Platforma .NET

Radek Janoštík

KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Databáze – úvod

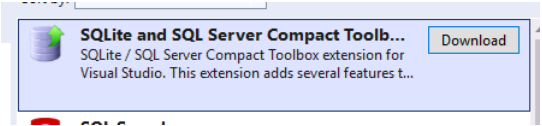
- = (ve stručnosti) persistentní úložiště dat
- Neřešíme konkrétní implementaci databáze, zajímá nás rozhraní
- SQL(Structured Query Language) – dotazovací jazyk, různé mutace
- Zde v kurzu – MSSQL, SQLite(Linux, MacOS)
- Data v tabulkách, každý sloupec pevný datový typ
- Řádky $\approx$ objekty, Sloupce $\approx$ vlastnosti objektů
- Více v kurzu databází – dnes jen zjednodušený pohled z C#

Databáze – vytvoření

- MSSQL Express – zdarma (určitý limit paměti, jader)
- Správa – MS SQL Server Management Studio
- Možnost testovat na `ts.inf.upol.cz` – připojení pomocí vzdálené plochy
- Integrace ve Visual Studiu – Server Explorer→Database Connection
- Alternativa MS SQL Server Database File
  - Databáze v souboru .mdf
  - Snadno přenositelné
  - Pro nás postačující
- Pro připojení k DB slouží tzv. Connection string (k nahlédnutí v Properties)

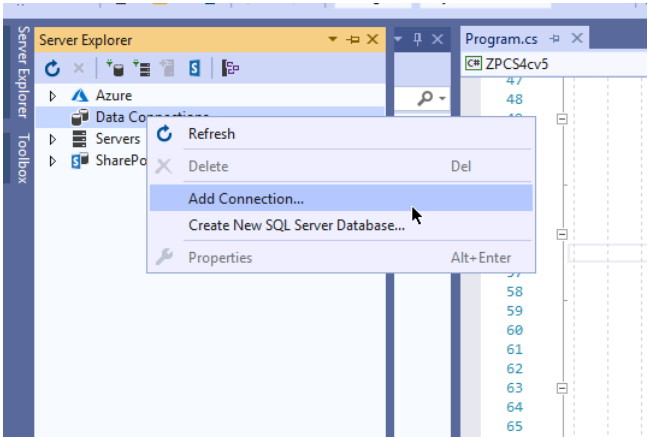
SQLite – VisualStudio

- Podpora SQLite ve VisualStudiu není plnohodnotná
  - Nejde třeba vytvářet a spouštět libovolné dotazy
  - Nutno doinstalovat SQLite and SQL Server Compact Toolbox

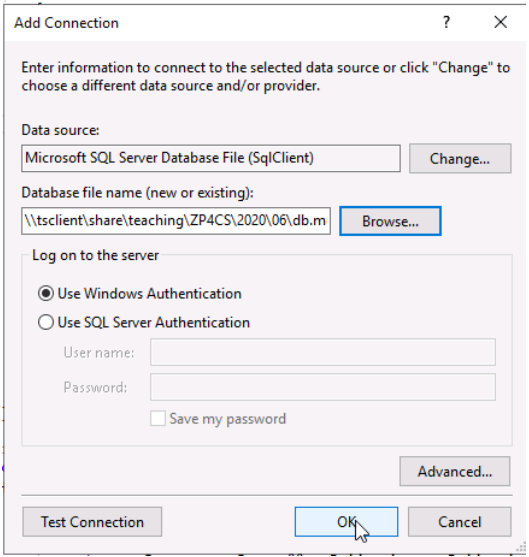


- V ServerExplorer se poté objeví ikonka rozšíření
- V Rider podpora lepší

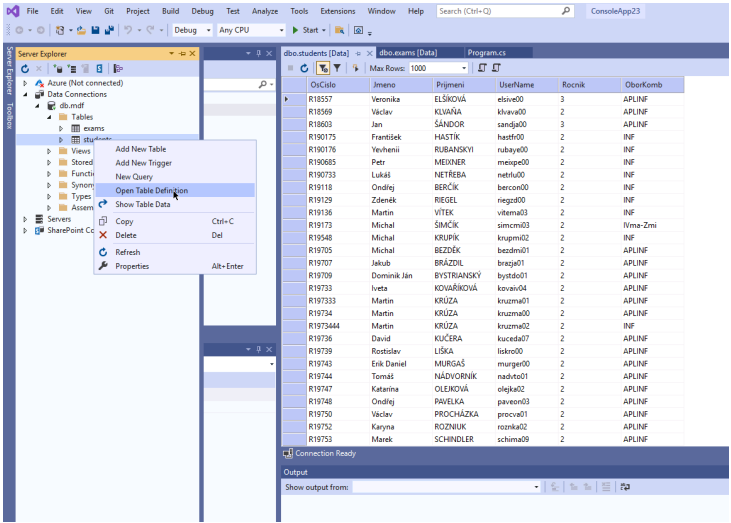
Připojení existující databáze ze souboru



Zvolíme soubor s databází



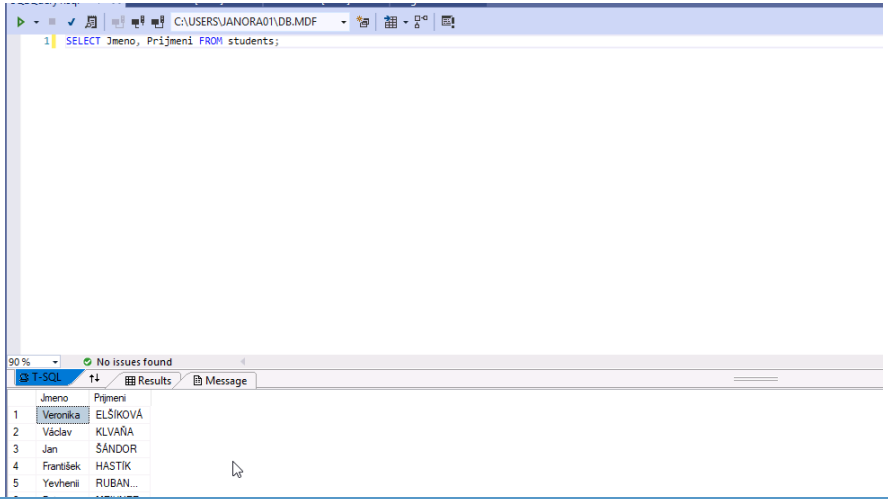
Prohlížení DB



- Můžeme se podívat na data v tabulce nebo zobrazit její definici.

Spouštění SQL dotazů

- Pravým tlačítkem myši na databázi nebo tabulku ⇒ New Query
- Tlačítkem „Play“ dotaz odešleme na server



Jazyk SQL (1/3) – Rychlé opakování

- Výběr dat

```
1 SELECT sloupec1, sloupec2 FROM tabulka WHERE podminka;
```

- Např.:

```
1 SELECT name, surname FROM students WHERE year=3;
2 SELECT * FROM subjects WHERE name = 'Jan';
```

- Řazení

```
1 SELECT name, surname FROM students ORDER BY surname DESC;
2 SELECT * FROM subjects WHERE obor = 'KMI' ORDER BY year ASC;
```

- Pozor – porovnání je pouze $=$ místo $==$

Jazyk SQL (2/3) – Rychlé opakování

- Editace záznamu

```
1 UPDATE tabulka SET sloupec1=hodnota1, sloupec2=hodnota2 WHERE
      podminka;
```

- Např.:

```
1 UPDATE students SET year=3, obor = 'MI' WHERE year=3;
2 UPDATE subjects SET name='XXXXXX';
```

- Vložení záznamu

```
1 INSERT INTO tabulka (sloupec1, sloupec2, ...) VALUES (hodnota1,
      hodnota2, ...);
```

- Např.:

```
1 INSERT INTO students (name, surname) VALUES ('Petr', 'Novak');
```

Jazyk SQL (3/3) – Rychlé opakování

- Smazání záznamu

```
1 DELETE FROM tabulka WHERE podminka;
```

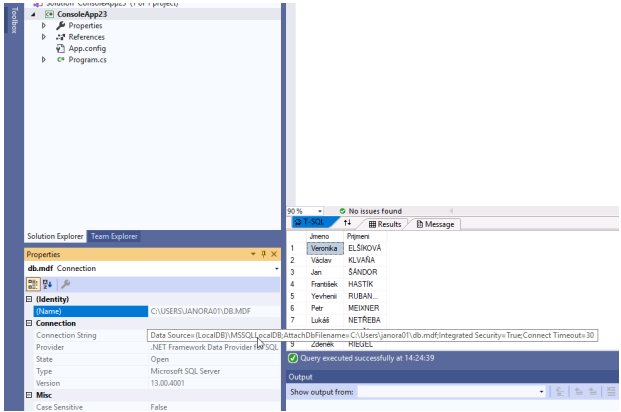
- Např.:

```
1 DELETE FROM students WHERE id=35;
2 DELETE FROM students;
```

- Další dotazy (`GROUP BY`, `CREATE TABLE`, `JOIN`,...) viz. dokumentace
- `https://technet.microsoft.com/en-us/library/ms189826(v=sql.90).aspx`

Zjištění connection stringu

- Abychom se mohli k DB připojit z kódu, potřebujeme tzv. `Connection String`
- Najdeme jej v `Properties` databáze:



Připojení DB v C#

- Namespace `Microsoft.Data.SqlClient`
- Pro SQLite `Microsoft.Data.Sqlite` https://learn.microsoft.com/cs-cz/dotnet/standard/data/sqlite/?tabs=netcore-cli
  - Možná nutné stáhnout přes NuGet
- Připojení a čtení výsledků dotazu
- Vytvoříme si `SqlConnection`, otevřeme spojení
- Poté vytvoříme příkaz`SqlCommand` a vytvoříme reader, ze kterého přečteme všechna data
- Data indexována podle pořadí sloupců v definici tabulky

Dotaz s parametry

```
1  try {
2    using (SqlConnection conn = new SqlConnection(connectionString))
3    {
4      conn.Open();
5      SqlCommand command = new SqlCommand("SELECT * FROM students;",
           conn);
6      using (SqlDataReader dr = command.ExecuteReader())
7      {
8        while (dr.Read())
9        {
10         Console.WriteLine($"{dr[0]}, {dr[1]}, {dr[2]}");
11       }
12     }
13   }
14 } catch (Exception e) {
15 ...
16 }
```

Dotaz s parametry

- Proč je špatně:

```
string someName <- nacteme data od uzivatele
string someSurname <- nacteme data od uzivatele
SqlCommand command = new SqlCommand($"SELECT * FROM students
        WHERE name='{someName}' OR surname='{someSurname}';",
            conn);
```

- A správně

```
SqlCommand command = new SqlCommand("SELECT * FROM students
        WHERE name = @someParam OR surname=@surname", conn);
command.Parameters.Add(new SqlParameter("someParam", someName));
command.Parameters.Add(new SqlParameter("surname", someSurname));
```

Dotaz s parametry

- Proč je špatně:

```
SqlCommand command = new SqlCommand($"SELECT * FROM students
        WHERE name='{someName}' OR surname='{someSurname}';",
            conn);
```

- A správně

```
SqlCommand command = new SqlCommand("SELECT * FROM students
        WHERE name = @someParam OR surname=@surname", conn);
command.Parameters.Add(new SqlParameter("someParam", someName));
command.Parameters.Add(new SqlParameter("surname", someSurname));
```

- Odpověď: nikdy bychom neměli vkládat do dotazů surový obsah proměnných. Zvláště, pokud je může modifikovat uživatel aplikace.
- Parametrické dotazy pomocí `@` pomůžou escapovat obsahy proměnných – zamezení SQL Injection útoku
- Viz `https://www.w3schools.com/sql/sql_injection.asp`

Vyhodnocení dotazu bez výsledků

- Metoda `ExecuteNonQuery()` – vrací počet ovlivněných prvků
- Lze volat asynchronně
- Vložení prvku

```
1 SqlCommand command = new SqlCommand("INSERT INTO students (id,
      name, surname) VALUES (@id, @name, @surname);", conn);
2 command.Parameters.Add(new SqlParameter("id", 3));
3 command.Parameters.Add(new SqlParameter("name", "Alois"));
4 command.Parameters.Add(new SqlParameter("surname", "Fridrich"));
5 int affected = command.ExecuteNonQuery();
```

- Vymazání prvku(ů)

```
1 SqlCommand command = new SqlCommand("DELETE FROM students WHERE
      surname=@srn", conn);
2 command.Parameters.Add(new SqlParameter("srn", "SOUKUP"));
3 int affected = command.ExecuteNonQuery();
4 Console.WriteLine("Affected: " + affected);
```

Reflexe

- = proces získávání znalostí o *typech* za běhu programu
- $\Rightarrow$ získávání metadat „o kódu“
- Objektově zapouzdřené
- Budeme používat funkcionalitu z namespace `System.reflection`
- Zaměříme se pouze na získávání metadat o objektech, metodách, atributech

Reflexe – typy

- Nejdříve potřebujeme získat informace o typu objektů
- Máme-li (můžeme-li vytvořit) instanci:

```
1 Custommer c = new Custommer() { ... };
2 Type t = c.GetType();
```

- Nemáme-li (nechceme vytvářet) instanci:

```
1 Type t = typeof(Custommer);
```

- Získání typu z řetězce: `Type t = Type.GetType("NejakaTrida");`
- Velké množství `true/false` hodnot – např.:`IsAbstract`, `IsArray`, `IsClass`, `IsCOMObject`, `IsEnum`, `IsInterface`, `IsPrimitive`, `IsNestedPrivate`, `IsNestedPublic`, `IsSealed`, `IsValueType`

Reflexe – typy – informace

- Získání informací o metodách – `GetMethods()`
  - Vrátí pole `MethodInfo`, ze kterého získáme veškeré informace
  - Jméno, typy parametrů, návratová hodnota, přístupnost, ...

```
1 MethodInfo[] mi = t.GetMethods();
2 foreach (MethodInfo m in mi) {
3   Console.WriteLine($"Metoda: {m.Name} vraci {m.ReturnType}");
4 }
```

- Získání informací o slotech

```
1 FieldInfo[] fis = t.GetFields();
2 foreach (FieldInfo fi in fis) {
3   Console.WriteLine(fi.Name);
4 }
```

- Obdobně `GetProperties()`, `GetInterfaces()`
- Obdobně `m.GetParameters()` pro parametry metod

Atributy

- Předchozí metadata generoval kompilátor, měli jsme je přístupné
- Atributy = způsob, jak dovolit programátorům poskytovat metadata o kódu
  - Tato metadata pak budou přístupná „programu“ skrze reflexi
- S atributy jsme s potkali z uživatelského pohledu
- Atributy = anotace k danému typu, (třídy, rozhraní, struktury, . . .) a jejich prvkům (vlastnosti, metody, sloty, . . .)
- Již jsme se mohli setkat s

```
[HttpGet], [HttpPost], [Serializable], [PrimaryKey], ...
```

Atributy – čtení

- Čtení atributů pomocí metody `GetCustomAttributes()`

```
FieldInfo[] fis = t.GetFields(BindingFlags.NonPublic |
                    BindingFlags.Instance);
foreach (FieldInfo i in fis) {
  Console.WriteLine(i.Name);
  object[] atts = i.GetCustomAttributes(false);
  foreach (object att in atts) {
    Console.WriteLine(att.GetType());
  }
}
```

- Případně kontrola na daný atribut

```
if (m.GetCustomAttributes(typeof(ThreadStaticAttribute), true)
                                        .Length > 0)) {
// metoda ma atribut
}
```

Atributy

- Atributy samy o sobě „nedělají nic“, pouze dodávají informace
- Vždy záleží na tom, kdo je zpracuje a co s tím udělá
  - Nějaký kód přes reflexi zjistí atributy (a jejich vlastnosti)
  - Na základě těchto získaných dat může manipulovat s objektem
  - Např. `[Serializable]` – Víme, že třída lze serializovat
  - `[NonSerialized]` – tento slot se serializovat nemá
- Velmi často nám jako jedinná informace postačí název atributu
  - Ale atributy mohou mít vlastnosti
  - Nastavovatelné přes konstruktur
  - Mohou mít i metody

Vytvoření vlastních atributů

- Pojmenovávací konvence – suffix `Attribute`
  - Suffix není potřeba uvádět do hrantých závorek, pouze první část jména
- Atributu z bezpečnostního hlediska vytvoříme jako `sealed`
- Budeme dědit ze `System.Attribute`

```
1 public sealed class MyAttribute : Attribute
2 {
3 }
```

- Omezení použití atributu – pomocí atributu `AttributeUsage`
  - Parametrem je maska z enumu: `AttributeTargets`

```
1 [AttributeUsage(AttributeTargets.Class | AttributeTargets.Field)]
2 public sealed class MyAttribute : Attribute
3 {
4 }
```

Vytvoření vlastních atributů

- Atribut je tedy úplně normální třída
- Ale měl by být používán pouze jako nosič dat

```
1  [AttributeUsage(AttributeTargets.Class)]
2  public sealed class MyORMContextAttribute : Attribute {
3      public Type[] types;
4      public string someData;
5      public int version;
6      public MyORMContextAttribute(Type[] types,
7                 string someData, int version) {
8        this.types = types;
9        this.someData = someData;
10       this.version = version;
11     }
12 }
```

Úskalí a doporučení k reflexi

- Reflexi bychom měli používat pro věci, které nejdou udělat jinak
  - Např. bychom přes ni neměli volat běžně metody
- Snižuje výkon – VM nemůže dělat tolik optimalizací
- Bezpečnost – v některých prostředích může být zakázaná
- Lze získat hodnoty a volat *non-public* členy tříd

Object Relation Mapping (ORM)

- Automatické mapovaní objektů do relační databáze
- Náš pohled – transparentní práce s databází pomocí objektů
- Odstínění od konkrétní implementace databáze (MSSQL, MySQL, PostgreSQL, . . .)
- Dva přístupy
  - `Code First` – Databáze se generuje z programu
  - `Database First` – Programový kód se generuje dle struktury DB

Zadání většího úkolu

Navrhněte a naprogramujte code-first ORM podporující základní operace.

Vlastnosti:

- Volba tříd, které bude spravovat a korektní vytvoření databázového schématu (2 body)
- Vložení instance dané třídy do DB (2 body)
- Výběr instance z DB a editace instance třídy v DB (2 body)
- Smazaní instance dané třídy z DB (2 body)
- Rozumná architektura, styl práce, celková funkčnost (2 body)
- Bonusové body: funkcionalita „navíc“ (WHERE, LIMIT, SKIP, DISTINCT, primární klíče, vazby, . . .)
- Prodloužený deadline – „dva týdny“ (půlnoc úterý 18. 11.)