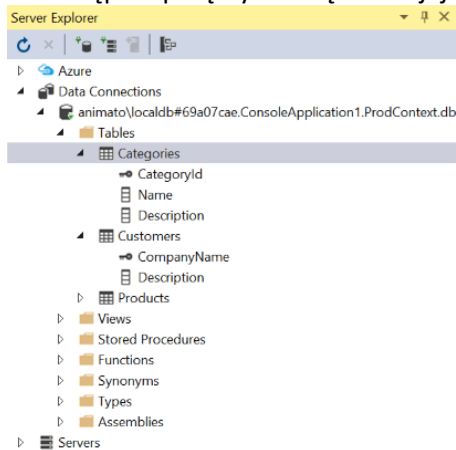
public class Customer
{
    [Key]
    public string CompanyName { get; set; }
    public string Description { get; set; }
}

public class ProdContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }
}
```

2. Poniżej znajduje się kod będący częścią klasy main, który realizuje dodanie nowego rekordu do tabeli Category

```
var wasCreated = db.Database.CreateIfNotExists();
db.Database.Connection.Open();
Console.WriteLine("Enter a name for a new Category: ");
var name = Console.ReadLine();
//zaistancjonuj kategorię o podanej nazwie
var category = new Category { Name = name };
//dodanie zaistancjonowanego obiektu do kontekstowej kolekcji kategorii
db.Categories.Add(category);
db.SaveChanges();
```

3. Następnie połączyłam się do mojej lokalnej bazy



4. Włączyłam możliwość migrowania oraz dodałam migracje.

5. Formularz CategoryForm:

```
public partial class CategoryForm : Form
{
    private ProdContext db = new ProdContext();

    public CategoryForm()
    {
        InitializeComponent();
    }

    //Load to domyślny Event całego formularza
    private void CategoryForm_Load(object sender, EventArgs e)
    {
        //ładuje do bufora danych na kliencie z serwera bazy danych
        db.Categories.Load();
        //połączenie kontrolki z danymi z Entity Framework
        categoryBindingSource.DataSource = db.Categories.Local.ToBindingList();
        db.Products.Load();
        productBindingSource.DataSource = db.Products.Local.ToBindingList();
    }

    //po kliknięciu na komórkę z Categories
    private void categoryDataGridView_CellContentClick(object sender, DataGridViewCellEventArgs e)
    {
        string a;
        if (e.ColumnIndex == 1)
        {
            var categoryId =
                Convert.ToInt32(categoryDataGridView.Rows[e.RowIndex].Cells[0].Value);
            filterProducts(categoryId);
        }
    }

    private void filterProducts(int categoryId)
    {
        //method syntax
        var products = db.Products
            .Where(product => product.CategoryId == categoryId)
            .ToList();
        productDataGridView.DataSource = products;
    }
}
```

```

private void filterProductsQuery(int categoryId)
{
    var query = from p in db.Products
                join c in db.Categories on p.CategoryId equals c.CategoryId
                where c.CategoryId == categoryId
                select p;
    List<Product> products = query.ToList<Product>();
}

private void Order_Add(object sender, EventArgs e)
{
    frm_Order f = new frm_Order();
    DialogResult res=f.ShowDialog(this);
    if (res==DialogResult.OK)
    {
        Order o = f.order;
    }
}

private void Order_Save(object sender, EventArgs e)
{
    db.SaveChanges();
    this.orderDataGridView.Refresh();
}

private void Category_Save(object sender, EventArgs e)
{
    db.SaveChanges();
    this.categoryDataGridView.Refresh();
}

private void Delete(object sender, EventArgs e)
{
    db.SaveChanges();
    this.orderDataGridView.Refresh();
}

private void button3_Click(object sender, EventArgs e)
{
    frm_Customer f = new frm_Customer();
    DialogResult res = f.ShowDialog(this);
    if (res == DialogResult.OK)
    {
        Customer c = f.customer;
        db.Customers.Add(c);
        db.SaveChanges();
    }
}
}

```

6. Następnie dodałam metody dostępne.

```

//w main:
method based syntax
var categories = db.Categories
    .Select(c => c.Name).ToList();

Console.WriteLine("Categories Names:");
foreach (String c in categories)
{
    Console.WriteLine(c);
}

```

```

//Display all Categories from the database
var query = from b in db.Categories
            orderby b.Name descending
            select b;

Console.WriteLine("All categories in the database (method syntax:");
foreach (var item in query)
{
    Console.WriteLine(item.Name);
}

Console.WriteLine("Products quantity for each category:");

Methods.CountProductsForCategoryQ(db);

CategoryForm f = new CategoryForm();
f.ShowDialog();
Console.WriteLine("Press any key to exit...");
Console.ReadKey();

```

7. Utworzyłam też nową klasę Methods, w której stworzyłam metody wyświetlające wszystkie kategorie i produkty w różnych wariantach (Joiny, Navigation Property, Eager Loading), a także inne metody.

Oznaczenie: M w końcówce oznacza, że jest to Method Syntax, zaś Q - Query Syntax

```

class Methods
{
    //METODY DOSTĘPNE
    //Navigation Property - z tabeli Category, po zależnościach dochodzimy do Produktów
    public static void PrintCategoriesAndProductsQ(ProdContext db)
    {
        var query = from b in db.Categories
                    orderby b.Name descending
                    select b;

        foreach (var categoryName in query)
        {
            Console.WriteLine("Category name: {0}", categoryName.Name);

            foreach (Product product in categoryName.Products)
                Console.WriteLine("description: {0}", product.Name);
        }
    }

    //Navigation Properties, wraz z kategorią nazwy produktów z danej kategorii
    public static void PrintCategoriesAndProductsM(ProdContext db)
    {
        IQueryable<Category> query = db.Categories;

        foreach (var categoryName in query)
        {
            Console.WriteLine(categoryName.Name);
            foreach (Product product in categoryName.Products)
            {
                Console.WriteLine("Product: {0}", product.Name);
            }
        }
    }
}

```

```
//Eager loading + Navigation Property
public static void PrintCategoriesAndProductsEagerLoadingQ(ProdContext db)
{
    var query = from b in db.Categories.Include("Products")
                orderby b.Name descending
                select b;

    foreach (var categoryName in query)
    {
        Console.WriteLine("Category name: {0}", categoryName.Name);

        foreach (Product product in categoryName.Products)
            Console.WriteLine("description: {0}", product.Name);
    }
}

//+Eager loading - zapytanie o jeden typ tabeli ładuje od razu także powiązaną tabelę jako
część zapytania
//+Navigation Property
public static void PrintCategoriesAndProductsEagerLoadingM(ProdContext db)
{
    var categories = db.Categories
        .Include(c => c.Products)
        //opcjonalnie po nazwie encji Include("Products")
        .ToList();

    foreach (var record in categories)
    {
        Console.WriteLine("Category Name: {0}", record.Name);
        foreach (var p in record.Products)
        {
            Console.WriteLine("Product: {0}", p.Name);
        }
    }
}

//Join
public static void PrintCategoriesAndProductsJoinM(ProdContext db)
{
    var query = db.Categories
        .Join(db.Products,
            product => product.CategoryId,
            category => category.CategoryId,
            (category, product) =>
                new {
                    c = category,
                    p = product
                });

    foreach (var record in query)
    {
        Console.WriteLine("Category Name " + record.c, "Product Name " + record.p);
    }
}

public static void PrintCategoriesAndProductsJoinQ(ProdContext db)
{
    var query = from ca in db.Categories
                join pr in db.Products
                  on ca.CategoryId equals pr.CategoryId
                //orderby c.Name
                select new
                {
                    c = ca,
                    p = pr
                };
}
```

```

        foreach (var record in query)
        {
            Console.WriteLine("Category Name: " + record.c.Name + " Product Name: "
                + record.p.Name);
        }
    }

//dodatkowe:
public static void PrintOnlyCategoriesNamesM(ProdContext db)
{
    List<String> categoryNames = db.Categories
        .Select(c => c.Name)
        .ToList();

    foreach (var categoryName in categoryNames)
    {
        Console.WriteLine(categoryName);
    }
}

public static void PrintOnlyCategoriesNamesQ(ProdContext db)
{
    var query = from c in db.Categories
                orderby c.Name descending
                select c.Name;

    foreach (var categoryName in query)
    {
        Console.WriteLine(categoryName);
    }
}

//Metoda zliczająca ilość produktów dla każdej kategorii:
//Agregacja - Count
public static void CountProductsForCategoryQ(ProdContext db)
{
    var query = from c in db.Categories
                orderby c.Name descending
                select new
                {
                    CategoryID = c.CategoryId,
                    CategoryName = c.Name,
                    ProductsQuantity = c.Products.Count()
                };

    foreach (var c in query)
    {
        Console.WriteLine("Category Name: {0} \t ProductsQuantity: {1}",
            c.CategoryName,
            c.ProductsQuantity);
    }
}
}

```

Część II

Rozszerzenie aplikacji o możliwość składania zamówień na produkty. Zrealizowałam jako rozszerzenie rozpoczętej aplikacji WindowsFormowej.

Dodałam tabelę Orders:

Klasa Order:

```
public class Order
{
    [Key]
    public int OrderId {get; set; }
    public virtual Customer Customer { get; set; }
    public virtual Product Product { get; set; }
    public int Quantity { get; set; }
    public string Status { get; set; }
    public DateTime Date { get; set; }
    public string CustomerName { get {
        return Customer==null ? "" : Customer.CompanyName;
    }}
    public string ProductName { get {
        return Product == null ? "" : Product.Name;
    }}
}
```

Dodanie zbioru obiektów typu Order:

```
public class ProdContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }
}
```

Formularz Categories_Form zmieniłam na Main_Form, wygląda on teraz tak:

The screenshot shows a Windows application window titled "Shop". It contains three main data grids and several buttons.

Categories:

CategoryId	Name	Description
1	kawa	
2	herbata	
11	mleko	
12	owoce	
14	woda	

Products:

ProductId	Name	UnitsInStock
1	parzona	20
2	rozpuszczalna	20
3	malinowa	10
4	miętowa	15
5	pomarańczowa	10
6	pomarańczowa	10
8	jablka	100

Orders:

OrderId	Customer	Product	Quantity	Status
1	Groszek	parzona	10	Zapłacone
2	Biedronka	rozpuszczalna	10	Nowe
3	Stokrotka	malinowa	10	Anulowane
4	Lewiatan	miętowa	10	Nowe
6	Piano	rozpuszczalna	17	Zapłacone
7	Biedronka	parzona	0	Nowe

Buttons:

- Top right: "Add order" (green), "New Client" (purple)
- Bottom left: "Filter orders" with buttons "Z", "N", "A", "All"
- Bottom right: "Change order status" (yellow), "Delete Expired Orders" (red)

Poza kontrolkami ProductsDataGridView oraz CategoryDataGridView, dodałam też kontrolkę Orders DataGridView, połączyłam ją z danymi ściągniętymi z kontekstu bazy danych (orderBindingSource). Dodałam też kilka przycisków realizujących różne funkcjonalności opisane poniżej.

Kod przygotowujący formularz MainForm do pracy na nim (czyli inicjalizacja + ładowanie danych):

```
private ProdContext db = new ProdContext();
//będą mi potrzebne do przekazywania dalej do nowych formularzy, żeby pracować na tym samym
kontekście:
BindingList<Product> products;
BindingList<Customer> customers;
BindingList<Category> categories;
BindingList<Order> orders;

public MainForm()
{
    InitializeComponent();
}

//Load to domyślny Event całego formularza
private void MainForm_Load(object sender, EventArgs e)
{
    //ładowanie do bufora danych na kliencie z serwera bazy danych
    db.Categories.Load();
    //połączenie kontrolki z danymi z Entity Framework
    categoryBindingSource.DataSource = db.Categories.Local.ToBindingList();
    categories = db.Categories.Local.ToBindingList();

    db.Products.Load();
    products = db.Products.Local.ToBindingList();
    productBindingSource.DataSource = products;

    db.Customers.Load();
    customers = db.Customers.Local.ToBindingList();

    db.Orders.Load();
    orders = db.Orders.Local.ToBindingList();
    orderBindingSource.DataSource = orders;
}
```

Filtrowanie produktów, w zależności od tego, na jaką nazwę kategorii klikniemy:

```
private void categoryDataGridView_CellContentClick(object sender, DataGridViewCellEventArgs e)
{
    //po kliknięciu na komórkę z nazwą kategorii
    if (e.ColumnIndex == 1)
    {
        var categoryId =
Convert.ToInt32(categoryDataGridView.Rows[e.RowIndex].Cells[0].Value);
        filterProductsQNP(categoryId);
    }
}

//Navigation Property, po zależnościach dochodzimy od wybranej kategorii do jej produktów
private void filterProductsQNP(int categoryId)
{
    var query = from b in db.Categories
                where b.CategoryId == categoryId
                select b;

    foreach (var categoryName in query)
    {
        List<Product> products = categoryName.Products;
        productDataGridView.DataSource = products;
    }
}
```

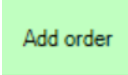

To samo co wyżej, Query Syntax i Method Syntax, wersje bez Navigation Property:

```
private void filterProducts(int categoryId)
{
    //query
    var query = from p in db.Products
                join c in db.Categories on p.CategoryId equals c.CategoryId
                where c.CategoryId == categoryId
                select p;
    List<Product> products = query.ToList<Product>();
    productDataGridView.DataSource = products;

    //method syntax
    /*var products = db.Products
        .Where(product => product.CategoryId == categoryId)
        .ToList();
    productDataGridView.DataSource = products;
    */
}
```

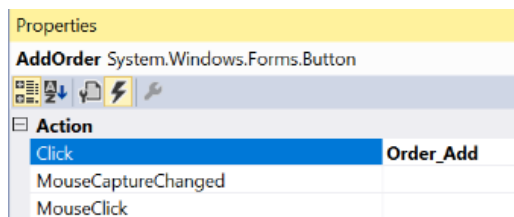
Dodane funkcjonalności:

1. Dodawanie nowego zamówienia

A green rectangular button with the text "Add order" in black.

Po kliknięciu tego przycisku, otwiera się nowy Formularz.

Ustawiony Event na zdarzenie kliknięcia na AddOrder (podobnie ustawion Eventy na inne przyciski, nie pokazuję już tego w dalszej części sprawozdania):

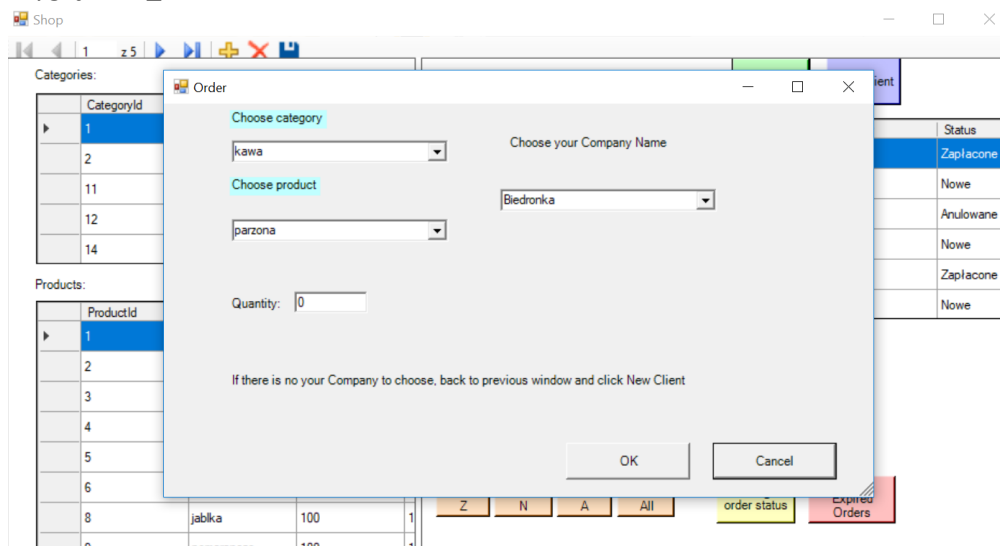


Funkcja w MainForm, która obsługuje formularz frm_Order

```
private void Order_Add(object sender, EventArgs e)
{
    frm_Order f = new frm_Order(products, customers, categories);
    DialogResult res=f.ShowDialog(this);
    if (res==DialogResult.OK)
    {
        Order o = f.order;
        if(getProductQuantity(o.Product.ProductId)<o.Quantity)
            errorProvider1.SetError(AddOrder, "No enough products in shop");
        else
        {
            Product p = db.Products.Local.FirstOrDefault(pr => pr.ProductId ==
                f.order.Product.ProductId);
            o.Product = p;
            Customer customer = db.Customers.Local.FirstOrDefault(c => c.CompanyName ==
                f.order.Customer.CompanyName);
            o.Customer = customer;
            db.Orders.Add(o);

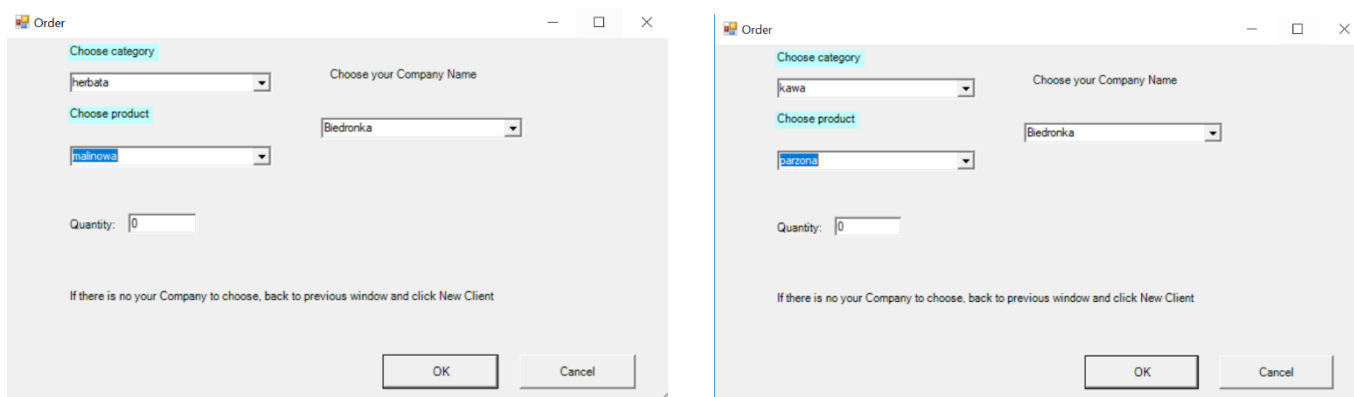
            db.SaveChanges();
            orderDataGridView.Refresh();
            errorProvider1.SetError(AddOrder, "");
        }
    }
}
```

Wygląd frm_Order:



Zamówienie składa się przez wybranie odpowiedniego produktu z listy rozwijanej zrealizowanej kontrolką ComboBox, do której są ładowane nazwy produktów z bazy – lista produktów jest przekazywana BindingList przy tworzeniu formularza od głównego formularza, który ma otwarty kontekst bazy.

Dodatkowo dodałam możliwość wybrania kategorii i wtedy lista produktów jest filtrowana.



Kod realizujący filtrowanie:

```
private void categoryComboBox_SelectedChangeCommitted(object sender, EventArgs e)
{
    int CategoryId = (int)categoryComboBox.SelectedValue;
    filterProducts(CategoryId);
}

private void filterProducts(int categoryId)
{
    //method syntax
    var p = this.products
        .Where(product => product.CategoryId == categoryId)
        .ToList();
    productBindingSource1.DataSource = p;
}
```

```

        //query
        // var query = from p in db.Products
        //     join c in db.Categories on p.CategoryId equals c.CategoryId
        //     where c.CategoryId == categoryId
        //     select p;
        //List<Product> products = query.ToList<Product>();
    }

```

Ustawia się ilość produktu i wybiera (podobnie z ComboBox) nazwę swojej firmy.

Następnie zatwierdza się wszystko przyciskiem OK, jeśli klikniemy Cancel to nic się nie stanie. Jeśli jednak OK to zostanie dodane zamówienie do bazy.

```

public void btOK_Click(object sender, EventArgs e)
{
    int ProductId = (int)productComboBox.SelectedValue;
    order.Product = new Product() { ProductId = ProductId };
    string CompanyName = (string)customerComboBox.SelectedValue;
    order.Customer = new Customer() { CompanyName = CompanyName };
    order.Date = DateTime.Now;
    order.Status = "Nowe";
}

```

order to pole w tym formularzu, do którego odwołuję się, gdy zamknie się już ten formularz i w MainForm będę chciała dodać to zamówienie to tabeli Orders.

Zanim jednak to nastąpi zostanie sprawdzone czy przypadkiem liczba dostępnych produktów nie jest za mała do realizacji zamówienia.

```

//zastosowanie immediate query execution
//qntReserved to pojedyncza wartość, aby ją zwrócić zapytanie musi zostać
//wykonane od razu
private int getProductQuantity(int id)
{
    var product = db.Products.FirstOrDefault(p => p.ProductId == id);
    int qnt = product.UnitsInStock;
    var Orders = db.Orders.Where(o => o.Product.ProductId == id).ToList<Order>();
    var qntReserved = 0;
    if (Orders != null)
        qntReserved = Orders.Sum(o => o.Quantity);
    return qnt - qntReserved;
}

```

Jeśli jest wyświetli się czerwony wykrzyknik (errorProvider) przy kontrolce Add Order i zamówienie nie zostanie dodane do bazy.

2. Dodawanie nowego klienta

Jeżeli nowy klient chce dokonać zamówienia, najpierw musi kliknąć na przycisk:



Metody z Main_Form umożliwiające realizację tej funkcjonalności:

```

private void newClient_Click(object sender, EventArgs e)
{
    frm_Customer f = new frm_Customer();
    DialogResult res = f.ShowDialog(this);
    if (res == DialogResult.OK)
    {
        Customer c = f.customer;
        if(isCustomeCorrect(c))
        {
            db.Customers.Add(c);
            db.SaveChanges();
            errorProvider1.SetError(NewClient, "");
        }
    }
}

```

```

    }
    else
    {
        errorHandler1.SetError(NewClient, "Client already exists");
        //alert, że już jest taki klient w bazie
    }
}

private bool isCustomeCorrect(Customer c)
{
    var customer = db.Customers.FirstOrDefault(cus => cus.CompanyName == c.CompanyName);
    return customer == null;
}
//2 wersja metody

//zastosowanie Immediate Query Execution
//wykorzystano tzw. metodę terminalną ToList(),
//rezultat staje się zmaterializowany,
//i tym samym załadowany do pamięci procesu tego programu
private bool isCustomerCorrectQ(Customer c)
{
    //query base syntax
    var query = from cus in db.Customers
                where cus.CompanyName == c.CompanyName
                select cus;
    List<Customer> customer = query.ToList<Customer>();

    return customer == null;
}

```

Pojawia się nowy formularz:

Użytkownik wpisuje nazwę swojej firmy i może dodać opis.

Po kliknięciu OK, sprawdzane jest czy w bazie nie występuje już taki klient (ponieważ nazwa jest kluczem głównym). Jeżeli tak, to analogicznie do błędu braku produktów wyskakuje wykrzyknik, tym razem z komentarzem, że jest już taki klient.

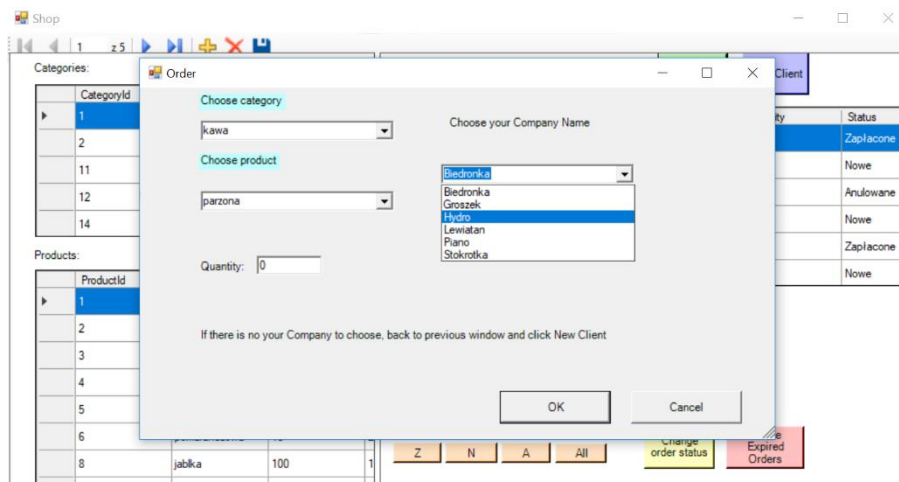
Przykład:

Chciałam dodać klienta „Biedronka”, ale miałam już takiego w bazie, wyskoczył wykrzyknik i komunikat:

Product	Quantity	Status
parzona	10	Zapłacone

Jeśli wszystko przejdzie bez błędu, po dodaniu nowego klienta, pojawia się on w liście rozwijanej w formularzu AddOrder

Przed:



New Customer

Enter your Company name

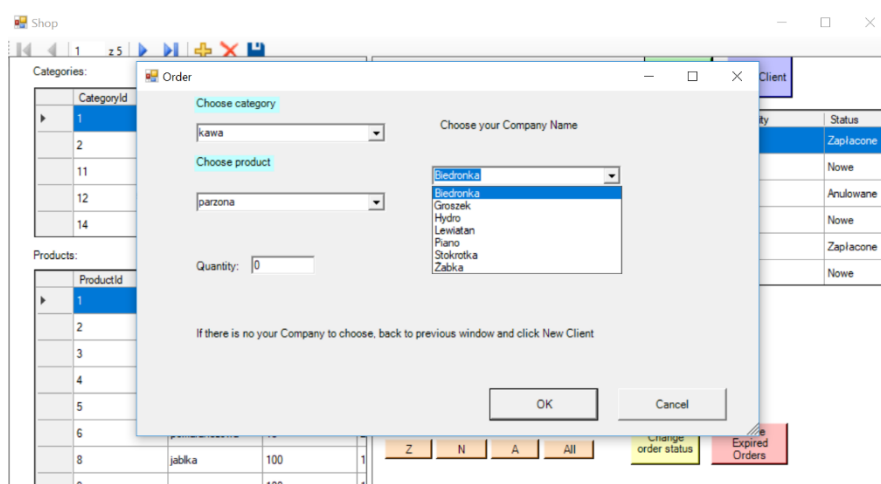
Zabka

Enter description

little shop

OK Cancel

Po:



3. Zmiana statusu zamówienia

Change
order status

Po kliknięciu powyższego przycisku, pojawia się nowe okienko z tylko jednym zamówieniem, tym, na którym obecnie jest zaznaczenie w orderDataGridView.

```
private void changeOrder_Click(object sender, EventArgs e)
{
    var rows = orderDataGridView.SelectedRows;
    int OrderId = (int)((DataGridViewRow)rows[0]).Cells[0].Value;

    var Order = from o in orders
                where o.OrderId == OrderId
                select o;
    var order = Order.First();

    frm_OrdersSettings f = new frm_OrdersSettings(order);
    DialogResult res = f.ShowDialog(this);
    if (res == DialogResult.OK)
    {
        db.SaveChanges();
        orderDataGridView.Refresh();
    }
}
```

Wygląda to tak:

	OrderId	Customer	Product	Quantity	Status
	1	Groszek	parzona	10	Zapłacone
▶	2	Biedronka	rozpuszczalna	10	Nowe
	3	Stokrotka	malinowa	10	Anulowane

Po kliknięciu Change order status, otwiera się nowy formularz:

frm_Orders

Customer Name: Biedronka

Date: środa, 8 listopad

Order Id: 2

Product Name: rozpuszczalna

Quantity: 10

Status: Nowe

Company Name: Biedronka

Description: little greengro

Category Id: 1

Name: rozpuszczaln.

Product Id: 2

Unit Price: 10.00

Units In Stock: 20

Status

☒ Nowe

☐ Zapłacone

☐ Anulowane

OK Cancel

Zmieniam status:

The screenshot shows a Windows form titled 'frm_Orders'. It contains several input fields for order details: Customer Name (Biedronka), Date (środa, 8 listopad), Order Id (2), Product Name (rozpuszczalna), Quantity (10), Status (Zapłacone), Company Name (Biedronka), Description (little greengro), Category Id (1), Name (rozpuszczalna), Product Id (2), Unit Price (10,00), and Units In Stock (20). At the bottom right, there is a 'Status' dialog box with three radio buttons: 'Nowe', 'Zapłacone' (which is selected), and 'Anulowane'. 'OK' and 'Cancel' buttons are at the bottom center.

Po zmianie:

	OrderId	Customer	Product	Quantity	Status
▶	2	Biedronka	rozpuszczalna	10	Zapłacone

Tym razem nie skorzystałam z GridView, bo jest to tylko 1 rekord, tu zastosowałam widok Details.

Po prawej stronie jest realizacja zmiany statusu, jest to wykonane za pomocą GroupBox (dzięki temu zawsze tylko jedna opcja może być wybrana) i każda opcja jest RadioButtonem. W chwili otworzenia okienka status jest ustawiony na obecny, po zmianie i kliknięciu Ok zmiana jest zapamiętywana w bazie.

Kod formularza:

```
public partial class frm_OrdersSettings : Form
{
    public Order order;

    public frm_OrdersSettings(Order _order)
    {
        InitializeComponent();
        order = _order;
    }

    private void frm_Orders_Load(object sender, EventArgs e)
    {
        orderBindingSource.DataSource = order;
        setSelectedStatus(order.Status);
    }

    private string setSelectedStatus()
    {
        if (RB_Nowe.Checked) return "Nowe";
        else if (RB_Zapłacone.Checked) return "Zapłacone";
        else return "Anulowane";
    }
}
```

```

private void setSelectedStatus(String status)
{
    switch(status)
    {
        case "Nowe":
            RB_Nowe.Checked = true;
            break;

        case "Zapłacone":
            RB_Zaplacone.Checked = true;
            break;

        case "Anulowane":
            RB_Anulowane.Checked = true;
            break;
    }
}

private void btOK_Click(object sender, EventArgs e)
{
    order.Status = setSelectedStatus();
}

```

4. Zamiana zamówień nowych, nie zapłaconych przed dłużej niż tydzień na Anulowane.



Po kliknięciu uruchamia się poniższa metoda:

```

//zastosowanie Deferred Execution
//zapytanie jest "zawieszone w powietrzu"
//"Entity Framework won't execute the query against the database until
// it needs the first result. During the first iteration of the foreach loop,
// the query is sent to the database."
//Method Syntax

```

```

private void DeleteExpiredOrders(object sender, EventArgs e)
{
    DateTime WeekAgo = DateTime.Now.AddDays(-7);
    var orders = db.Orders
        .Where(o => o.Status == "Nowe" && o.Date < WeekAgo);

    foreach(var o in orders )
    {
        o.Status = "Anulowane";
    }
    db.SaveChanges();
    this.orderDataGridView.Refresh();
}

```

5. Filtrowanie zamówień po statusie.



Po kliknięciu na odpowiedni przycisk, uruchamiana jest odpowiednia metoda filtrująca, wszystkie query syntax:


```

private void N_filterOrders(object sender, EventArgs e)
{
    var query = from o in db.Orders
                where (o.Status == "Nowe")
                select o;
    List<Order> orders = query.ToList<Order>();
    orderDataGridView.DataSource = orders;
}

private void A_filterOrders(object sender, EventArgs e)
{
    var query = from o in db.Orders
                where (o.Status == "Anulowane")
                select o;
    List<Order> orders = query.ToList<Order>();
    orderDataGridView.DataSource = orders;
}

private void Z_filterOrders(object sender, EventArgs e)
{
    var query = from o in db.Orders
                where (o.Status == "Zapłacone")
                select o;
    List<Order> orders = query.ToList<Order>();
    orderDataGridView.DataSource = orders;
}

private void All_Click(object sender, EventArgs e)
{
    orderDataGridView.DataSource = orders;
}

```

Przykłady:

Po naciśnięciu „Z”:

Add order

New Client

	Orderid	Customer	Product	Quantity	Status
▶	1	Groszek	parzona	10	Zapłacone
	2	Biedronka	rozpuszczalna	10	Zapłacone
	6	Piano	rozpuszczalna	17	Zapłacone
	7	Biedronka	parzona	0	Zapłacone
	8	Piano	parzona	2	Zapłacone
	10	Biedronka	parzona	4	Zapłacone

Filter orders

Z

N

A

All

Change order status

Delete Expired Orders

Po naciśnięciu „N”:

Add order

New Client

Orderid	Customer	Product	Quantity	Status
4	Lewiatan	miętowa	10	Nowe
9	Groszek	miętowa	3	Nowe
11	Biedronka	parzona	2	Nowe
12	Lewiatan	jablka	10	Nowe
13	Hydro	jablka	10	Nowe
14	Biedronka	pomarańcze	10	Nowe

Filter orders

Z

N

A

All

Change order status

Delete Expired Orders

Po naciśnięciu „A”:

Add order

New Client

	Orderid	Customer	Product	Quantity	Status
▷	3	Stokrotka	malinowa	10	Anulowane

Filter orders					Change order status	Delete Expired Orders
Z	N	A	All			

Po naciśnięciu „All”:

Add order

New Client

	Orderid	Customer	Product	Quantity	Status
▷	1	Groszek	parzona	10	Zaplacone
	2	Biedronka	rozpuszczalna	10	Zaplacone
	3	Stokrotka	malinowa	10	Anulowane
	4	Lewiatan	miętowa	10	Nowe
	6	Piano	rozpuszczalna	17	Zaplacone
	7	Biedronka	parzona	0	Zaplacone

Filter orders					Change order status	Delete Expired Orders
Z	N	A	All			

Zastosowane mechanizmy:

Lazy loading – opóźnione ładowanie, dane ładowane są dopiero gdy są potrzebne, a przechowywane jest samo zapytanie.

Domyślnie wszystko jest wykonywane Lazy loading.

Ponieważ zamówienie zawiera tylko ID produktu i ID klienta (w kodzie są to całe obiekty klas Product i Customer, ale w bazie to tylko klucze obce), dodałam możliwość wyświetlania zamówień z dokładniejszymi informacjami o produkcie i kliencie.

```
//Lazy loading - jest domyślne
//Navigation Property - z tabeli Orders, po zależnościach dochodzimy do Product i
Customer, do szczegółów
//Query syntax
public static void PrintOrderWithDetails(ProdContext db)
{
    db.Configuration.LazyLoadingEnabled = true;
    //to nie jest konieczne, bo jest to domyślne ustawienie, ale tak dla pewności...

    var query = from o in db.Orders
                 orderby o.OrderId descending
                 select o;
```

```

foreach (var order in query)
{
    Console.WriteLine("Order nr: {0}, Date: {1}, Quantity {2}, Status {3}",
        order.OrderId, order.Date, order.Quantity, order.Status);

    Product product = order.Product;
    Console.WriteLine("Product details: {0}, {1} {2}", product.ProductId,
        getCategoryName(db,product.CategoryId), product.Name);
    Customer customer = order.Customer;
    Console.WriteLine("Customer details: {0}, {1}\n", customer.CompanyName,
        customer.Description);
}
}

public static string getCategoryName(ProdContext db, int id)
{
    var name = from a in db.Categories
                where a.CategoryId == id
                select a;
    Category category = name.ToList().FirstOrDefault();
    return category.Name;
}

```

Eager loading – zachłanne ładowanie

Powyższa metoda zrealizowana ładowaniem zachłannym.

Od razu ładujemy wszystkie tabele Orders, Products i Customers.

```

//Eager loading + Navigation Property
public static void PrintOrderWithDetailsEL(ProdContext db)
{
    var orders = db.Orders.Include(o => o.Product).Include(o => o.Customer);

    foreach (var order in orders)
    {
        Console.WriteLine("Order nr: {0}, Date: {1}, Quantity {2}, Status {3}",
            order.OrderId, order.Date, order.Quantity, order.Status);

        Product product = order.Product;
        Console.WriteLine("Product details: {0}, {1}", product.ProductId, product.Name);
        Customer customer = order.Customer;
        Console.WriteLine("Customer details: {0}, {1}\n", customer.CompanyName,
customer.Description);
    }
}

```

Deferred execution – wykonanie zapytania jest odraczane do momentu iterowania po zmiennej zapytania, w pamięci jest przechowywane samo zapytanie.

```

//zastosowanie Deferred Execution
//zapytanie jest "zawieszone w powietrzu"
//"Entity Framework won't execute the query against the database until
// it needs the first result. During the first iteration of the foreach loop,
// the query is sent to the database."
//Method Syntax

private void DeleteExpiredOrders(object sender, EventArgs e)
{
    DateTime WeekAgo = DateTime.Now.AddDays(-7);
    var orders = db.Orders
        .Where(o => o.Status == "Nowe" && o.Date < WeekAgo);
}

```

```

        foreach(var o in orders )
        {
            o.Status = "Anulowane";
        }
        db.SaveChanges();
        this.orderDataGridView.Refresh();
    }
}

```

Immediate execution (natychmiastowe wykonanie) - zawsze wtedy, gdy zapytanie zwraca pojedynczą wartość, albo gdy wywołana będzie metoda typu ToList, ToDictionary lub ToArray.

```

//zastosowanie Immediate Query Execution
//wykorzystano tzw. metodę terminalną ToList(),
//rezultat staje się zmaterializowany,
//i tym samym załadowany do pamięci procesu tego programu
private bool isCustomerCorrectQ(Customer c)
{
    //query base syntax
    var query = from cus in db.Customers
                where cus.CompanyName == c.CompanyName
                select cus;
    List<Customer> customer = query.ToList<Customer>();

    return customer == null;
}

//zastosowanie immediate query execution
//qntReserved to pojedyncza wartość, aby ją zwrócić zapytanie musi zostać
//wykonane od razu
private int getProductQuantity(int id)
{
    var product = db.Products.FirstOrDefault(p => p.ProductId == id);
    int qnt = product.UnitsInStock;
    var Orders = db.Orders.Where(o => o.Product.ProductId == id).ToList<Order>();
    var qntReserved = 0;
    if (Orders != null)
        qntReserved = Orders.Sum(o => o.Quantity);
    return qnt - qntReserved;
}

```

Navigation Property – po zależnościach dochodzimy z jednej tabeli do drugiej, jest to alternatywne rozwiązanie do Join, bardzo wygodne.

Przykłady zastosowania:

Filtrowanie produktów na głównym formularzu, po kliknięciu na nazwę kategorii zrealizowane za pomocą NP. Z tabeli Category przechodzę łatwo do Produktów w niej i je podstawiam jako dane do productDataGridView.

```

//Navigation Property, po zależnościach dochodzimy od wybranej kategorii do jej produktów
private void filterProductsQNP(int categoryId)
{
    var query = from b in db.Categories
                where b.CategoryId == categoryId
                select b;

    foreach (var categoryName in query)
    {
        List<Product> products = categoryName.Products;
        productDataGridView.DataSource = products;
    }
}

```

```
//Navigation Properties, wraz z kategorią nazwy produktów z danej kategorii
public static void PrintCategoriesAndProductsM(ProdContext db)
{
    IQueryable<Category> query = db.Categories;

    foreach (var categoryName in query)
    {
        Console.WriteLine(categoryName.Name);
        foreach (Product product in categoryName.Products)
        {
            Console.WriteLine("Product: {0}", product.Name);
        }
    }
}
```

Także w samej klasie Order:

```
public class Order
{
    [Key]
    public int OrderId {get; set; }
    public virtual Customer Customer { get; set; }
    public virtual Product Product { get; set; }
    public int Quantity { get; set; }
    public string Status { get; set; }
    public DateTime Date { get; set; }
    public string CustomerName { get {
        return Customer==null ? "" : Customer.CompanyName;
    }}
    public string ProductName { get {
        return Product == null ? "" : Product.Name;
    }}
}
```

Dochodzę od Order po Customer i Product do Company Name i Name produktu. (Potrzebne mi to było żeby mieć dane do wyświetlania w ComboBox- nie dałoby się wyświetlić obiektu Customer czy Product).

Także wcześniejsze przykłady z Lazy loading i eager loading zawierają Navigation Property.

Fluent API – generalnie oznacza to sposób wywoływania funkcji (podobny do prozy literackiej) polegający na kontynuowaniu wyrażenia i wywoływaniu kolejnych funkcji po kolejnych kropkach.

Przykład:

```
var Orders = db.Orders.Where(o => o.Product.ProductId == id).ToList<Order>();
```

Przykład użycia Fluent API do określania parametrów mapowania pomiędzy kodem w C# a modelem w bazie:

Klasa Order bez Fluent API (przed dodanie atrybutu, określenie mapowania pomiędzy klasami w C# a modelem w bazie danych)

```
public class Order
{
    [Key]
    public int OrderId {get; set; }
```

To samo za pomocą Fluent API wewnątrz funkcji OnModelCreating:

```
modelBuilder.Entity<Order>().HasKey(o => o.OrderId);
```

Inne ustawienia mapowania wykorzystane w programie:

```
public class ProdContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //configuration goes here
        modelBuilder.Entity<Order>().HasKey(o => o.OrderId);
        modelBuilder.Entity<Customer>().Property(c => c.Description).HasMaxLength(255);
        modelBuilder.Entity<Customer>().HasKey(c => c.CompanyName);
        modelBuilder.Entity<Customer>().Property(c => c.CompanyName).HasMaxLength(255);
        modelBuilder.Entity<Product>().HasKey(p => p.ProductId);
        modelBuilder.Entity<Category>().HasKey(c => c.CategoryId);

        base.OnModelCreating(modelBuilder);
    }
}
```

Kod całej aplikacji (ConsoleApplication1) znajduje się tu:

<https://github.com/McDusia/Databases/tree/master/Entity%20Framework>