Wybrane elementy praktyki projektowania oprogramowania
Wykład 13/15
Relacyjne bazy danych (2)

Wiktor Zychla 2023/2024

1 Spis treści

2 Zapytania

Przypomnijmy skrypt bazy którą tworzyliśmy na wykładzie. Skrypt przeznaczony jest dla SQL Server

```
CREATE TABLE [dbo].[Child](
 [ID] [int] IDENTITY(1,1) NOT NULL,
 [ChildName] [nvarchar](150) NOT NULL,
 [ID_PARENT] [int] NOT NULL,
 CONSTRAINT [PK_Child] PRIMARY KEY CLUSTERED
(
 [ID] ASC
)
) ON [PRIMARY]
GO


CREATE TABLE [dbo].[Parent](
 [ID] [int] IDENTITY(1,1) NOT NULL,
 [ParentName] [nvarchar](150) NOT NULL,
 CONSTRAINT [PK_Parent] PRIMARY KEY CLUSTERED
(
 [ID] ASC
)
) ON [PRIMARY]

ALTER TABLE [dbo].[Child]  WITH CHECK ADD  CONSTRAINT
[FK_Child_Parent] FOREIGN KEY([ID_PARENT])
REFERENCES [dbo].[Parent] ([ID])
```

W przypadku PostgreSQL należy zmienić definicje na

```
CREATE TABLE parent (
 id SERIAL PRIMARY KEY,
 parentName varchar(150) NOT NULL
) ;


CREATE TABLE child (
 id SERIAL PRIMARY KEY,
 childName varchar(150) NOT NULL,
 id_parent int NOT NULL
);

ALTER TABLE child
ADD  CONSTRAINT fk_Child_Parent
FOREIGN KEY(id_parent)
REFERENCES parent (id)
```

Różnice w składni są kosmetyczne i największa sprowadza się do tego że domyślnie w PostgreSQL nazwy obiektów (tabel, kolumn) są sprowazdzane do **małych liter** chyba że są zamykane w cudzysłowach (co może prowadzić do nieporozumień).

W przypadku PostgreSQL, można skorzystać z onlineowego poligonu.

2.1 SELECT INSERT, UPDATE, DELETE

2.1.1 SELECT

Do pobrania danych z tabeli służy zapytanie typu SELECT

```
SELECT * from Parent
```

Zbiór wynikowy może być dodatkowo ograniczony klauzulą WHERE

```
SELECT * from Parent WHERE ParentName='parent1'
```

Klauzul może być wiele i mogą tworzyć całe wyrażenie logiczne

```
SELECT * from Parent WHERE 'parent'<ParentName AND ParentName<'parent1'
```

Zbiór może być dodatkowo porządkowany za pomocą ORDER BY

```
SELECT * from Parent WHERE 'parent'<ParentName AND ParentName<'parent1'
ORDER BY ParentName
```

2.2 INSERT

Dodawanie danych do tabeli możliwe jest za pomocą zapytania typu INSERT

```
INSERT INTO Parent (ParentName) VALUES ('Parent1')
```

Należy zwrócić uwagę na to że w przypadku tabeli o kluczu głównym typu IDENTITY (automatycznie nadawana unikalna wartość) w zapytaniu nie podaje się jawnie wartości klucza głównego (tu: ID).

Z kolei w sytuacji gdy jako generator wartości klucza głównego służy sekwencja:

```
CREATE SEQUENCE [dbo].[Sequence1]
 AS [int]
 START WITH 1
 INCREMENT BY 1
 MINVALUE -2147483648
 MAXVALUE 2147483647
 CACHE
```

a klucz główny nie ma anotacji IDENTITY, dodawanie rekordów wymagałoby pobierania kolejnych wartości sekwencji

```
SELECT NEXT VALUE FOR dbo.Sequence1
```

i używania ich jawnie w klauzuli INSERT

```
DECLARE @id int = NEXT VALUE FOR dbo.Sequence1

SET IDENTITY_INSERT Parent ON
INSERT INTO Parent (ID, ParentName) VALUES (@id, 'Parent2')
SET IDENTITY_INSERT Parent OFF
```

(tu dodatkowo używamy polecenia SET IDENTITY_INSERT wymuszającego wymaganie wartości identyczności dla nowo wstawianego rekordu)

Jeżeli w wyniku zadanego zapytania do klienta ma zwrotnie wrócić wartość identyfikatora automatycznie nadana przez bazę danych to należy użyć składni:

- Dla SQL Server

```
INSERT INTO Parent (ParentName) VALUES ('Parent1') SELECT SCOPE_IDENTITY()
```

- Dla PostgreSQL

```
INSERT INTO Parent (ParentName) VALUES ('Parent1') RETURNING ID
```

2.2.1 UPDATE, DELETE

Do modyfikacji rekordów służy zapytanie typu UPDATE np.

```
UPDATE Parent SET ParentName='new name' WHERE ID=1
```

Do usuwania rekordów – DELETE

```
DELETE FROM Parent WHERE ID=1
```

2.3 GROUP/HAVING, COUNT/AVG, MIN, MAX

Gdyby tabela zawierała, oprócz opisu rekordu (ParentName) również wartości typów liczbowych, umożliwiających agregację, możliwe byłoby korzystanie z funkcji agregujących lub grupowania.

W celu demonstracji

```
ALTER TABLE PARENT ADD Val INT NULL
```

Następnie można zmodyfikować wartości, np.:

```
UPDATE Parent set Val = LEN(ParentName)
```

2.3.1 Agregacje

Przykłady zapytań agregujących

```
SELECT COUNT(*) FROM Parent
```

zwraca liczbę rekordów

```
SELECT AVG(Val) FROM Parent
```

zwraca średnią wartość kolumny Val (podobnie zadziała MIN czy MAX)

2.3.2 Grupowanie

Grupowanie rekordów po wartości kolumny wymaga wybrania kolumny do grupowania i zastosowania jakiejś funkcji agregującej do grupy, np.:

```
SELECT ParentName, COUNT(ParentName) from Parent
GROUP BY ParentName
```

zwraca rekordy pogrupowane po identycznej wartości ParentName a dla każdej grupy – jej liczność.

Klazula HAVING pozwala na filtrowanie grup:

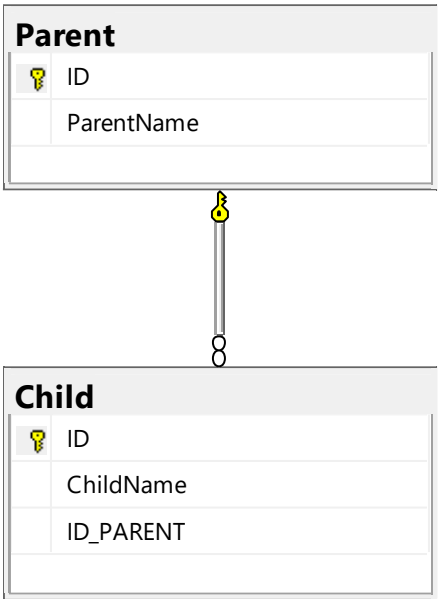
```
SELECT ParentName, COUNT(ParentName) from Parent
GROUP BY ParentName
HAVING COUNT(ParentName) >= 3
```

i jest czymś istotnie innym niż WHERE (które filtruje pojedyncze wiersze)

3 Relacje, JOIN

3.1 Relacje jeden-wiele

Utworzona tabela demonstruje najczęściej spotykany typ relacji – relację jeden-wiele:



Dzięki zdefiniowanemu kluczowi (klucz obcy: kolumna ID_PARENT z tabeli Child jest kluczem obcym do kolumny ID z tabeli Parent) baza danych dba o **spójność referencyjną** nie pozwalając na posługiwanie się wartościami spoza zbioru dostępnych identyfikatorów.

Pobranie rekordów z tabel połączonych relacjami możliwe jest dzięki klauzuli złączenia kartezjańskiego JOIN w zapytaniu:

```
SELECT Child.ChildName, Parent.ParentName
FROM Child
JOIN Parent on Child.ID_PARENT = Parent.ID
```

Tę samą relację można przeglądać od strony tabeli-rodzica

```
SELECT Parent.ParentName, Child.ChildName
FROM Parent
LEFT JOIN Child on Parent.ID = Child.ID_PARENT
```

otrzymując w wyniku również te rekordy-rodziców dla których nie istnieją wskazujące na nie rekordy-dzieci.

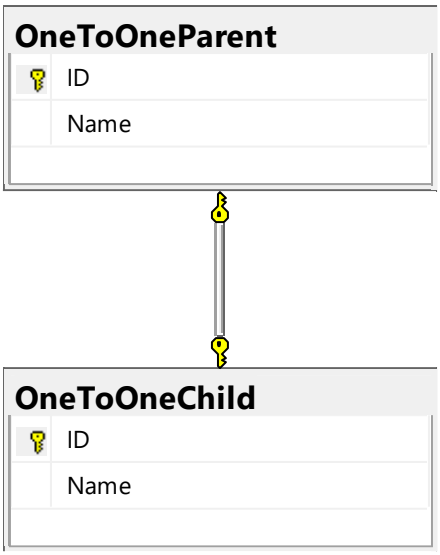
Warto zauważyć że wybieranie połączonych danych może być ograniczone inaczej niż przez złączenie kartezjańskie, na przykład

```
SELECT
   Parent.ParentName,
   (SELECT COUNT(*) FROM Child WHERE Child.ID_PARENT = Parent.ID)
FROM Parent
```

wybiera listę rekordów-rodziców oraz odpowiadającą każdemu liczbę rekordów-dzieci.

3.2 Relacja jeden-jeden

Relacja jeden-jeden obsługiwana jest przez relację klucza obcego z kolumny klucza głównego:



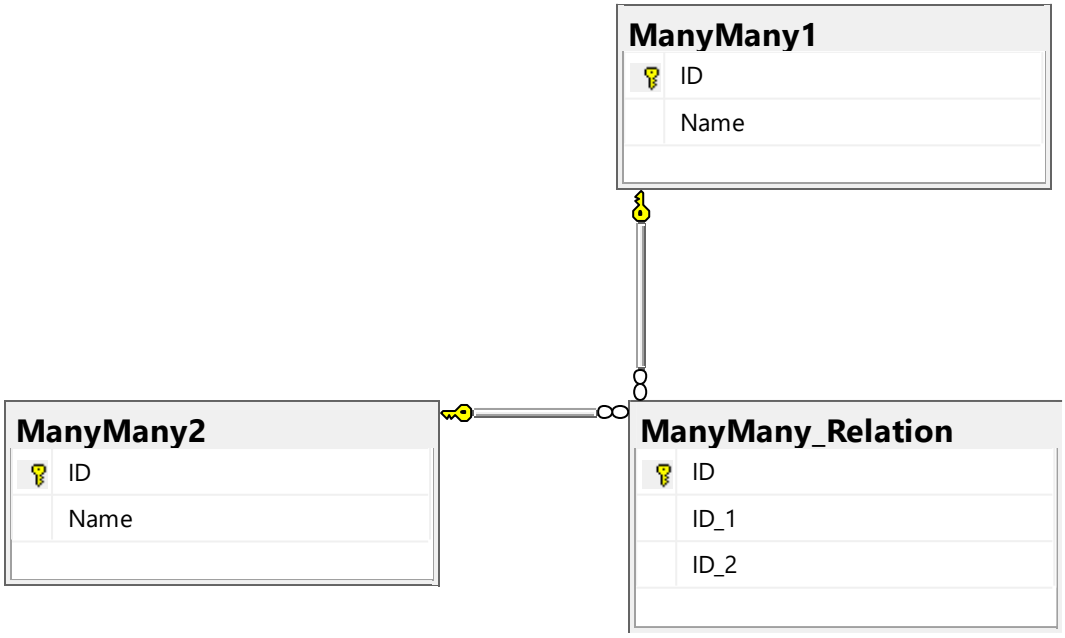

```
CREATE TABLE [dbo].[OneToOneParent](
	[ID] [int] IDENTITY(1,1) NOT NULL,
	[Name] [nchar](10) NOT NULL,
 CONSTRAINT [PK_OneToOneParent] PRIMARY KEY CLUSTERED
(
	[ID] ASC
))

CREATE TABLE [dbo].[OneToOneChild](
	[ID] [int] NOT NULL,
	[Name] [nchar](10) NULL,
 CONSTRAINT [PK_OneToOneChild] PRIMARY KEY CLUSTERED
(
	[ID] ASC
))

ALTER TABLE [dbo].[OneToOneChild]
  ADD CONSTRAINT [FK_OneToOneChild_OneToOneParent] FOREIGN KEY([ID])
REFERENCES [dbo].[OneToOneParent] ([ID])
```

3.3 Relacja wiele-wiele

Relacja wiele-wiele wymaga pomocniczej tabeli:

```
CREATE TABLE [dbo].[ManyMany_Relation](
	[ID] [int] IDENTITY(1,1) NOT NULL,
	[ID_1] [int] NOT NULL,
	[ID_2] [int] NOT NULL,
 CONSTRAINT [PK_ManyMany_Relation] PRIMARY KEY CLUSTERED
(
	[ID] ASC
))

CREATE TABLE [dbo].[ManyMany1](
	[ID] [int] IDENTITY(1,1) NOT NULL,
	[Name] [nchar](10) NOT NULL,
 CONSTRAINT [PK_ManyMany1] PRIMARY KEY CLUSTERED
(
	[ID] ASC
))
```

```
CREATE TABLE [dbo].[ManyMany2](
	[ID] [int] IDENTITY(1,1) NOT NULL,
	[Name] [nchar](10) NOT NULL,
 CONSTRAINT [PK_ManyMany2] PRIMARY KEY CLUSTERED
(
	[ID] ASC
))

ALTER TABLE [dbo].[ManyMany_Relation]  ADD  CONSTRAINT
[FK_ManyMany_Relation_ManyMany1] FOREIGN KEY([ID_1])
REFERENCES [dbo].[ManyMany1] ([ID])

ALTER TABLE [dbo].[ManyMany_Relation] CHECK CONSTRAINT
[FK_ManyMany_Relation_ManyMany1]

ALTER TABLE [dbo].[ManyMany_Relation]  ADD  CONSTRAINT
[FK_ManyMany_Relation_ManyMany2] FOREIGN KEY([ID_2])
REFERENCES [dbo].[ManyMany2] ([ID])
```



4 Programowanie baz danych w node.js

Współczesne pakiety do komunikacji z bazami danych wykorzystują wzorzec **async-await**. Jeśli tylko jest taka możliwość, warto z niej korzystać.

4.1 Pakiet mssql

W przypadku Sql Server, podstawowy pakiet to **mssql**.

Do nawiązania połączenia wymagane jest podanie parametrów połączenia, a następnie można wykonać zapytanie:

```
var mssql = require('mssql');

async function main() {

    var conn = new mssql.ConnectionPool(
      'server=localhost,1433;database=foo;user id=foo;password=foo');
    try {

        await conn.connect();

        var request = new mssql.Request(conn);
        var result  = await request.query('select * from Parent');

        result.recordset.forEach( r => {
            console.log( `${r.ID} ${r.ParentName}`);
        })

        await conn.close();

    }
    catch ( err ) {
        if ( conn.connected )
            conn.close();
        console.log( err );
    }
}

main();
```

Uwaga! W przypadku instalacji lokalnej serwera, może być konieczne dopisanie do ciągu połączenia parametru

```
TrustServerCertificate=true
```

Ten prosty przykład ilustruje zasadę zgodnie z którą rekordy zwracane przez podsystem komunikacji z bazami danych są materializowane do obiektów o polach odpowiadających kolumnom tabeli.

Zasada jest następująca – obiekt typu **ConnectionPool** należy skonstruować raz i wykorzystywać go przez cały czas życia aplikacji, nawet w aplikacji obsługującej równolegle wiele połączeń (= w aplikacji internetowej). Obiekt ten odpowiada za utrzymywanie tak zwanej *puli połączeń* dzięki czemu kolejne wykonywane z poziomu kodu zapytania nie muszą zwykle wyzwalać kolejnych negocjacji parametrów połączenia. Wzorzec puli połączeń jest niezwykle istotny przy programowaniu aplikacji bazodanowych o dużym obciążeniu.

Z kolei obiekt typu **Request** należy konstruować zawsze, tam gdzie jest potrzebny.

4.2 Pakiet pg

W przypadku PostgreSQL korzysta się z pakietu **pg**, w tym przypadku zasada jest podobna – obiekt **Pool** służy do konstrukcji parametrów połączenia i albo użyje się metody **connect** do uzyskania osobnego obiektu typu **Client** (który po wykonaniu zapytania należy zwrócić do puli metodą **release**) albo użyje się metody **query** wprost na obiekcie **Pool**.

```
var pg = require('pg');

(async function main() {

    var pool = new pg.Pool({
        host: 'localhost',
        database: 'PersonCatalog',
        user: 'pg',
        password: 'pg'
    });

    try {
        var result = await pool.query('select * from Parent');

        result.rows.forEach( r => {
            console.log( `${r.ID} ${r.ParentName}` );
        });
    }
    catch ( err ) {
        console.log( err );
    }
})();
```

4.3 Zagrożenie SQL Injection

Jednym z typowych błędów jakie można popełnić przy projektowaniu aplikacji korzystającej z bazy danych jest otwarcie podatności SQL Injection. Podatność ta pojawia się wtedy, kiedy programista chce zbudować kwerendę do której parametry podaje użytkownik.

Typowy przykład do kwerenda dodająca dane do bazy danych – użytkownik na formularzu wprowadza swoje imię i nazwisko, a aplikacja buduje zapytanie typu INSERT które dodaje rekord do bazy.

```
var name = 'Jan1 ' + new Date().toString();

var request = new mssql.Request(conn);
await request.query( `insert into Parent (ParentName) values ('${name}')`);
```

Od strony technicznej tak zbudowany kod działa – wartość wprowadzona przez użytkownika (tu: w zmiennej **name**) trafia do bazy. Użytkownik może jednak nadużyć danej mu możliwości i przedstawić się na formularzu jako

```
var name = "'); delete from Parent; --";
```

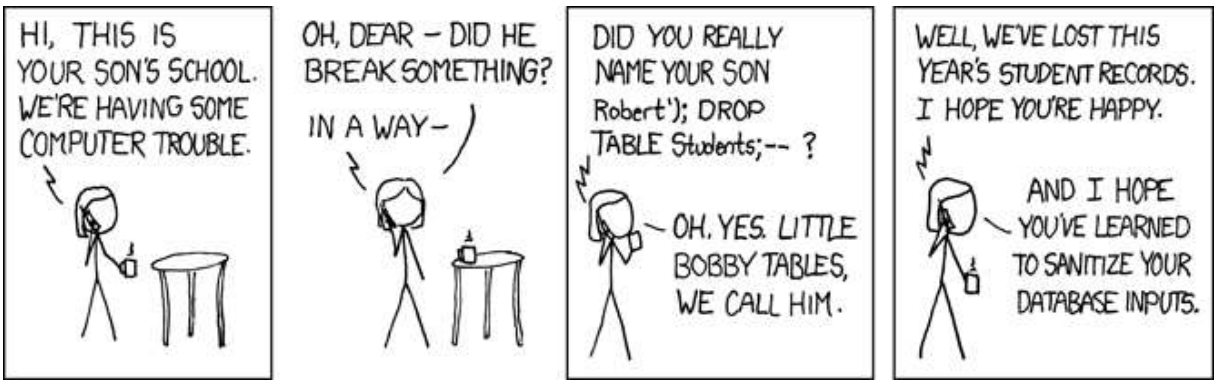
Po uzupełnieniu szablonu zapytania przez parametr użytkownika, do bazy trafia zapytanie

```
insert into Parent (ParentName) values (''); delete from Parent; --')
```

Celem użytkownika było takie uzupełnienie kwerendy w którym

- Część kwerendy napisana przez programistę jest poprawnie uzupełniona tak żeby tworzyć poprawny początek zapytania
- Za częścią napisaną przez programistę pojawia się separator ; i część kwerendy napisana przez użytkownika
- Aby koniec zapytania napisanego przez programistę nie stanowił problemu dla parsera SQL, użytkownik na końcu parametru dodaje – – czyli symbol komentarza, powodujący że ostatnia część zapytania jest ignorowana

Jak widać, użytkownik mógł w ten sposób wprowadzić do zapytania dowolną kwerendę.



Rysunek 1 https://xkcd.com/327/

W celu uniemożliwienia nadużycia parsera SQL, programista powinien pamiętać o podstawowej zasadzie

Tam gdzie część zapytania SQL pochodzi od użytkownika, należy używać **kwerendy parametryzowanej!**

Kwerenda parametryzowana oddziela część zapytania podlegającą parsowaniu od części która jest traktowana jako nieparsowana i jest zastępowana wartością już po sparsowaniu całej kwerendy. Nawet jeśli pararametr otrzymuje wartość, która stanowi poprawny SQL lub fragment SQL, wartość nigdy nie podlega parsowaniu.

```
var name = "'); delete from Parent; --";

var request = new mssql.Request(conn);
request.input("ParentName", name);
await request.query( `insert into Parent (ParentName) values (@ParentName)`);
```

4.4 Wzorzec Repository

Tę obserwację o odpowiedniości rekordów z tabeli i instancji obiektów można uogólnić i przygotować warstwę pośredniczącą między aplikacją a bazą danych, dla której językiem komunikacji staną się obiekty i operacje na nich wykonywane – w prostym przypadku może być to wręcz interfejs typu CRUD (create, retrieve, update, delete).

Oddzielenie logiki aplikacji od szczegółów komunikacji z bazą danych spłaca się wtedy, kiedy okazuje się, że można wymienić implementację repozytorium bez potrzeby modyfikacji kodu aplikacji:

```
var mssql = require('mssql');

class ParentRepository {

    constructor( conn ) {
        this.conn = conn;
    }

    async retrieve(name = null) {
        try {
            var req = new mssql.Request( this.conn );
            if ( name ) req.input('name', name);

            var res = await req.query( 'select * from Parent' + ( name ? '
where ParentName=@name' : '') );

            return name ? res.recordset[0] : res.recordset;
        }
        catch ( err ) {
            console.log( err );
```

```
            return [];
        }
    }

    async insert(simpleParent) {

        if ( !simpleParent) return;
        try {
            var req = new mssql.Request( this.conn );
            req.input("Name", simpleParent.Name);

            var res = await req.query( 'insert into Parent (ParentName)
values (@Name) select scope_identity() as id');
            return res.recordset[0].id;
        }
        catch ( err ) {
            console.log( err );
            throw err;
        }
    }

    async update(simpleParent) {

        if ( !simpleParent || !simpleParent.ID ) return;

        try {
            var req = new mssql.Request(this.conn);
            req.input("id", simpleParent.ID);
            req.input("Name", simpleParent.Name);

            var ret = await req.query('update Parent set ParentName=@Name
where id=@id');

            return ret.rowsAffected[0];
        }
        catch ( err ) {
            console.log( err );
            throw err;
        }
    }
}

async function main() {

    var conn = new
mssql.ConnectionPool('server=localhost,1433;database=foo;user
id=foo;password=foo');
    try {
```

```
        await conn.connect();

        var repo = new ParentRepository(conn);

        // pobierz wszystkie rekordy
        var items = await repo.retrieve();

        items.forEach( e => {
            console.log( JSON.stringify(e));
        });

        // pobierz rekord spełniający warunki
        var item = await repo.retrieve('Parent55');
        if ( item ) {
            item.Name = 'new name';
            // jeśli jest to go popraw
            var rowsAffected = await repo.update( item );

            console.log( `zmodyfikowano ${rowsAffected} rekordów` );
        } else {
            item = {
                Name : 'Parent55'
            };
            // jeśli nie ma to go utwórz i zwróć identyfikator nowo
wstawionego
            var id = await repo.insert( item );
            console.log( `identyfikator nowo wstawionego rekordu ${id}` );
        }

    }
    catch ( err ) {
        if ( conn.connected )
            conn.close();
        console.log( err );
    }
}

main();
```