1. Praca z bazami danych w .NET: SqlConnection i SqlCommand

1. Wprowadzenie do SqlConnection i SqlCommand

Czym są SqlConnection i SqlCommand?

SqlConnection i SqlCommand to klasy w frameworku .NET, które służą jako sterowniki baz danych zaprojektowane specjalnie dla Microsoft SQL Server. Są częścią przestrzeni nazw System.Data.SqlClient (w .NET Framework) lub Microsoft.Data.SqlClient (w nowoczesnym .NET).

Każdy system zarządzania bazą danych zazwyczaj ma własny zestaw sterowników, które ułatwiają komunikację między aplikacjami a bazą danych. Dla SQL Server, SqlConnection i SqlCommand są natywnymi dostawcami ADO.NET, którzy oferują zoptymalizowaną wydajność i dostęp do specyficznych funkcji SQL Server.

- **SqlConnection**: Reprezentuje połączenie z bazą danych SQL Server
- **SqlCommand**: Reprezentuje instrukcję SQL lub procedurę składowaną do wykonania na bazie danych SQL Server

Zalety SqlConnection i SqlCommand

1. **Natywna integracja**: Zoptymalizowane specjalnie dla SQL Server, zapewniające lepszą wydajność niż ogólne dostawcy baz danych
2. **Bogate w funkcje**: Bezpośredni dostęp do specyficznych funkcji i typów danych SQL Server
3. **Silne typowanie**: Bezpieczny dostęp do danych z typami specyficznymi dla SQL Server
4. **Funkcje bezpieczeństwa**: Wbudowana obsługa bezpiecznych ciągów połączeń i zapytań parametryzowanych
5. **Wydajność**: Wbudowana i zoptymalizowana dla SQL Server pula połączeń

Wady SqlConnection i SqlCommand

1. **Uzależnienie od dostawcy**: Kod jest ściśle powiązany z SQL Server
2. **Ręczne zarządzanie zasobami**: Wymaga jawnej obsługi połączeń i ich usuwania
3. **Niższy poziom abstrakcji**: Więcej kodu szablonowego w porównaniu z ORM-ami jak Entity Framework
4. **Ograniczone wsparcie wieloplatformowe**: Chociaż poprawione w .NET Core, historycznie miało ograniczenia
5. **Ręczne SQL**: Wymaga bezpośredniego pisania i utrzymywania zapytań SQL

Pula połączeń i zarządzanie zasobami

Jednym z najważniejszych aspektów pracy z SqlConnection jest zrozumienie puli połączeń. Otwieranie połączenia z bazą danych jest kosztowną operacją, obejmującą:

1. Obsługę protokołu sieciowego
2. Uwierzytelnianie
3. Alokację zasobów serwera

4. Kontrole bezpieczeństwa

Pula połączeń SQL Server działa poprzez utrzymywanie zestawu dostępnych połączeń, które mogą być ponownie wykorzystane, zamiast ustanawiania nowych połączeń dla każdej operacji bazodanowej.

Dlaczego szybkie zamykanie połączeń jest ważne:

1. **Efektywność zasobów**: Połączenia bazodanowe zużywają zasoby serwera (pamięć, wątki robocze)
2. **Skalowalność**: Dostępna jest ograniczona liczba jednoczesnych połączeń
3. **Wydajność**: Zbyt wiele otwartych połączeń może obniżyć wydajność bazy danych
4. **Wyczerpanie połączeń**: Inne części aplikacji mogą nie być w stanie się połączyć

Używanie instrukcji `using`

Instrukcja `using` w C# zapewnia czysty sposób zapewnienia prawidłowego usunięcia zasobów, w tym połączeń z bazą danych:

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    // Wykonaj operacje bazodanowe
    // Nie ma potrzeby jawnego wywoływania connection.Close()
} // Połączenie jest automatycznie zamykane i usuwane w tym miejscu
```

Instrukcja `using` zapewnia wywołanie `Dispose()` na obiekcie połączenia, co zwraca je do puli połączeń, nawet jeśli wystąpi wyjątek.

2. SQL Injection i bezpieczeństwo parametrów

Zrozumienie SQL Injection

SQL Injection jest jedną z najczęstszych i najniebezpieczniejszych luk bezpieczeństwa w aplikacjach bazodanowych. Występuje, gdy niezaufane dane wprowadzone przez użytkownika są bezpośrednio łączone z zapytaniami SQL.

Rozważ ten podatny kod:

```
string username = userInput; // Z danych wprowadzonych przez użytkownika
string query = "SELECT * FROM Users WHERE Username = '" + username + "'";

using (SqlConnection connection = new SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(query, connection))
{
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    // Przetwarzanie wyników
}
```

Jeśli złośliwy użytkownik wprowadzi: `' OR 1=1 --`, wynikowe zapytanie staje się:

sql

```
SELECT * FROM Users WHERE Username = '' OR 1=1 --'
```

`--` komentuje resztę zapytania, a `1=1` jest zawsze prawdziwe, potencjalnie zwracając wszystkich użytkowników w bazie danych.

Bardziej niebezpieczne iniekcje mogą:

- Usunąć dane: `'; DROP TABLE Users; --`
- Modyfikować dane: `'; UPDATE Users SET IsAdmin = 1 WHERE Username = 'attacker'; --`
- Wydobyć wrażliwe informacje: `'; SELECT TOP 1 Password FROM Users; --`

Używanie zapytań parametryzowanych

Zapytania parametryzowane oddzielają kod SQL od danych, zapobiegając SQL Injection:

```
string username = userInput; // Z danych wprowadzonych przez użytkownika
string query = "SELECT * FROM Users WHERE Username = @Username";

using (SqlConnection connection = new SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(query, connection))
{
    command.Parameters.AddWithValue("@Username", username);

    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    // Przetwarzanie wyników
}
```

Dlaczego parametry są ważne:

1. Bezpieczeństwo: Parametry SQL zapewniają, że dane użytkownika są traktowane jako dane, a nie wykonywalny kod
2. Wydajność: Zapytania parametryzowane umożliwiają buforowanie i ponowne wykorzystanie planów zapytań
3. Bezpieczeństwo typów danych: Parametry obsługują konwersję typów i zapobiegają problemom z formatowaniem
4. Czytelność: Kod jest czystszy, a intencja SQL jest jaśniejsza

3. Pobieranie danych za pomocą metod asynchronicznych

Nowoczesne aplikacje .NET powinny wykorzystywać programowanie asynchroniczne, aby poprawić skalowalność. Oto jak pobierać dane za pomocą async/await:

```
public async Task<List<Customer>> GetCustomersAsync()
{
    List<Customer> customers = new List<Customer>();
    string query = "SELECT CustomerId, FirstName, LastName, Email FROM Customers";

    using (SqlConnection connection = new SqlConnection(_connectionString))
    using (SqlCommand command = new SqlCommand(query, connection))
    {
        // Otwórz połączenie asynchronicznie
        await connection.OpenAsync();
```

```
        // Wykonaj zapytanie asynchronicznie
        using (SqlDataReader reader = await command.ExecuteReaderAsync())
        {
            // Odczytaj wyniki asynchronicznie
            while (await reader.ReadAsync())
            {
                customers.Add(new Customer
                {
                    CustomerId = reader.GetInt32(0),
                    FirstName = reader.GetString(1),
                    LastName = reader.GetString(2),
                    Email = reader.GetString(3)
                });
            }
        }
    }

    return customers;
}
```

Kluczowe metody asynchroniczne w ADO.NET:

- `OpenAsync()`: Otwiera połączenie asynchronicznie
- `ExecuteReaderAsync()`: Wykonuje polecenie zwracające czytnik asynchronicznie
- `ExecuteNonQueryAsync()`: Wykonuje polecenie nie-zapytanie asynchronicznie
- `ExecuteScalarAsync()`: Wykonuje polecenie zwracające wartość skalarną asynchronicznie
- `ReadAsync()`: Przesuwa czytnik do następnego rekordu asynchronicznie

Korzyści asynchronicznych operacji bazodanowych:

1. Skalowalność: Wątki nie są blokowane podczas oczekiwania na operacje bazodanowe
2. Responsywność: Interfejs użytkownika pozostaje responsywny w aplikacjach klienckich
3. Przepustowość: Więcej jednoczesnych operacji przy mniejszej liczbie wątków
4. Efektywność zasobów: Lepsze wykorzystanie zasobów serwera

4. Modyfikowanie danych: Operacje Insert, Update, Delete

Wstawianie danych

```
public async Task<int> AddCustomerAsync(Customer customer)
{
    string query = @"
        INSERT INTO Customers (FirstName, LastName, Email)
        VALUES (@FirstName, @LastName, @Email);
        SELECT SCOPE_IDENTITY();";

    using (SqlConnection connection = new SqlConnection(_connectionString))
    using (SqlCommand command = new SqlCommand(query, connection))
    {
        command.Parameters.AddWithValue("@FirstName", customer.FirstName);
```

```
        command.Parameters.AddWithValue("@LastName", customer.LastName);
        command.Parameters.AddWithValue("@Email", customer.Email);

        await connection.OpenAsync();

        // ExecuteScalarAsync zwraca pierwszą kolumnę pierwszego wiersza
        var result = await command.ExecuteScalarAsync();
        return Convert.ToInt32(result);
    }
}
```

Aktualizowanie danych

```
public async Task<bool> UpdateCustomerAsync(Customer customer)
{
    string query = @"
        UPDATE Customers
        SET FirstName = @FirstName,
            LastName = @LastName,
            Email = @Email
        WHERE CustomerId = @CustomerId";

    using (SqlConnection connection = new SqlConnection(_connectionString))
    using (SqlCommand command = new SqlCommand(query, connection))
    {
        command.Parameters.AddWithValue("@CustomerId", customer.CustomerId);
        command.Parameters.AddWithValue("@FirstName", customer.FirstName);
        command.Parameters.AddWithValue("@LastName", customer.LastName);
        command.Parameters.AddWithValue("@Email", customer.Email);

        await connection.OpenAsync();

        int rowsAffected = await command.ExecuteNonQueryAsync();
        return rowsAffected > 0;
    }
}
```

Usuwanie danych

```
public async Task<bool> DeleteCustomerAsync(int customerId)
{
    string query = "DELETE FROM Customers WHERE CustomerId = @CustomerId";

    using (SqlConnection connection = new SqlConnection(_connectionString))
    using (SqlCommand command = new SqlCommand(query, connection))
    {
        command.Parameters.AddWithValue("@CustomerId", customerId);

        await connection.OpenAsync();

        int rowsAffected = await command.ExecuteNonQueryAsync();
        return rowsAffected > 0;
    }
}
```

Kluczowe punkty dotyczące modyfikacji danych:

1. **Typ polecenia**: Dla instrukcji SQL używaj `CommandType.Text` (domyślnie)
2. **ExecuteNonQuery**: Dla operacji INSERT, UPDATE lub DELETE, które nie zwracają wyników
3. **ExecuteScalar**: Gdy potrzebujesz pojedynczej wartości (np. identyfikatora)
4. **Parametry**: Zawsze używaj parametrów dla danych dostarczonych przez użytkownika lub zmiennych
5. **Wartości zwracane**: Rozważ znaczące wartości zwracane, aby potwierdzić sukces lub porażkę

5. Wykonywanie procedur składowanych

Procedury składowane to prekompilowane instrukcje SQL przechowywane na serwerze bazy danych, oferujące lepszą wydajność i lepsze oddzielenie zagadnień.

Podstawowe wykonanie procedury składowanej

```
public async Task<List<Customer>> GetCustomersByCountryAsync(string country)
{
    List<Customer> customers = new List<Customer>();

    using (SqlConnection connection = new SqlConnection(_connectionString))
    using (SqlCommand command = new SqlCommand("GetCustomersByCountry", connection))
    {
        // Określ, że wykonujemy procedurę składowaną
        command.CommandType = CommandType.StoredProcedure;

        // Dodaj parametry
        command.Parameters.AddWithValue("@Country", country);

        await connection.OpenAsync();

        using (SqlDataReader reader = await command.ExecuteReaderAsync())
        {
            while (await reader.ReadAsync())
            {
                customers.Add(new Customer
                {
                    CustomerId = reader.GetInt32(reader.GetOrdinal("CustomerId")),
                    FirstName = reader.GetString(reader.GetOrdinal("FirstName")),
                    LastName = reader.GetString(reader.GetOrdinal("LastName")),
                    Email = reader.GetString(reader.GetOrdinal("Email"))
                });
            }
        }
    }

    return customers;
}
```

Obsługa parametrów wyjściowych

```
public async Task<(bool Success, string Message)> ProcessOrderAsync(int orderId)
{
    using (SqlConnection connection = new SqlConnection(_connectionString))
    using (SqlCommand command = new SqlCommand("ProcessOrder", connection))
    {
        command.CommandType = CommandType.StoredProcedure;

        // Parametr wejściowy
        command.Parameters.AddWithValue("@OrderId", orderId);

        // Parametry wyjściowe
        SqlParameter successParam = new SqlParameter
```

```
        {
            ParameterName = "@Success",
            SqlDbType = SqlDbType.Bit,
            Direction = ParameterDirection.Output
        };

        SqlParameter messageParam = new SqlParameter
        {
            ParameterName = "@Message",
            SqlDbType = SqlDbType.NVarChar,
            Size = 255,
            Direction = ParameterDirection.Output
        };

        command.Parameters.Add(successParam);
        command.Parameters.Add(messageParam);

        await connection.OpenAsync();
        await command.ExecuteNonQueryAsync();

        bool success = (bool)successParam.Value;
        string message = messageParam.Value?.ToString() ?? string.Empty;

        return (success, message);
    }
}
```

Korzyści z procedur składowanych:

1. Wydajność: Prekompilowane plany wykonania
2. Bezpieczeństwo: Szczegółowe uprawnienia mogą być stosowane do procedur zamiast tabel
3. Efektywność sieci: Zmniejszony ruch sieciowy (wysyłanie nazwy procedury zamiast pełnego zapytania)
4. Enkapsulacja: Logika biznesowa może być enkapsulowana w bazie danych
5. Utrzymanie: SQL może być modyfikowany bez zmiany kodu aplikacji

6. Transakcje: Zapewnienie integralności procesów biznesowych

Zrozumienie transakcji

Transakcja to sekwencja operacji wykonywanych jako pojedyncza logiczna jednostka pracy. Transakcja ma cztery kluczowe właściwości, często określane jako ACID:

1. **Atomowość**: Wszystkie operacje w transakcji kończą się sukcesem lub wszystkie zawodzą
2. **Spójność**: Baza danych pozostaje w spójnym stanie przed i po transakcji
3. **Izolacja**: Transakcje działają niezależnie bez zakłóceń
4. **Trwałość**: Po zatwierdzeniu zmiany przetrwają awarie systemu

Implementacja transakcji z SqlConnection

Rozważmy scenariusz bankowy, w którym musimy przelewać pieniądze między kontami:

```
public async Task<bool> TransferFundsAsync(int fromAccountId, int toAccountId, decimal
amount)
{
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();

        // Rozpocznij transakcję SQL
        SqlTransaction transaction = connection.BeginTransaction();

        try
        {
            // 1. Odejmij z konta źródłowego
            using (SqlCommand withdrawCommand = new SqlCommand(
                "UPDATE Accounts SET Balance = Balance - @Amount WHERE AccountId =
@AccountId",
                connection, transaction))
            {
                withdrawCommand.Parameters.AddWithValue("@Amount", amount);
                withdrawCommand.Parameters.AddWithValue("@AccountId", fromAccountId);

                int rowsAffected = await withdrawCommand.ExecuteNonQueryAsync();
                if (rowsAffected == 0)
                {
                    // Konto nie znalezione lub inny problem
                    transaction.Rollback();
                    return false;
                }
            }

            // 2. Dodaj do konta docelowego
            using (SqlCommand depositCommand = new SqlCommand(
                "UPDATE Accounts SET Balance = Balance + @Amount WHERE AccountId =
@AccountId",
                connection, transaction))
            {
```

```
                depositCommand.Parameters.AddWithValue("@Amount", amount);
                depositCommand.Parameters.AddWithValue("@AccountId", toAccountId);

                int rowsAffected = await depositCommand.ExecuteNonQueryAsync();
                if (rowsAffected == 0)
                {
                    // Konto nie znalezione lub inny problem
                    transaction.Rollback();
                    return false;
                }
            }

            // 3. Zapisz transakcję w historii transakcji
            using (SqlCommand logCommand = new SqlCommand(
                "INSERT INTO TransactionHistory (FromAccountId, ToAccountId, Amount,
TransactionDate) " +
                "VALUES (@FromAccountId, @ToAccountId, @Amount, @TransactionDate)",
                connection, transaction))
            {
                logCommand.Parameters.AddWithValue("@FromAccountId", fromAccountId);
                logCommand.Parameters.AddWithValue("@ToAccountId", toAccountId);
                logCommand.Parameters.AddWithValue("@Amount", amount);
                logCommand.Parameters.AddWithValue("@TransactionDate", DateTime.Now);

                await logCommand.ExecuteNonQueryAsync();
            }

            // Jeśli dotarliśmy tutaj, wszystko powiodło się, więc zatwierdź transakcję
            transaction.Commit();
            return true;
        }
        catch (Exception)
        {
            // Jeśli cokolwiek pójdzie nie tak, wycofaj całą transakcję
            transaction.Rollback();
            throw;
        }
    }
}
```

Zalety transakcji:

1. **Integralność danych**: Zapewnia, że powiązane zmiany są wykonywane razem
2. **Spójny stan**: Baza danych pozostaje prawidłowa nawet po awariach
3. **Odzyskiwanie po błędach**: Prosty mechanizm wycofywania dla złożonych operacji
4. **Izolacja**: Chroni przed zakłóceniami z innych operacji

Wady transakcji:

1. **Narzut wydajnościowy**: Rejestrowanie i zarządzanie transakcjami dodaje narzut
2. **Blokowanie zasobów**: Może prowadzić do blokowania i problemów z współbieżnością
3. **Złożoność**: Wymagany dodatkowy kod i obsługa błędów
4. **Możliwe zakleszczenia**: Nieprawidłowo zaprojektowane transakcje mogą powodować zakleszczenia

7. Współbieżność: Podejścia pesymistyczne vs. optymistyczne

Współbieżność w bazach danych odnosi się do sposobu, w jaki system obsługuje wielu użytkowników próbujących jednocześnie uzyskać dostęp do tych samych danych lub je modyfikować.

Podejścia do współbieżności w bazach danych można podzielić na dwie główne kategorie: pesymistyczne i optymistyczne. Każde z nich opiera się na innym założeniu dotyczącym prawdopodobieństwa konfliktu danych i oferuje różne kompromisy w zakresie wydajności, spójności i izolacji.

W przypadku kontroli współbieżności pesymistycznej, system zakłada, że konflikty są prawdopodobne i aktywnie im zapobiega poprzez blokowanie zasobów. Gdy użytkownik rozpoczyna transakcję, która obejmuje dane, system blokuje te dane przed dostępem innych użytkowników do czasu zakończenia transakcji. Jest to jak rezerwowanie pokoju w hotelu - nikt inny nie może go zarezerwować, dopóki nie zwolnisz rezerwacji. Podejście to zapewnia spójność danych, ale może powodować opóźnienia i potencjalne zakleszczenia, gdy wiele transakcji oczekuje na siebie nawzajem.

Z kolei kontrola współbieżności optymistyczna zakłada, że konflikty są rzadkie, więc pozwala wielu użytkownikom na pracę z tymi samymi danymi jednocześnie. Zamiast blokować dane, system sprawdza, czy dane zostały zmodyfikowane przez kogoś innego przed zatwierdzeniem transakcji. Jeśli wykryty zostanie konflikt, transakcja jest wycofywana i musi zostać ponownie uruchomiona. To jak edytowanie dokumentu online - wszyscy mogą edytować, ale system musi rozwiązać konflikty, jeśli dwie osoby próbują zmienić to samo pole jednocześnie.

Wybór między tymi podejściami zależy od specyfiki aplikacji. Systemy o wysokim współczynniku odczytu do zapisu i niskim ryzyku konfliktu często korzystają na podejściu optymistycznym, podczas gdy aplikacje, w których spójność danych jest krytyczna i konflikty są częste, mogą lepiej działać z kontrolą pesymistyczną.

Współczesne systemy baz danych często oferują hybrydowe rozwiązania, łączące elementy obu podejść i umożliwiając programistom wybór odpowiedniego poziomu izolacji transakcji odpowiedniego dla konkretnego przypadku użycia, równoważąc w ten sposób wydajność z gwarancją spójności danych.

Pesymistyczna kontrola współbieżności

Pesymistyczna współbieżność zakłada, że konflikty wystąpią i zapobiega im, blokując zasoby przed ich modyfikacją.

Implementacja z SqlConnection:

```
public async Task<bool> UpdateProductStockPessimisticAsync(int productId, int
quantityChange)
{
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();

        using (SqlTransaction transaction = connection.BeginTransaction())
        {
            try
            {
                // 1. Wybierz z blokadą, aby zapobiec modyfikacji tego wiersza przez
inne transakcje
```

```
                string selectQuery = "SELECT StockQuantity FROM Products WITH (UPDLOCK)
WHERE ProductId = @ProductId";
                int currentStock;

                using (SqlCommand selectCommand = new SqlCommand(selectQuery,
connection, transaction))
                {
                    selectCommand.Parameters.AddWithValue("@ProductId", productId);

                    var result = await selectCommand.ExecuteScalarAsync();
                    if (result == null || result == DBNull.Value)
                    {
                        transaction.Rollback();
                        return false; // Produkt nie znaleziony
                    }

                    currentStock = Convert.ToInt32(result);
                }

                // 2. Reguła biznesowa: Nie pozwól na ujemny stan magazynowy
                int newStock = currentStock + quantityChange;
                if (newStock < 0)
                {
                    transaction.Rollback();
                    return false; // Niewystarczający stan magazynowy
                }

                // 3. Zaktualizuj stan magazynowy
                string updateQuery = "UPDATE Products SET StockQuantity = @NewStock
WHERE ProductId = @ProductId";
                using (SqlCommand updateCommand = new SqlCommand(updateQuery,
connection, transaction))
                {
                    updateCommand.Parameters.AddWithValue("@NewStock", newStock);
                    updateCommand.Parameters.AddWithValue("@ProductId", productId);

                    await updateCommand.ExecuteNonQueryAsync();
                }

                // Zatwierdź transakcję
                transaction.Commit();
                return true;
            }
            catch
            {
                transaction.Rollback();
                throw;
            }
        }
    }
}
```

Kluczowe wskazówki dotyczące blokowania w SQL Server obejmują:

- `UPDLOCK`: Umieszcza blokady aktualizacji na dostępnych wierszach
- `HOLDLOCK`: Utrzymuje blokady współdzielone do zakończenia transakcji

- `ROWLOCK`: Wymusza blokowanie na poziomie wiersza zamiast strony lub tabeli
- `XLOCK`: Umieszcza blokady wyłączne na zasobach

Optymistyczna kontrola współbieżności

Optymistyczna współbieżność pozwala wielu użytkownikom na próby modyfikacji bez blokowania, ale weryfikuje, czy dane nie zmieniły się przed zatwierdzeniem.

Implementacja z SqlConnection:

```
public async Task<bool> UpdateProductOptimisticAsync(Product product, int
originalVersion)
{
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();

        // Użyj RowVersion (timestamp) dla optymistycznej współbieżności
        string updateQuery = @"
            UPDATE Products
            SET Name = @Name,
                Price = @Price,
                StockQuantity = @StockQuantity
            WHERE ProductId = @ProductId
              AND RowVersion = @OriginalVersion;

            SELECT @@ROWCOUNT;";

        using (SqlCommand command = new SqlCommand(updateQuery, connection))
        {
            command.Parameters.AddWithValue("@ProductId", product.ProductId);
            command.Parameters.AddWithValue("@Name", product.Name);
            command.Parameters.AddWithValue("@Price", product.Price);
            command.Parameters.AddWithValue("@StockQuantity", product.StockQuantity);
            command.Parameters.AddWithValue("@OriginalVersion", originalVersion);

            int rowsAffected = Convert.ToInt32(await command.ExecuteScalarAsync());

            return rowsAffected > 0;
        }
    }
}
```

Jeśli zwracana wartość wynosi 0, oznacza to, że dane zostały zmodyfikowane przez innego użytkownika od czasu ich ostatniego odczytu.

Porównanie pesymistycznej i optymistycznej współbieżności:

Pesymistyczna współbieżność:

- Zalety: Zapobiega konfliktom zanim wystąpią, gwarantuje udaną transakcję
- Wady: Zmniejsza przepustowość systemu, może prowadzić do zakleszczeń, dłuższe czasy blokowania

Optymistyczna współbieżność:

- **Zalety**: Wyższa współbieżność, brak narzutu blokowania, lepsza skalowalność
- **Wady**: Wykrywanie konfliktów zamiast zapobiegania, wymaga logiki ponawiania, może prowadzić do frustracji użytkownika

8. Podsumowanie: Zalety i wady SqlConnection i SqlCommand

Zalety

1. **Wydajność**: Natywny sterownik SQL Server z zoptymalizowaną komunikacją
2. **Kontrola**: Precyzyjna kontrola nad operacjami bazodanowymi
3. **Elastyczność**: Obsługuje wszystkie funkcje i możliwości SQL Server
4. **Lekkość**: Minimalny narzut w porównaniu do rozwiązań ORM
5. **Pula połączeń**: Wbudowane efektywne zarządzanie połączeniami
6. **Kompatybilność**: Działa ze wszystkimi wersjami SQL Server
7. **Obsługa transakcji**: Solidne możliwości zarządzania transakcjami
8. **API asynchroniczne**: Pełne wsparcie async dla skalowalnych aplikacji

Wady

1. **Rozwlekłość**: Wymaga więcej kodu niż abstrakcje wyższego poziomu
2. **Ręczne SQL**: Programiści muszą pisać i utrzymywać SQL
3. **Ryzyko SQL Injection**: Wymaga starannej parametryzacji, aby uniknąć luk
4. **Zarządzanie zasobami**: Wymagana jawna obsługa połączeń i ich usuwanie
5. **Specyficzne dla SQL Server**: Nie jest bezpośrednio przenośne na inne systemy baz danych
6. **Brak mapowania obiektów**: Ręczna konwersja między bazą danych a modelami obiektowymi
7. **Ograniczona abstrakcja**: Mniejsze oddzielenie od fizycznego schematu bazy danych
8. **Krzywa uczenia**: Wymaga zrozumienia koncepcji ADO.NET

Kiedy używać SqlConnection/SqlCommand

Najlepsze przypadki użycia:

- Aplikacje krytyczne dla wydajności
- Aplikacje ze złożonymi wymaganiami SQL
- Gdy potrzebujesz bezpośredniej kontroli nad wykonaniem zapytania
- Mikrousługi, gdzie dostęp do bazy danych jest enkapsulowany
- Podczas pracy z systemami odziedziczonymi
- Dla specyficznych funkcji SQL Server niedostępnych w ORM-ach

Rozważ alternatywy, gdy:

- Budujesz aplikacje obsługujące różne bazy danych
- Szybki rozwój jest priorytetem
- Zespół woli pracować z obiektami niż z SQL
- Schemat bazy danych często się zmienia
- Testowanie jednostkowe jest głównym obszarem zainteresowania

W większości nowoczesnych aplikacji .NET często stosowane jest podejście hybrydowe - używanie Entity Framework do standardowych operacji CRUD, a jednocześnie korzystanie z SqlConnection/SqlCommand dla operacji krytycznych pod względem wydajności lub złożonych interakcji z bazą danych.

Najlepsze praktyki

1. Zawsze używaj zapytań parametryzowanych
2. Właściwie usuwaj połączenia i polecenia używając instrukcji `using`
3. Efektywnie wykorzystuj pulę połączeń
4. Korzystaj z metod asynchronicznych dla lepszej skalowalności
5. Wdrażaj odpowiednią obsługę błędów i logikę ponawiania
6. Rozważ transakcje dla operacji wieloetapowych
7. Wybierz odpowiedni mechanizm kontroli współbieżności

W podsumowaniu, SqlConnection i SqlCommand są potężnymi i wydajnymi narzędziami do pracy z bazami danych SQL Server w środowisku .NET. Choć wymagają więcej ręcznego kodu i uwagi niż rozwiązania ORM wyższego poziomu, oferują maksymalną kontrolę, wydajność i dostęp do pełnych możliwości SQL Server. Zrozumienie ich właściwego wykorzystania, zwłaszcza w zakresie zarządzania zasobami, bezpieczeństwa i współbieżności, jest kluczowe dla tworzenia wysokowydajnych i niezawodnych aplikacji bazodanowych.

Wybór między bezpośrednim dostępem do bazy danych przy użyciu SqlConnection/SqlCommand a rozwiązaniami ORM, takimi jak Entity Framework, zależy od konkretnych wymagań projektu, priorytetów wydajnościowych i umiejętności zespołu. W wielu przypadkach najlepszym rozwiązaniem może być podejście hybrydowe, które wykorzystuje mocne strony obu metodologii.