# Бази данни

## Бази данни (БД, DB)

База данни е колекция от взаимносвързана информация, организирана във вид, удобен за софтуерна обработка.

## Релационни бази данни

Релационните бази данни съхраняват информацията във формата на формално дефинирани таблици.

Релационните бази данни за най-употребяваните в съвременните софтуерни системи.

### Таблици (релации)

Колони (полета, атрибути) – характеризират се с име и дефиниционно множество (домейн); няма наредба.

Редове (записи, n-торки) – притежават стойност за всяка колона; няма наредба.

### Първични ключове (primary keys)

Колона или група колони, която определя еднозначно всеки запис (няма два записа с еднакви стойности).

Използват се също за описване на зависимости между данните в различни таблици.

### Външни ключове (foreign keys)

Колона или група колони, рефериращи първичен ключ (обикновено в друга таблица).

Реализират зависимости (връзки) между данните в различни таблици.

Един или няколко записа в таблицата, в която е дефиниран външния ключ, са свързани с точно един запис в таблицата, чийто първичен ключ се реферира.

### Индекси

Служат за ускоряване манипулацията със записите в таблиците.

„Съдържания”, които улесняват търсенето на конкретни записи в големи обеми информация.

### Схема (schema)

Структурата от таблици, колони, ключове и други ограничения, индекси.

Схемата описва структурата на данните, а записите – самите данни.

## Системи за управление на бази данни (СУБД, DBMS)

Софтуерна система, позволяваща създаването, поддръжката, управлението и употребата на бази данни.

Силно оптимизирани за ефективно манипулиране на данните.

Позволяват дефиниране на предварително определени сложни действия върху данните и изпълнението им (functions, stored procedures, triggers), както и на предварително определени проекции на данните (views).

Съществуват различни СУБД с различни силни и слаби страни.

Големи (enterprise) СУБД (MS SQL Server, Oracle) – за големи софтуерни системи.

Леки (lightweight) СУБД (SQLite, MS Access, MS SQL Server Compact) – за малки приложения.

Ние ще се занимаваме единствено с MS SQL Server.

# Проектиране на бази данни

## Модел на обектите и зависимостите (Entity/Relationship Model)

Подобно на ООП, предметната област, чийто модел ще представлява базата данни, се разделя на множества обекти (entity sets). Описват се и зависимостите между обектите.

Producer

Brand

Product

Ingredient

producer of

brand of

contains

Name

Country

Name

Description

Name

Energy value

Proteins

Fats

Carbohydrates

Name

Amount

1

n

n

1

n

m

### Множества обекти

Подобно на класовете в ООП, множествата обекти описват набор от характеристики (атрибути), присъщи на обектите от това множество.

### Зависимости едно-към-едно

На един обект от едното множество съответства най-много един обект от другото множество.

*Пример: в някое предприятие всеки отдел си има един началник и един служител може да бъде началник на не повече от един отдел.*

### Зависимости едно-към-много

На един обект от едното множество може да съответстват много обекти от другото множество. Най-често разпространените зависимости.

*Пример: един производител произвежда множество различни продукти.*

### Зависимости много-към-много

На един обект от първото множество съответстват може да съответстват много обекти от второто множество и обратно.

*Пример: един автор може да има множество написани книги и една книга може да има множество автори.*

### Атрибути

Всяко множество обекти има набор от атрибути, също както всеки клас в ООП има набор от полета (член-данни).

*Пример: множество обекти „книга” има атрибути „заглавие”, „година на издаване”, „издателство”.*

Зависимостите също могат да имат атрибути, които предоставят информация за конкретната връзка между обекти.

*Пример: тип обекти „книга” и тип обекти „стелаж” са свързани със зависимост „(книга) е картотекирана в (стелаж)”. Към зависимостта може да се прикачи атрибут „дата на картотекиране”.*

### Ключови атрибути

(Почти) всяко множество обекти има набор от ключови атрибути – това са атрибутите, които еднозначно определят всеки обект от това множество.

## Изграждане на базата данни

След като е описан модела на предметната област, изграждането на схемата на базата данни се извършва лесно.

### Множества обекти

Всяко множество обекти обикновено се превръща в една таблица, като атрибутите му описват колоните на таблицата. Ключовите атрибути описват първичния ключ на таблицата. Обектите от това множество се представят като редове в таблицата.

### Зависимости едно-към-едно и едно-към-много

Тези зависимости обикновено се осъществяват с външен ключ от таблицата, съответстваща на едното множество обекти, към таблицата, съответстваща на другото множество обекти.

### Зависимости много-към-много

Всяка зависимост много-към-много се превръща в таблица. Колоните ѝ се определят от ключовите атрибути на двете множества обекти. Свързана е с външни ключове към таблиците, съответстващи на двете множества обекти.

## Нормализиране и нормални форми

Нормализирането на една база данни цели избягването на излишни повтоврения на данни и зависимости между данните, създаващи предпоставки за аномалии.

Различните нормални форми определят набор от критерии, които дадена таблица в базата данни трябва да покрива.

### Първа нормална форма

1NF: В таблицата няма колона, която може да съдържа множества (таблици в таблицата).

### Втора нормална форма

Суперключ наричаме множество от колони на таблицата, което уникално описва всеки запис (т.е. може да бъде първичен ключ).

Минимален суперключ е такъв суперключ, че никое негово подмножество да не бъде суперключ.

Неключова колона е колона, която не участва в никой минимален суперключ на таблицата.

2NF: 1NF и не съществува минимален суперключ, за който някоя неключова колона да се определя еднозначно от негово подмножество.

### 3NF, BCNF, 4NF, 5NF

Черна магия!

# Structured Query Language (SQL)

SQL е специализиран език за програмиране, предназначен за управление и работа с данните в релационни СУБД.

Има стандартизирано описание на SQL, но всяка СУБД има свой диалект, който разширява и/или се отклонява от стандарта.

Ние ще разглеждаме диалекта Transact SQL (T-SQL), който се използва в MS SQL Server.

## Синтаксис на SQL

SQL е (предимно) декларативен език – декларират се съждения (statements), които описват някаква манипулация върху базата данни. Съжденията се разделят на три вида, които оформят трите подезика на SQL: Data Definition Language, Data Control Language и Data Manipulation Language.

SQL не разграничава главни и малки букви (за разлика от C/C++/C#/Java, подобно на Pascal/Visual Basic). По конвенция ключовите думи в езика се пишат с главни букви.

Съжденията по стандарт се разделят с точка и запетая (;), но в T-SQL тя може да се пропусне. Съжденията в T-SQL могат да се групират в блокове от съждения, разделени с GO.

### Идентификатори

Както обикновено, в повечето случаи са поредица от букви, долни черти и цифри, започващи с буква или долна черта. В допълнение може да се ползва синтаксиса ***[****идентификатор****]***, при който е разрешена употребата на интервали и спациални символи между квадратните скоби. При включена специална опция на СУБД е възможно идентификатор да бъде заграден с двойни кавички вместо с квадратни скоби.

test, TEST, TeSt, [test], [TEST], "test", "TEST" са различни записи на един и същи идентификатор.

### Запазени и стандартни думи

Съществуват много запазени и стандартни думи в езика. Всяка запазена/стандартна дума може да бъде използвана като идентификатор, като бъде заградена с квадратни скоби (или двойни кавички). Стандартните думи могат да бъдат използвани като идентификатори без квадратни скоби, когато са извън подходящия за тях контекст, но се препоръчва винаги да се заграждат за по-голяма яснота на кода.

### Променливи

Променливите се означават с незаградени идентификатори, започващи с @. Обхватът на променливите е един блок от съждения (т.е. до следващия оператор GO).

## Data Definition Language (DDL)

Съжденията в DDL служат за създаване, модификация и изтриване на таблици, изгледи, съхранени процедури, функции, тригери, индекси и потребителски типове данни. Схемата на базата данни се дефинира с помощта на DDL.

DDL съжденията започват с една от следните ключови думи: CREATE, ALTER, DROP, TRUNCATE.

### Таблици

Таблици се създават с CREATE TABLE съждения.

CREATE TABLE [Producers] (
 [ProducerID] int NOT NULL IDENTITY(1, 1),
 [Name] nvarchar(100) NOT NULL,
 [Country] nvarchar(100) NULL,

 CONSTRAINT [PK\_Producers] PRIMARY KEY ([ProducerID])
)
GO

CREATE TABLE [Brands] (
 [BrandID] int NOT NULL IDENTITY(1, 1),
 [ProducerID] int NOT NULL,
 [Name] nvarchar(100) NOT NULL,

 CONSTRAINT [PK\_Brands] PRIMARY KEY ([BrandID]),
 CONSTRAINT [FK\_Brands\_Producer] FOREIGN KEY ([ProducerID])
 REFERENCES [Producers] ([ProducerID]) ON DELETE CASCADE
)
GO

Таблици се модифицират с ALTER TABLE съждения.

ALTER TABLE [Brands] ADD
 [Description] nvarchar(MAX) NULL
GO

Таблици се изтриват с DROP TABLE съждения.

DROP TABLE [Brands]
GO

Записите в една таблица могат бързо да се изтрият, без да се изтрива самата таблица, с TRUNCATE TABLE съждение.

TRUNCATE TABLE [Brands]
GO

### Съхранени процедури

CREATE PROCEDURE [CreateProducer]
 @vName nvarchar(100), @vCountry nvarchar(100), @vProducerID int OUTPUT
AS
 INSERT INTO [Producers] ([Name], [Country])
 VALUES (@vName, @vCountry)

 SELECT @vProducerID = SCOPE\_IDENTITY()

GO

ALTER PROCEDURE, DROP PROCEDURE

### Функции

CREATE FUNCTION [GetProducerBrandsCount] (@vProducerID int) RETURNS int
BEGIN
 DECLARE @varCount int

 SELECT @varCount = COUNT(\*) FROM [Brands]
 WHERE ([ProducerID] = @vProducerID)

 RETURN @varCount
END

ALTER FUNCTION, DROP FUNCTION

### Изгледи

CREATE VIEW [BrandProducers]
AS
 SELECT [B].[BrandID], [B].[Name] AS [BrandName], [P].[Name] AS [ProducerName]
 FROM [Brands] AS [B] INNER JOIN
 [Producers] AS [P] ON ([B].[ProducerID] = [P].[ProducerID])
GO

ALTER VIEW, DROP VIEW

### Индекси

CREATE INDEX [IX\_Producers] ON [Producers] (
 [Country] ASC, [Name] ASC
)
GO

DROP INDEX [IX\_Producers] ON [Producers]
GO

## Data Control Langauge (DCL)

Съжденията в DCL служат за предоставяне и отнемане на права за достъп до обекти в базата данни на опредени потребители/групи/роли.

DCL съжденията започват с ключовите думи GRANT, DENY или REVOKE.

GRANT SELECT, INSERT, UPDATE, DELETE ON [Producers] TO [someone]
GO

DENY INSERT, UPDATE, DELETE ON [Producers] TO [someone]
GO

REVOKE DELETE ON [Producers] ТО [someone]
GO

## Data Manipulation Language (DML)

Съжденията в DML служат за записване, променяне, изтриване и извличане на данни от базата данни. В повечето случаи приложенията, използващи бази данни, използват предимно DML съждения.

DML съжденията започват с една от следните ключови думи: SELECT, INSERT, UPDATE, DELETE.

### SELECT

SELECT съжденията (заявки) се използват за извличане на данни от базата данни. Това са най-често използваните съждения в SQL.

Резултатът от изпълнението на SELECT съждение обикновено е набор от записи (rowset). Той е много подобен на таблица в базата данни – има колони (атрибути) и редове (записи).

SELECT съжденията имат следните клаузи (в този ред):

SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...

Единствено SELECT и FROM клаузите са задължителни.

Във FROM клаузата се указва таблицата (или таблиците), от която да бъдат извлечени данните.

В SELECT калузата се посочва списък от колони (или изрази), които описват колоните в резултата. Може да се използва \*, за да бъдат избрани всички колони от таблицата.

SELECT \* FROM [Producers]

SELECT [BrandID], [Name] FROM [Brands]

#### Филтриране

В WHERE клаузата се посочва критерий за отсяване на записите от таблицата. Това е израз от булев тип, като може да се състои от подизрази, съединени с булеви операции (NOT, OR, AND). Използват се стандартните оператори за сравнение (=, <>, <, <=, >, >=) както и няколко други оператори (EXISTS, IS NULL, LIKE, IN, …).

SELECT \* FROM [Products]
WHERE ([ProductID] > 10)

SELECT \* FROM [Products]
WHERE ([BrandID] = 2)

SELECT \* FROM [Brands]
WHERE ([Name] IN (N'Coca-Cola', N'Fanta', N'Sprite', N'Schweppes'))
 AND ([Description] IS NOT NULL)

Възможно е влагане на заявки:

SELECT \* FROM [Brands]
WHERE ([BrandID] NOT IN (SELECT DISTINCT [BrandID] FROM [Products]))

DISTINCT служи за премахване на излишните записи от резултата на заявката.

#### Сортировка

В ORDER BY клаузата се посочва поредица от колони, по които да бъде сортиран резултатът. Редът на колоните е от значение. След всяка колона може да се укаже посока на сортиране – ASC за възходящо и DESC за низходящо. По подразбиране сортирането е възходящо.

SELECT \* FROM [Brands]
ORDER BY [Name]

SELECT \* FROM [Products]
ORDER BY [EnergyValue] DESC, [Name] ASC

#### Агрегиране

Агрегиращите функции са функции, които се прилагат върху всеки запис от временния резултат (получен след филтрирането по критерия в WHERE клаузата) и връщат единичен резултат. Примери за агрегиращи функции са COUNT, MAX, MIN, AVG, SUM.

COUNT(...) връща броя на записите, за които стойността на израза в скобите е различна от NULL.

COUNT(\*) връща броя на всички записи във временния резултат.

SELECT COUNT([Description]) FROM [Brands]

SELECT COUNT(\*) FROM [Products]
WHERE ([BrandID] = 4)

SELECT COUNT(DISTINCT [BrandID]) FROM [Products]

#### Групиране

Групирането на записи в заявка позволява прилагането на агрегиращи функции в рамките на сходна група записи от таблицата вместо на цялата таблица.

В GROUP BY клаузата трябва да се укажат всички колони от SELECT клаузата, върху които не са приложени агрегиращи функции.

SELECT [BrandID], AVG([EnergyValue]) FROM [Products]
GROUP BY [BrandID]

В HAVING клаузата може да се укаже допълнителен критерий за филтриране на получения след GROUP BY временен резултат. Филтрирането в WHERE клаузата се случва преди групирането, а това в HAVING клаузата – след групирането.

SELECT [BrandID], COUNT(\*) FROM [Products]
GROUP BY [BrandID]
HAVING (COUNT(\*) > 1)

#### Съединения

Силата на SQL идва от възможността за комбиниране на записи от различни таблици. Това се прави с JOIN оператори (съединения) във FROM клаузата.

Различават се три вида съединения - INNER JOIN, OUTER JOIN и CROSS JOIN.

INNER JOIN комбинира всеки запис от едната таблица с един или няколко записа от другата таблица, отговарящи на определен критерий. Ако за някой от записите в едната таблица не съществува запис в другата таблица, отговарящ на критерия, то този запис не участва в резултата.

SELECT [P].[ProductID], [B].[Name] AS [BrandName], [P].[Name] AS [ProductName]
FROM [Products] AS [P] INNER JOIN
 [Brands] AS [B] ON ([P].[BrandID] = [B].[BrandID])

Ключовата дума AS се използва за задаване на псевдоними на таблици във FROM клаузата, както и за преименуване на колони в SELECT клаузата. Тъй като и в двете таблици има колона с име [Name], в SELECT клаузата сме длъжни да укажем коя от двете колони искаме да извлечем (в случая – чрез псевдонима на таблицата).

OUTER JOIN комбинира всеки запис от едната таблица с нула, един или няколко записа от другата таблица, отговарящи на определен критерий. Ако за някой от записите в едната таблица не съществува запис в другата таблица, отговарящ на критерия, то този запис все пак участва в резултата, но стойностите на колоните от втората таблица за него автоматично са NULL.

SELECT [P].[Name] AS [ProducerName], COUNT([B].[BrandID]) AS [BrandCount]
FROM [Producers] AS [P] LEFT OUTER JOIN
 [Brands] AS [B] ON ([P].[ProducerID] = [B].[ProducerID])
GROUP BY [P].[Name]

CROSS JOIN комбинира всеки запис от едната таблица с всеки запис от другата таблица. Прилага се рядко.

Поддържа се съединяване на повече от две таблици, като резултатът от съдеинението на първите две таблици се съединява с третата и т.н.

### INSERT

INSERT съжденията служат за добавяне на запис или много записи в таблица.

Има две версии на синтаксиса:

INSERT INTO ... (...) VALUES (...)

INSERT INTO ... (...) SELECT ...

След INSERT INTO се указва таблицата, в която ще се добавят записите, следвана от заграден в скоби списък от колони, чиито стойности ще бъдат попълнени.

Първата версия на синтаксиса се използва за добавяне на единичен запис в таблицата, като стойностите на съответните колони се посочват в скобите след VALUES. Стойностите трябва да пасват по брой и тип на колоните, указани след името на таблицата.

INSERT INTO [Brands] ([ProducerID], [Name])
VALUES (5, 'Lindt')

Втората версия на синтаксиса се използва за добавяне на запис или записи, получени като резултат от SELECT заявка. Колоните на заявката трябва да пасват по брой и тип на колоните, указани след името на таблицата.

INSERT INTO [Products] ([BrandID], [Name], [EnergyValue])
 SELECT 9, [Name], [Energy]
 FROM [LegacyProducts]
 WHERE ([Brand] = N'Lindt')

### UPDATE

UPDATE съжденията се използват за промяна на съществуващи записи в дадена таблица.

Синтаксисът е следния:

UPDATE ...
SET ...
WHERE ...

След UPDATE се посочва таблицата, чиито записи ще бъдат променяни. След SET се изброяват колоните, чиито стойности ще бъдат променяни и изразите, посочващи новите им стойности. WHERE клаузата се използва за отсяване на записите, които да бъдат променени (при отсъствие на WHERE клауза това са всички записи в таблицата).

UPDATE [Brands] SET [Description] = N'Швейцарско качество!'
WHERE ([Name] = N'Lindt')

UPDATE [Products] SET [EnergyValue] = [EnergyValue]\*4.184

### DELETE

DELETE съжденията служат за изтриване на записите от таблица.

Синтаксисът е следния:

DELETE FROM ...
WHERE ...

След DELETE FROM се указва таблицата, чиито записи да бъдат изтрити. WHERE клаузата се използва за фитриране на записите, които да бъдат изтрити (при липса на такава това са всички записи в таблицата).

DELETE FROM [Products]
WHERE ([EnergyValue] < 0)

## Трансакции

Трансакцията е поредица от операции, която трябва да бъде изпълнена цялостно и независимо от всички други трансакции. Използва се за гарантиране консистентността на данните в базата данни, независимо от възможността за възникване на грешка по време на поредицата от операции и предотвратявайки риска данните да бъдат променени от странични операции, изпълняващи се по същото време.

Трансакциите се характеризират със следните свойства:

* Атомарност – независимо от броя на операциите в трансакцията е разрешено изпълнението или на всички, или на нито една от операциите;
* Консистентност – независимо дали трансакцията е завъшила успешно или неуспешно, състоянието на базата данни трябва да е консистентно, т.е. отговарящо на всички ограничения (формални и неформални);
* Изолираност – трансакцията няма достъп до промените, причинени от все още незавършили трансакции; с други думи, резултатът от едновременното изпълнение на две или повече трансакции трябва да може да бъде получен и при последователното им изпълнение;
* Устойчивост – резултатът от изпълнението на трансакция трябва да бъде надеждно съхранен (независимо от следващи грешки и т.н.).

В SQL трансакциите се осъществяват с помощта на съжденията:

BEGIN TRANSACTION

COMMIT TRANSACTION

ROLLBACK TRANSACTION

# ADO.NET

ADO.NET (съкратено от *ActiveX Data Objects .NET*) е част от .NET Framework, която предоставя удобни начини за достъп до бази данни. Инфраструктурата на ADO.NET е проектирана по начин, позволяващ минимизиране зависимостта на приложенията от това каква СУБД се използва. ADO.NET предоставя няколко различни доставчика (data providers), които реализират връзката с бази данни от съвместими с тях СУБД. Ние ще разглеждаме само *.NET Framework Data Provider for SQL Server*.

ADO.NET предоставя няколко модела за достъп до база данни: *свързан (connected)*, *несвързан (disconnected)*, *LINQ to SQL* и *Entity Framework*. Ще разглеждаме само свързания модел и LINQ to SQL.

## Свързване с базата данни

Свързването с база данни през ADO.NET се осъществява на connection string. Това е символен низ, описващ параметрите на връзката под формата на двойки „параметър=стойност”, разделени с точка и запетая. Различните доставчици поддържат различни набори от параметри, но основните параметри са сходни.

### Connection strings

Server=(local); Database=HealthyFood; Integrated Security=SSPI

Server=(local); Database=HealthyFood; Integrated Security=False; User ID=sa; Password=$secret$; Persist Security Info=False

#### Server

Указва IP адрес или име на машината, където е разположена СУБД (сървъра).

#### Database

Указва името на базата данни, към която се прави връзката.

#### Integrated Security

Указва дали аутентикацията към СУБД да бъде извършена с помощта на Windows акаунта на потребителя (при стойности True или SSPI) или с помощта на потребител от СУБД с име и парола (при стойност False).

#### User ID

Указва потребителското име на потребителя от СУБД.

#### Password

Указва паролата на потребителя от СУБД.

#### Persist Security Info

Посочва дали аутентикационната информация да бъде изтрита от connection string-а след успешна аутентикация (от съображения за сигурност).

### SqlConnectionStringBuilder

SqlConnectionStringBuilder е помощен клас, който се използва за програмно построяване на connection string-ове. Параметрите са изведени като свойства на обекта.

### Съхраняване на connection string в конфигурационен файл

Препоръчва се съхраняването на connection string-овете в app.config или web.config файла на приложението с цел лесната му промяна без прекомпилиране на приложението. Конфигурационния файл има специална секция <connectionStrings>, която съхранява списък от connection string-ове. Програмно се достъпват през ConfigurationManager.ConnectionStrings.

<configuration>
 <connectionStrings>
 <add name="connection"
 providerName="System.Data.SqlClient"
 connectionString="Server=(local); Database=HealthyFood; Integrated Security=SSPI" />
 </connectionStrings>
</configuration>

connection.ConnectionString =
 ConfigurationManager.ConnectionStrings["connection"].ConnectionString;

## Свързан модел

### SqlConnection

SqlConnection е клас, който служи за осъществяване на връзката със СУБД и изпълнението на SQL операции върху базата данни. Свойството ConnectionString трябва да се запълни с валидния connection string към базата данни. Връзката се отваря с метода Open().

SqlConnection имплементира IDisposable и е строго препоръчително да се използва в using конструкция с цел правилно затваряне и озвобождаване на връзката след приключване на работата с нея.

using (var connection = new SqlConnection())
{
 connection.ConnectionString =
 ConfigurationManager.ConnectionStrings["connection"].ConnectionString;
 connection.Open();

 // операции върху базата...
}

### SqlCommand

SqlCommand е клас, служещ за изпълнение на SQL операция (или операции) върху базата данни. Той се свързва с SqlConnection обект (през конструктора или свойството Connection). SQL заявката се описва в свойството CommandText. Изпълнението се извършва с един от методите ExecuteNonQuery(), ExecuteScalar() или ExecuteReader(). SqlCommand имплементира IDisposable.

using (SqlCommand command = connection.CreateCommand())
{
 command.CommandText = "SELECT COUNT(\*) FROM [Producers]";
 int producerCount = (int)command.ExecuteScalar();
 Console.WriteLine("Producers count: {0}", producerCount);
}

ExecuteScalar() връща единствен (скаларен) резултат.

ExecuteNonQuery() връща единствено броя на променените от заявката записи.

ExecuteReader() връща обект от тип DbDataReader, който се използва за четене на набора от записи, получен от изпълнението на заявката. DbDataReader също имплементира IDisposable.

using (SqlCommand command = connection.CreateCommand())
{
 command.CommandText = "SELECT [ProducerID], [Name] FROM [Producers]";

 using (DbDataReader reader = command.ExecuteReader())
 {
 while (reader.Read())
 {
 Console.WriteLine("ProducerID: {0}; Name: {1}",
 reader["ProducerID"],
 reader["Name"]);
 }
 }
}

### Параметри

Параметрите се използват за предаване на динамични стойности за части от SQL заявката. Параметрите имат име, тип и стойност. Името на всеки параметър започва с @. Всеки SqlCommand обект има свойство Parameters, представляващо колекция от параметри.

Console.Write("Input producer ID: ");
int producerID = Convert.ToInt32(Console.ReadLine());

using (SqlCommand command = connection.CreateCommand())
{
 command.CommandText =
@"SELECT [BrandID], [Name] FROM [Brands]
 WHERE ([ProducerID] = @producer\_id)";
 command.Parameters.AddWithValue("@producer\_id", producerID);

 using (DbDataReader reader = command.ExecuteReader())
 {
 while (reader.Read())
 {
 Console.WriteLine("BrandID: {0}; Name: {1}",
 reader["BrandID"],
 reader["Name"]);
 }
 }
}

### SQL injection атаки

SQL injection атаките представляват експлоатиране на слаби места в заявките от страна на потребител на приложението с цел нарушаване сигурността на базата данни. Такава атака е възможна, когато SQL заявката се построява динамично чрез конкатениране на стойностите, въведени от потребителя, вместо тези стойности да бъдат предадени през параметри. Строго препоръчително е винаги да се използват параметри!

Console.WriteLine("Input brand name to search for: ");
string name = Console.ReadLine();

using (SqlCommand command = connection.CreateCommand())
{
 command.CommandText =
@"SELECT [BrandID], [Name] FROM [Brands]
 WHERE ([Name] = N'" + name + "')";

 using (DbDataReader reader = command.ExecuteReader())
 {
 while (reader.Read())
 {
 Console.WriteLine("BrandID: {0}; Name: {1}",
 reader["BrandID"],
 reader["Name"]);
 }
 }
}

Ако потребителят въведе следния текст:

'); DELETE FROM [Products];--

текста на заяката ще бъде:

SELECT [ProductID], [Name] FROM [Products]
WHERE ([Name] = N''); DELETE FROM [Products];--')

Двете тирета означават начало на коментар, продължаващ до края на реда. Това прави заявката валидна синтактично и цялото съдържание на таблицата ще бъде изтрито!

Правилна имплементация:

Console.WriteLine("Input brand name to search for: ");
string name = Console.ReadLine();

using (SqlCommand command = connection.CreateCommand())
{
 command.CommandText =
@"SELECT [BrandID], [Name] FROM [Brands]
 WHERE ([Name] = @name)";
 command.Parameters.AddWithValue("@name", name);

 using (DbDataReader reader = command.ExecuteReader())
 {
 while (reader.Read())
 {
 Console.WriteLine("BrandID: {0}; Name: {1}",
 reader["BrandID"],
 reader["Name"]);
 }
 }
}

### Съхранени процедури

SqlCommand позволява и изпълнение на съхранени процедури в базата данни. За целта свойството CommandType се променя на CommandType.StoredProcedure, а името на съхранената процедура се записва в CommandText. Стойностите на параметрите се предават през колекцията Parameters.

using (SqlCommand command = connection.CreateCommand())
{
 command.CommandType = CommandType.StoredProcedure;
 command.CommandText = "[CreateProduct]";
 command.Parameters.AddWithValue("@vBrandID", 5);
 command.Parameters.AddWithValue("@vName", "Шоколад Milka алпийско мляко");
 command.Parameters.AddWithValue("@vEnergyValue", (decimal)532);
 command.Parameters.AddWithValue("@vProteins", (decimal)6.8);
 command.Parameters.AddWithValue("@vFats", (decimal)29.6);
 command.Parameters.AddWithValue("@vCarbohydrates", (decimal)58);
 command.Parameters.Add("@vProductID", SqlDbType.Int)
 .Direction = ParameterDirection.Output;

 command.ExecuteNonQuery();
 Console.WriteLine("Created product ID: {0}",
 (int)command.Parameters["@vProductID"].Value);
}

### Трансакции

Трансакциите в ADO.NET се правят най-често с помощта на класа TransactionScope. Той имплементира IDisposable и обикновено се използва в using блок. Обхвата на трансакцията започва от създаването на TransactionScope обекта и завършва при неговото освобождаване (т.е. продължава от началото до края на using блока). Ако преди освобождаването на TransactionScope обекта се извика неговия метод Complete(), трансакцията се счита за успешна и автоматично се прави COMMIT при освобождаването на обекта. В противен случай трансакцията е неуспешна и се прави ROLLBACK.

using (var ts = new TransactionScope())
using (var connection = new SqlConnection())
{
 connection.ConnectionString =
 ConfigurationManager.ConnectionStrings["connection"].ConnectionString;
 connection.Open();

 int productID;

 using (SqlCommand command = connection.CreateCommand())
 {
 command.CommandType = CommandType.StoredProcedure;
 command.CommandText = "[CreateProduct]";
 command.Parameters.AddWithValue("@vBrandID", 5);
 command.Parameters.AddWithValue("@vName", "Шоколад Milka лешници");
 command.Parameters.AddWithValue("@vEnergyValue", (decimal)544);
 command.Parameters.AddWithValue("@vProteins", (decimal)7.6);
 command.Parameters.AddWithValue("@vFats", (decimal)32.4);
 command.Parameters.AddWithValue("@vCarbohydrates", (decimal)54);
 command.Parameters.Add("@vProductID", SqlDbType.Int)
 .Direction = ParameterDirection.Output;

 command.ExecuteNonQuery();
 productID = (int)command.Parameters["@vProductID"].Value;
 Console.WriteLine("Created product ID: {0}", productID);
 }

 using (SqlCommand command = connection.CreateCommand())
 {
 command.CommandType = CommandType.StoredProcedure;
 command.CommandText = "[AddProductIngredient]";
 command.Parameters.AddWithValue("@vProductID", productID);
 command.Parameters.AddWithValue("@vIngredientID", 4);
 command.Parameters.AddWithValue("@vAmount", DBNull.Value);

 command.ExecuteNonQuery();
 Console.WriteLine("Ingredient added to product");
 }

 ts.Complete();
}

# Language INtegrated Query (LINQ)

## Лирическо отклонение – функционално програмиране

### Императивни езици

C, C++, C#, Java, Pascal, Visual Basic…

Програмите в императивните езици за програмиране представляват поредици от инструкции към процесора/интерпретатора/компилатора (наричат се още *съждения*). Съжданията зависят едно от друго индиректно – чрез стойности, записани в паметта на машината (променливи).

### Функционални езици

Scheme, Haskell, F#, Scala…

Функционалните езици спадат към декларативните езици за програмиране.

Програмите във функционалните езици за програмиране представляват декларации на множество функции. Всяка функция се характеризира с набор от входни параметри, резултат и някаква формална зависимост между входните параметри и резултата. Тази зависимост се представя под формата на *израз*. Това прави функциите във функционалните езици много сходни с функциите в математиката и позволява лесното превеждане на формални алгоритми в програмен вид.

### Пример 1: Умножение с константа

Имеративна версия:

function MulBy2(x)
{
 return x\*2;
}

Функционална версия:

function MulBy2(x) = x\*2;

### Пример 2: Умножение на всички елементи на списък с константа

Императивна версия:

function MulElementsBy2(list)
{
 result = EmptyList();
 foreach (x in list)
 {
 AddElement(x\*2, list);
 }
 return result;
}

Функционална версия:

function MulElementsBy2(list) = Map(MulBy2, list);

Map(f, list) е функция с два аргумента: функция и списък. Map прилага функцията f върху всеки елемент от списъка и връща нов списък с резултатите.

### Пример 3: Отсяване на елементи от списък

Императивна версия:

function GetPositiveElements(list)
{
 result = EmptyList();
 foreach (x in list)
 {
 if (x > 0)
 {
 AddElement(x, list);
 }
 }
 return result;
}

Функционална версия:

function IsPositive(x) = x > 0;
function GetPositiveElements(list) = Filter(IsPositive, list);

Filter(f, list) е функция с два аргумента: функция, която връща булев резултат, и списък. Filter прилага функцията f върху всеки елемент от списъка и връща нов списък, съдъжащ само елементите, за които f връща стойност true.

### Пример 4: Намиране сумата на елементите на списък

Императивна версия:

function Sum(list)
{
 result = 0;
 foreach (x in list)
 {
 result += x;
 }
 return result;
}

Функционална версия:

function Add(x, y) = x + y;
function Sum(list) = Fold(Add, 0, list);

Fold(f, n, list) е функция с три аргумента: функция с два агумента, начална стойност и списък. Fold агрегира елементите на списъка, използвайки функцията f и започвайки от началната стойност n, като накрая връща резултата от агрегирането.

### Пример 5: Намиране средно аритметично на елементите на списък

Императивна версия:

function Average(list)
{
 sum = 0;
 count = 0;
 foreach (x in list)
 {
 sum += x;
 count++;
 }
 return sum/count;
}

Функционална версия:

function Add1(x, y) = x + 1;
function Count(list) = Fold(Add1, 0, list);
function Average(list) = Sum(list)/Count(list);

## Разширителни методи на IEnumerable<T>

В .NET Framework 3.5 са добавени множество разширителни методи на IEnumerable<T>, които следват функционалната парадигма. Те са дефинирани в пространството от имена System.Linq.

IEnumerable<U> Select<T, U>(this IEnumerable<T> source, Func<T, U> selector)

Методът Select е аналогичен на Map от функционалните примери.

IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate)

Методът Where е аналогичен на Filter от функционалните примери.

U Aggregate<T, U>(this IEnumerable<T> source, U seed, Func<U, T, U> function)

Методът Aggregate е аналогичен на Fold от функционалните примери.

IEnumerable<T> Skip<T>(this IEnumerable<T> source, int count)

Методът Skip връща колекцията без пъвите count елемента.

IEnumerable<T> Take<T>(this IEnumerable<T> source, int count)

Методът Таке връща първите count елемента от колекцията.

First, Last, FirstOrDefault, LastOrDefault

Sum, Min, Max, Average, Count

ToList, ToArray, ToDictionary

### Отложено изпълнение

Методите, които връщат IEnumerable<T>, са с отложено изпълнение. Те не извършват посочената операция при извикването си, а вместо това връщат обект, описващ операцията, която трябва да се извърши. Когато върху резултата се приложи метод, който не е с отложено изпълнение, целият записан набор от отложени операции се изпълнява и методът се изпълнява върху реалния резултат.

var numbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

IEnumerable<int> newNumbers = numbers
 .Where(x => x % 2 == 0)
 .Select(x => x\*2);

// в този момент newNumbers съхранява информация за операциите,
// които трябва да се извършат

numbers.Add(10);

// точно преди началото на обхождането на newNumbers с foreach,
// операциите се изпълняват и резултатът се подава на цикъла
foreach (int n in newNumbers)
{
 Console.WriteLine(n);
}

### Сортировка

IOrderedEnumerable<T> OrderBy<T, U>(this IEnumerable<T> source,
 Func<T, U> keySelector)

Методът OrderBy сортира несортирана колекция по някакъв ключ.

IOrderedEnumerable<T> ThenBy<T, U>(this IOrderedEnumerable<T> source,
 Func<T, U> keySelector)

Методът ThenBy сортира вече сортирана по един или повече ключове колекция по още един ключ.

OrderByDescending, ThenByDescending

### Групиране

IEnumerable<IGrouping<U, T>> GroupBy<T, U>(this IEnumerable<T> source,
 Func<T, U> keySelector)

Методът GroupBy групира елементите на колекцията по някакъв ключ, подобно на GROUP BY в SQL. Резултатът е колекция от групирания, всяко от което се характеризира със стойност на ключа и колекция от елементи, чийто ключ съвпада с тази стойност.

### Съединения

IEnumerable<W> Join<T, U, V, W>(this IEnumerable<T> first, IEnumerable<U> second,
 Func<T, V> firstKeySelector,
 Func<U, V> secondKeySelector,
 Func<T, U, W> resultSelector)

Методът Join е сходен с INNER JOIN от SQL. Той съединява две колекции (first и second) по равенство на ключовете (зададени с firstKeySelector и secondKeySelector). Върху всяка получена двойка елементи от първата и втората колекция се прилага resultSelector и резултатът се добавя в резултатната колекция.

IEnumerable<W> GroupJoin<T, U, V, W>(this IEnumerable<T> first, IEnumerable<U> second,
 Func<T, V> firstKeySelector,
 Func<U, V> secondKeySelector,
 Func<T, IEnumerable<U>, W> resultSelector)

Методът GroupJoin е сходен с OUTER JOIN от SQL. Той съединява две колекции (first и second) по равенство на ключовете (зададени с firstKeySelector и secondKeySelector). За всеки елемент от първата колекция се получава двойка от него и колекция от съединени елементи от втората колекция. Върху всяка получена двойка се прилага resultSelector и резултатът се добавя в резултатната колекция.

## LINQ

*Language INtegrated Query* е специален синтаксис в C# (и други .NET езици), въведен в .NET Framework 3.5. Той наподобява синтаксиса на SELECT заявките в SQL. Компилаторът автоматично трансформира LINQ в извиквания на описаните по-горе разширителни методи. За да се използва LINQ, трябва да се реферира пространството от имена System.Linq.

var example1 = from n in numbers
 select n\*2;

var example1 = numbers.Select(n => n\*2);

var example2 = from n in numbers
 where n > 15
 orderby n % 4, n % 7 descending
 select n;

var example2 = numbers
 .Where(n => n > 15)
 .OrderBy(n => n % 4)
 .ThenByDescending(n => n % 7);

var example3 = from n in numbers
 group n by n % 3 into ng
 where ng.Count() == 2
 select ng;

var example3 = numbers
 .GroupBy(n => n % 3)
 .Where(ng => ng.Count() == 2);

var example4 = from n in numbers
 join m in numbers on n % 3 equals m % 3
 select Tuple.Create(n, m);

var example4 = numbers
 .Join(numbers, n => n % 3, m => m % 3, (n, m) => Tuple.Create(n, m));

## Други имплементации на LINQ

Показаните до момента примери бяха от *LINQ to Objects*. Това са заявки към произволни обекти, имплементиращи IEnumerable<T>.

Интерфейсът IQueryable<T> притежава голяма част от разширителните методи на IEnumerable<T> и също позволява прилагането на LINQ заявки към обектите, които го имплементират.

Разликата между двата интерфейса е във вътрешния механизъм на изпълнението на отложените операции. Докато разширителните методи на IEnumerable<T> имат твърдо посочен начин на изпълнение на операциите върху колекцията, разширителните методи на IQueryable<T> позволяват трансформиране на операциите в някаква друга форма, например XPath заявка към XML документ или SQL заявка към база данни.

.NET Framework 3.5 предоставя няколко имплементации на IQueryable<T> с различно предназначение:

* *LINQ to XML*
* *LINQ to DataSet*
* *LINQ to SQL*
* *LINQ to Entities*

Ние ще разглеждаме по-подробно *LINQ to SQL*.

# LINQ to SQL

LINQ to SQL е част от ADO.NET, която позволява лесно манипулиране с бази данни през LINQ заявки. В основата на LINQ to SQL лежи Object-Relational Mapping (ORM) – модел, асоцииращ таблици от базата данни в класове от кода на приложението. Този модел може да се дефинира по два начина: ръчно (чрез атрибути в кода) или през DBML документ (специализиран XML документ), от който след това автоматично да се генерират класове.

За да се използва LINQ to SQL, проектът трябва да реферира библиотеката System.Data.Linq.

LINQ to SQL може да се използва единствено с SqlClient доставчика от ADO.NET.

## Ръчно описване на ORM модела

Атрибутите, служещи за описание на ORM модела в LINQ to SQL се намират в пространството от имена System.Data.Linq.Mapping.

### Таблици

Атрибутът Table се използва за маркиране на клас като асоцииран с таблица в базата данни (по подразбиране името на таблицата съвпада с името на класа).

Атрибутът Column се използва за да се укаже, че свойство или поле на класа трябва да се асоциира с колона от таблицата (по подразбиране името на колоната съвпада с името на свойството/полето). Свойството IsPrimaryKey на атрибута Column служи за маркиране на колона, участваща в първичния ключ на таблицата, а свойството CanBeNull указва дали колоната разрешава NULL стойности (по подразбиране true).

[Table(Name = "Producers")]
public class Producer
{
 [Column(IsPrimaryKey = true)]
 public int ProducerID { get; set; }

 [Column(CanBeNull = false)]
 public string Name { get; set; }

 [Column]
 public string Country { get; set; }
}

### Връзки

Описването на връзките между таблици в базата, реализирани с помощта на външни ключове, се прави с помощта на атрибута Association. Той се използва върху свойство с тип T или EntitySet<T>, където T също е асоцииран с таблица клас. Свойствата ThisKey и OtherKey на атрибута Association служи за указване името на колоната от таблицата с външния ключ, която участва във външния ключ.

[Table(Name = "Producers")]
public class Producer
{
 private EntitySet<Brand> \_brands;

 [Column(IsPrimaryKey = true)]
 public int ProducerID { get; set; }

 [Column(CanBeNull = false)]
 public string Name { get; set; }

 [Column]
 public string Country { get; set; }

 [Association(Storage = "\_brands", OtherKey = "ProducerID")]
 public EntitySet<Brand> Brands
 {
 get { return \_brands; }
 set { \_brands.Assign(value); }
 }
}

[Table(Name = "Brands")]
public class Brand
{
 private EntityRef<Producer> \_producer;

 [Column(IsPrimaryKey = true)]
 public int BrandID { get; set; }

 [Column]
 public int ProducerID { get; set; }

 [Column(CanBeNull = false)]
 public string Name { get; set; }

 [Column]
 public string Description { get; set; }

 [Association(Storage = "\_producer", ThisKey = "ProducerID")]
 public Producer Producer
 {
 get { return \_producer.Entity; }
 set { \_producer.Entity = value; }
 }
}

### Контекст

Контекст се нарича основния обект, през който се осъществява достъпът до базата данни. Класът му е DataContext или негов наследен клас.

public class HealthyFoodContext : DataContext
{
 public HealthyFoodContext(string connectionString)
 : base(connectionString) { }

 public Table<Producer> Producers
 {
 get { return GetTable<Producer>(); }
 }

 public Table<Brand> Brands
 {
 get { return GetTable<Brand>(); }
 }
}

string connectionString =
 ConfigurationManager.ConnectionStrings["connection"].ConnectionString;
var context = new HealthyFoodContext(connectionString);

### Пример 1: Вземане на всички марки

IEnumerable<Brand> brands = from b in context.Brands
 select b;

### Пример 2: Вземане на част от марките

IEnumerable<Brand> brands = from b in context.Brands
 where b.BrandID > 3
 select b;

### Пример 3: Вземане на част от свойствата на марките

var brands = from b in context.Brands
 select new { BrandID = b.BrandID, BrandName = b.Name };

### Пример 4: Вземане на производителите на марките

var brands = from b in context.Brands
 select new { ProducerName = b.Producer.Name, BrandName = b.Name };

### Пример 5: Вземане на марките на част от производителите

IEnumerable<Brand> brands = context.Producers
 .Where(p => p.Name != "Kraft Foods")
 .SelectMany(p => p.Brands);

### Пример 6: Вземане на единичен производител

Producer producer = context.Producers.Single(p => p.ProducerID == 3);

## Описване на ORM модела с DBML

Visual Studio предоставя възможност за описване на ORM модела с DBML под формата на визуална диаграма от класове и връзки.

За целта се добавя нов *LINQ to SQL Classes* файл към проекта. Визуалният му редактор позволява добавяне на класове, описване на свойствата им и връзките между класовете, както и подробно настройване на атрибутите.

След това класовете за достъп до данните се генерират автоматично на базата на описания DBML. Тъй като получения CS файл се генерира наново при всяка промяна на DBML файла, ръчни добавки в него не трябва да се правят. Вместо това за добавяне на допълнителни свойства и методи към класовете може да се използва втори файл и те да се маркират като partial класове.

## Модифициране на данните през LINQ to SQL

### Добавяне на запис

Добавянето на запис в базата данни се осъществява просто чрез добавяне на нов обект в таблицата на класа му, или в някое релационна колекция.

Brand brand = context.Brands.Single(b => b.BrandID == 5);
var product = new Product
{
 Name = "Шоколад Milka алпийско мляко",
 EnergyValue = 532,
};
brand.Products.Add(product);

### Редактиране на запис

Редактирането на запис се осъществява чрез редакция на свойствата на обекта.

Product product = context.Products.Single(p => p.ProductID == 19);
product.Proteins = (decimal)6.8;
product.Fats = (decimal)29.6;
product.Carbohydrates = (decimal)58;

### Изтриване на запис

Изтриването на запис се осъществява чрез премахването на обекта от таблицата на класа му.

Product product = context.Products.Single(p => p.ProductID == 19);
context.Products.DeleteOnSubmit(product);

### Записване на промените в базата данни

Всички направени до момента промени се запазват в базата данни чрез метода SubmitChages() на DataContext обекта. Възможно е той да хвърли изключение, ако има конфликт при запазването на данните в базата (например поради промяната им от друг клиент).

context.SubmitChanges();