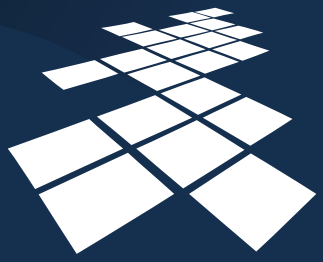
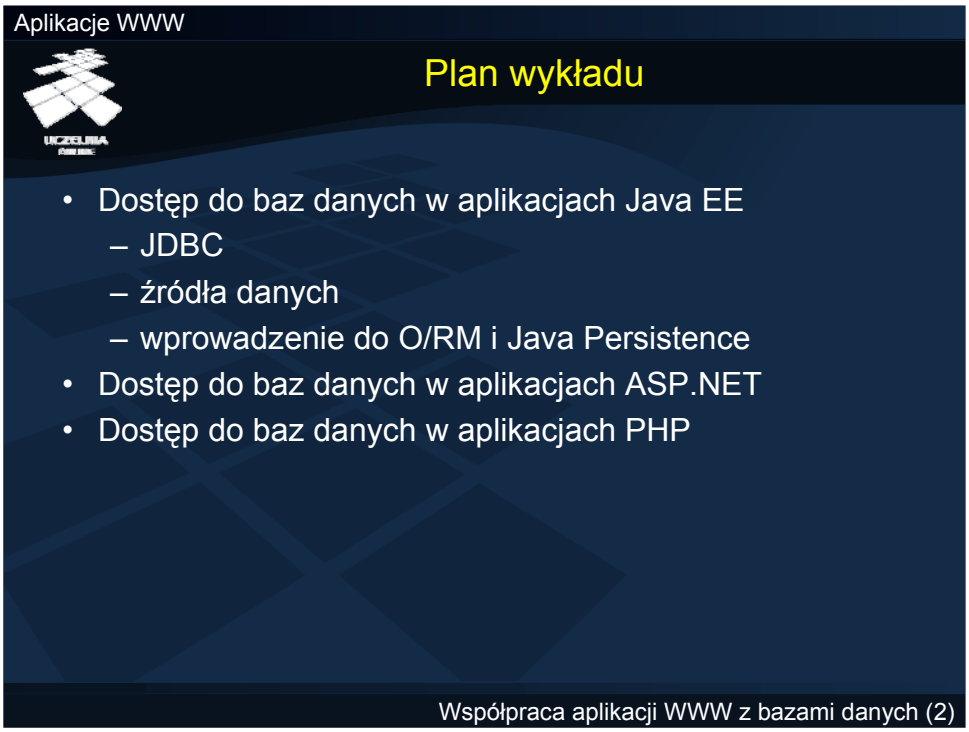
Współpraca aplikacji WWW z bazami danych

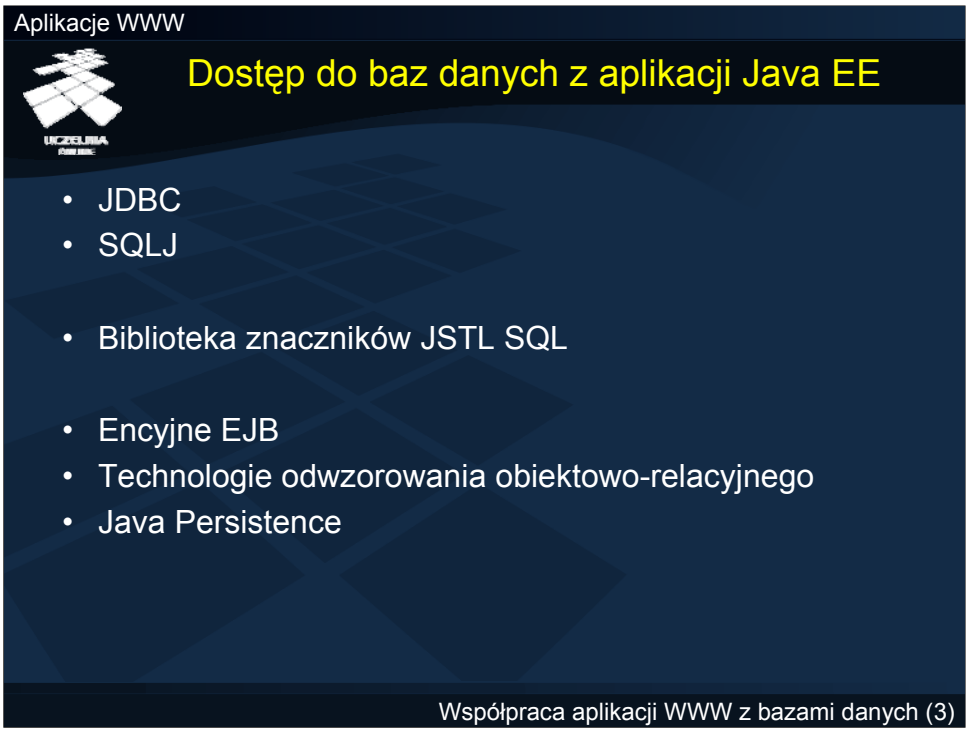
Wykład prowadzi:
Marek Wojciechowski



UCZELNIA
ONLINE



Celem wykładu jest przedstawienie mechanizmów dostępu do baz danych w najpopularniejszych technologiach do tworzenia aplikacji WWW: Java EE, ASP.NET i PHP. W kontekście technologii Java EE przedstawione będą podstawy JDBC (dla przypomnienia i kompletności wykładu), charakterystyczny dla aplikacji Java EE mechanizm uzyskiwania połączeń z bazą danych poprzez źródła danych oraz wprowadzenie do technologii odwzorowania obiektowo-relacyjnego (O/RM) i standardu Java Persistence. Technologie O/RM i Java Persistence oraz zaawansowane mechanizmy transakcyjne dla platformy Java EE będą szczegółowo omówione w ramach wykładów z przedmiotu „Zaawansowane aplikacje internetowe”.

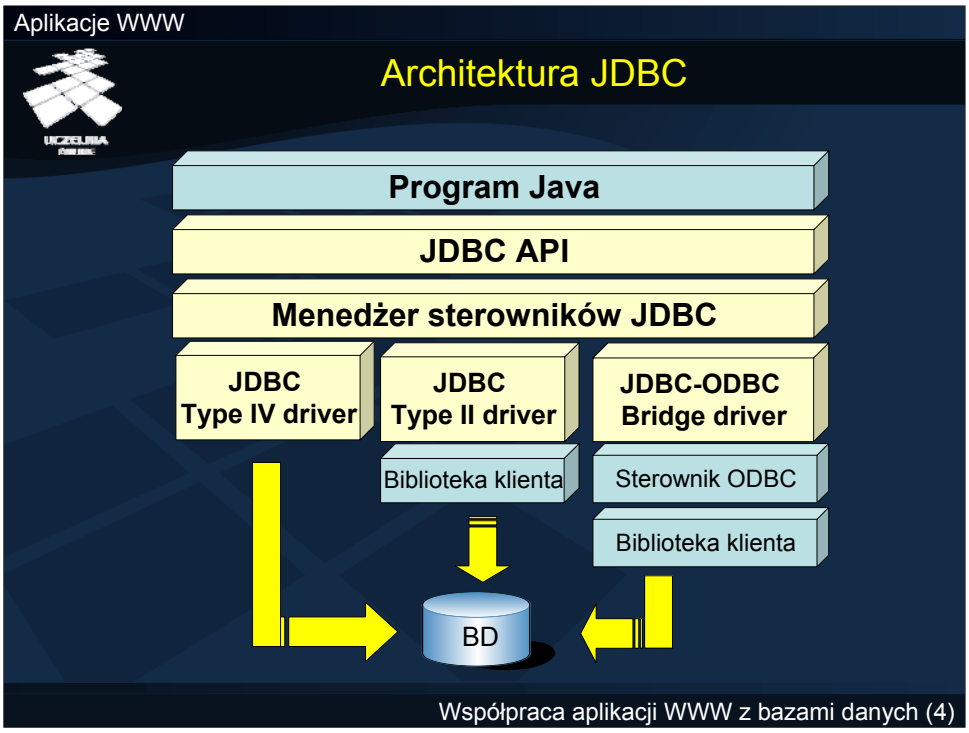


Aplikacje Java Enterprise Edition (Java EE), podobnie jak wszystkie aplikacje tworzone w języku Java, komunikują się z bazą danych poprzez interfejs JDBC. Ręczne kodowanie w JDBC jest uciążliwe i podatne na błędy, choćby dlatego, że w JDBC polecenia SQL są zapisane w łańcuchach znaków i ich treść nie jest weryfikowana na etapie kompilacji aplikacji. Odpowiedzią na wady JDBC miał być standard SQLJ, umożliwiający zagnieżdżanie poleceń SQL w kodzie Java za pomocą specjalnych konstrukcji, przetwarzanych przed kompilacją translatorem SQLJ. SQLJ nie zyskał jednak popularności i jest dziś prawie martwym standardem. Głównie dlatego, że twórcy dużych aplikacji w języku Java, w tym aplikacji Java EE, nie kodują bezpośrednio w JDBC, ale wykorzystują technologie oparte o JDBC, ale działające na wyższym poziomie abstrakcji.

Dostęp do bazy danych w aplikacjach WWW opartych o JSP może być realizowany poprzez bibliotekę znaczników JSTL SQL. Jest to jednak rozwiązanie akceptowalne tylko dla bardzo prostych aplikacji, gdyż narusza zasadę separacji logiki biznesowej od logiki prezentacji.

Lansowanym przez wiele lat „oficjalnym” mechanizmem komunikacji z bazą danych w aplikacjach Java EE były encyjne komponenty EJB. Ich architektura nie wykorzystywała jednak możliwości systemów baz danych, przez co korzystanie z nich było nienaturalne, a sama technologia w praktyce nieefektywna. Wraz z pojawieniem się specyfikacji EJB 3.0, encyjne komponenty EJB zostały zarzucone i zastąpione w ramach technologii Java EE przez Java Persistence.

Jako alternatywę dla problematycznych encyjnych EJB opracowano szereg „lżejszych” technologii odwzorowania obiektowo-relacyjnego (w tym m.in. Hibernate i Toplink), które zyskały dużą popularność. Nowy standard Java Persistence, opracowany wspólnie z EJB 3.0, oparty jest o technologie odwzorowania obiektowo-relacyjnego i standaryzuje API do obsługi trwałości danych w aplikacjach Java.



JDBC definiuje standardowy interfejs programistyczny (API) do współpracy aplikacji Java z relacyjną bazą danych.

Interfejs JDBC jest implementowany przez sterowniki JDBC (ang. JDBC drivers), obsługujące poszczególne systemy zarządzania bazą danych. Wyróżniono cztery typy architektur sterowników JDBC:

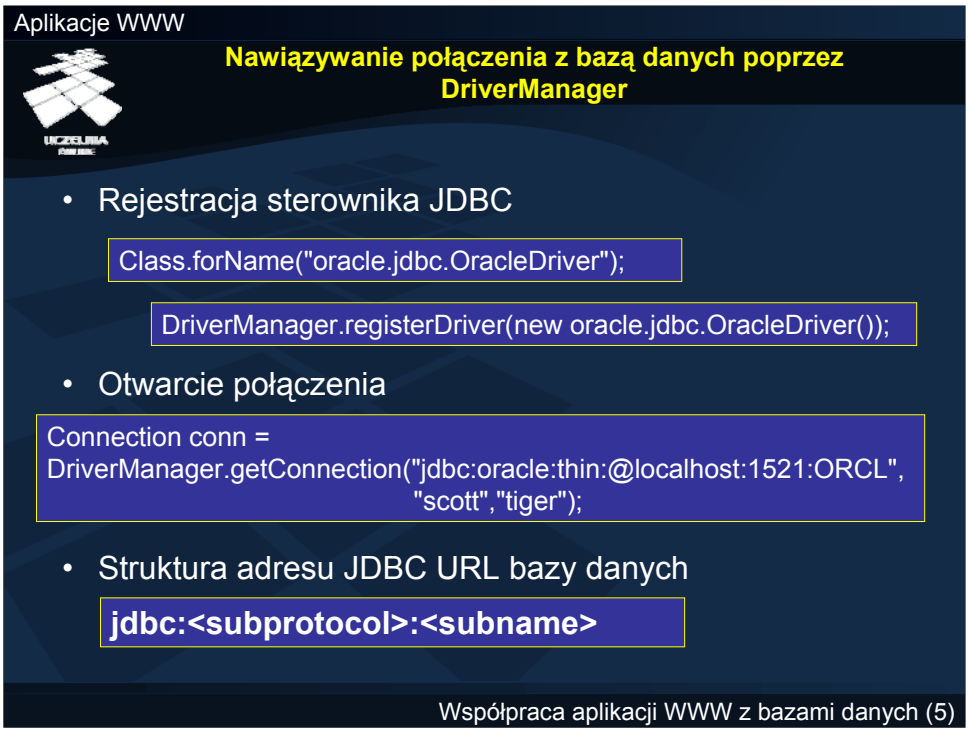
Pierwszy typ (Type I) to sterownik mający postać mostu JDBC-ODBC. Umożliwia on połączenie z każdą bazą danych, dla której istnieje sterownik ODBC. Ten sterownik ma znaczenie historyczne i był używany przed pojawieniem się dedykowanych sterowników JDBC.

Typ drugi (Type II) to sterownik napisany w Javie, ale wymagający biblioteki klienta bazy danych najczęściej napisanej w innym języku. Jest to efektywne rozwiązanie, ale wymaga preinstalowanego oprogramowania klienta bazy danych.

Typ czwarty (Type IV) to sterownik napisany w całości w Javie, komunikujący się bezpośrednio z serwerem bazy danych. W początkach technologii Java to rozwiązanie było uważane za mniej efektywne niż sterownik typu II, ale dziś gdy dostępne są efektywne maszyny wirtualne Java, argument ten stracił na znaczeniu i sterowniki typu czwartego są powszechnie używane.

Zaproponowano również rozwiązanie określane jako typ trzeci (Type III), z uniwersalnym sterownikiem napisanym w czystej Javie i obsługą poszczególnych rodzajów baz danych w warstwie pośredniej.

Sterowniki JDBC dostępne dla aplikacji są zarządzane przez menedżera sterowników, który w oparciu o parametry połączenia z bazą danych wybiera odpowiedni sterownik do komunikacji z bazą danych.

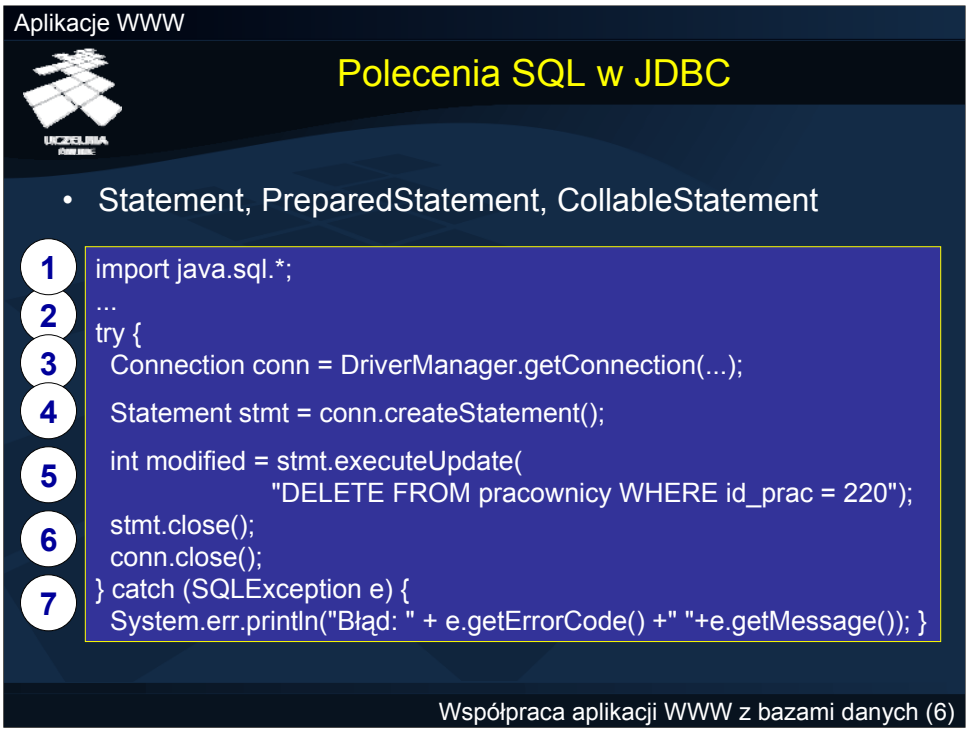


Pierwotny sposób nawiązywania połączeń z bazą danych w JDBC to wykorzystanie do tego celu menedżera sterowników, dostępnego w formie klasy DriverManager.

Przed otwarciem połączenia należy w menedżerze sterowników zarejestrować sterownik JDBC dla bazy danych, z którą aplikacja będzie się łączyć, jednym z dwóch pokazanych na slajdzie sposobów. Pierwszy, preferowany sposób, wykorzystuje fakt, że sterowniki JDBC są implementowane tak aby same rejestrowały się w menedżerze sterowników po załadowaniu ich klasy przez maszynę wirtualną Java. Drugi sposób to jawne utworzenie instancji klasy sterownika i zarejestrowanie go metodą klasy DriverManager. Przykłady na slajdzie pokazują sposób rejestracji sterownika JDBC dla Oracle (klasa oracle.jdbc.OracleDriver). Biblioteka (plik JAR) ze sterownikiem JDBC musi być dostępna na ścieżce CLASSPATH aplikacji.

Otwarcie połączenia następuje w wyniku wywołania metody getConnection() klasy DriverManager. Metoda ta jest przeciążona, jeden z możliwych zestawów parametrów to JDBC URL wskazujący bazę danych oraz nazwa użytkownika i hasło. Metoda getConnection() zwraca instancję klasy java.sql.Connection.

JDBC URL składa się z trzech członów: pierwszy to zawsze „jdbc”, następnie po dwukropku podprotokół określający typ systemu, a po nim znowu dwukropek i nazwa lub łańcuch połączenia wskazujący konkretną bazę danych. Przykład na slajdzie otwiera połączenie z bazą danych Oracle poprzez sterownik typu IV, określany jako cienki klient (ang. thin driver). W tym wypadku, łańcuch połączenia wskazuje serwer, numer portu procesu nasłuchu bazy danych i nazwę instancji bazy danych.



Do wykonywania poleceń SQL w JDBC służą klasy Statement, Preparedstatement i CallableStatement. Obiekty Statement umożliwiają wykonywanie zapytań lub operacji DML (INSERT, UPDATE, DELETE) i DDL (np. CREATE TABLE). Obiekty klasy PreparedStatement (dziedziczącej ze Statement) służą do wykonywania poleceń prekompilowanych, z możliwością ich parametryzacji i wielokrotnego wykonania dla różnych wartości parametrów. Obiekty klasy CallableStatement (dziedziczącej z PreparedStatement) służą do wywoływania procedur i funkcji składowanych w bazie danych, z zachowaniem możliwości parametryzacji poleceń.

Na slajdzie pokazano przykład wykonania polecenia DELETE. Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Import pakietu java.sql, zawierającego podstawowe klasy i interfejsy JDBC.

2. Metody klas JDBC rzucają wyjątek SQLException, dlatego kod korzystający z JDBC został zawarty w bloku try.

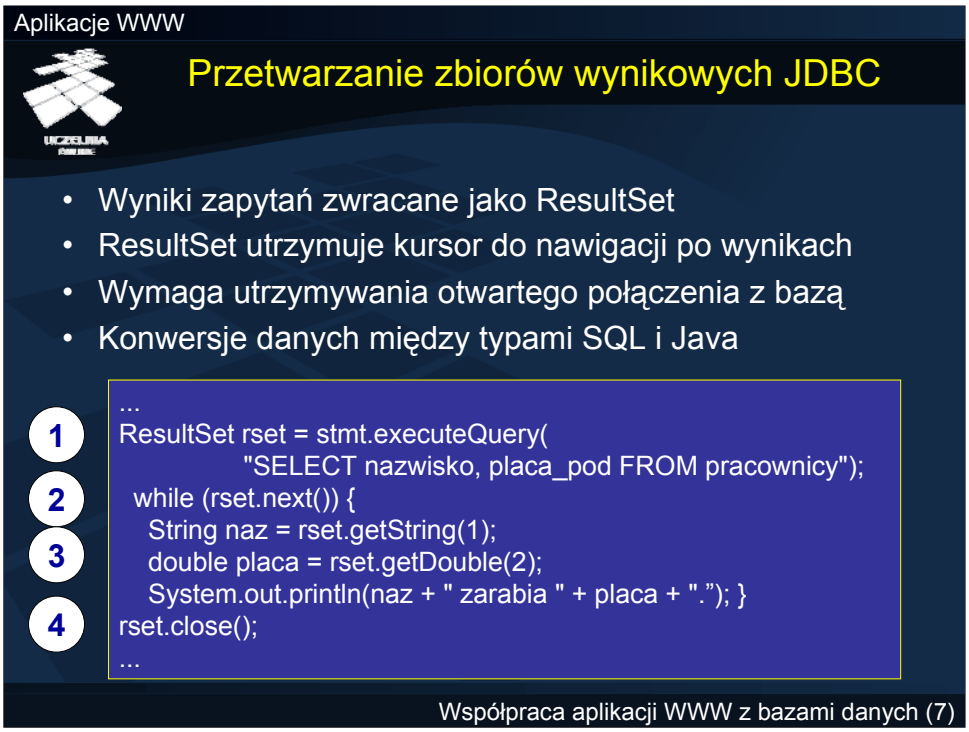
3. Otwarcie połączenia z bazą danych.

4. Utworzenie obiektu Statement, poprzez który można będzie wykonywać niesparametryzowane polecenia SQL.

5. Wykonanie polecenia DELETE metodą executeUpdate(), służącą do wykonywania poleceń DML i DDL. Metoda ta dla instrukcji DML zwraca liczbę wierszy, których dotyczyła operacja, a 0 dla instrukcji DDL.

6. Zwolnienie zasobów poprzez jawne wywołanie metody close() na rzecz obiektów Statement i Connection. Nie należy polegać w tym względzie na garbage collection.

7. Przechwycenie wyjątku SQLException i jego obsługa poprzez wyświetlenie komunikatu o błędzie, zawierającego odczytane z obiektu wyjątku: kod i tekstowy komunikat o błędzie zwrócone przez serwer bazy danych.



Do wykonywania zapytań SELECT służy metoda executeQuery() obiektu Statement. Metoda ta zwraca obiekt ResultSet, udostępniający tabelę wierszy wyniku zapytania. ResultSet utrzymuje kursor do nawigacji po wierszach wyniku i wymaga utrzymywania otwartego połączenia z bazą danych. Kursor startuje od pozycji przed pierwszym rekordem zbioru wynikowego i domyślnie jest jednokierunkowy. Od wersji JDBC 2.0 istnieje możliwość zwrócenia przez zapytanie obiektu ResultSet umożliwiającego swobodną nawigację po wierszach wyniku zapytania, a także modyfikację danych poprzez kursor.

Odczyt poszczególnych atrybutów bieżącego wiersza w kursorze jest realizowany metodami getXXX(), określającymi typ danych języka Java, do którego ma zostać skonwertowana wartość odczytana z bazy danych. Parametrem metod getXXX() jest nazwa kolumny lub pozycja wyrażenia w klauzuli SELECT zapytania.

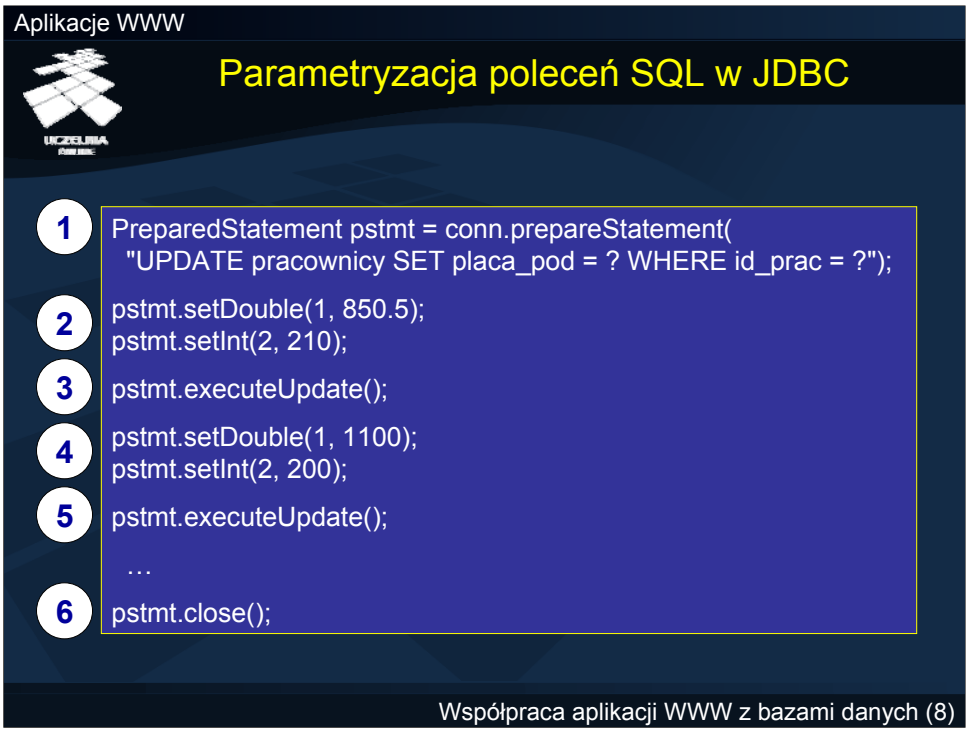
Przykładowy fragment kodu na slajdzie odczytuje i wyświetla na konsoli nazwiska i płace pracowników. Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Wykonanie zapytania poprzez wywołanie metody executeQuery() na rzecz utworzonego wcześniej obiektu Statement. Metoda zwraca zbiór wynikowy ResultSet.

2. Kolejne wiersze pobierane są z obiektu ResultSet w pętli while. Na początku kursor znajduje się przed pierwszym wierszem wyniku napytania. Przejście do kolejnego wiersza jest realizowane metodą next(), zwracającą prawdę gdy znaleziono następny wiersz, a fałsz w przeciwnym wypadku. Gdy metoda next() zwróci fałsz nastąpi wyjście z pętli.

3. Odczyt nazwiska (pierwszego na liście wyrażeń SELECT) jako String i płacy (drugiej na liście wyrażeń SELECT) jako double.

4. Zamknięcie obiektu DataSet po przetworzeniu wyników zapytania. Ważne z punktu widzenia oszczędzania zasobów serwera, gdyż zwalnia kursor.



Aplikacje WWW często wykonują polecenia SQL, w których treść wstawiane są wartości wprowadzone przez użytkownika. Typowe sytuacje to wstawianie do bazy danych wartości wprowadzonych do formularzy czy zawężanie wyników zapytania w oparciu o warunki selekcji podane przez użytkownika. Wklejanie podanych przez użytkownika wartości do treści poleceń SQL za pomocą konkatenacji łańcuchów znaków jest niebezpieczne, gdyż umożliwia złośliwym użytkownikom modyfikację struktury polecenia SQL i wykonanie operacji, których aplikacja nie miała z założenia umożliwiać. Ta technika włamań do systemu jest znana pod nazwą SQL injection. Bezpiecznym sposobem wykonywania sparametryzowanych poleceń SQL w JDBC jest wykorzystanie obiektów PreparedStatement.

Polecenia PreparedStatement mają jeszcze jedną zaletę. Są prekompilowane i w przypadku wielokrotnego wykonania polecenia dla różnych wartości parametrów mogą być efektywniejsze niż polecenia Statement. Zależy to jednak od konkretnej platformy.

Na slajdzie pokazano przykład wykorzystania PreparedStatement do zmodyfikowania dwóch wierszy tabeli. Znaczenie wyróżnionych fragmentów kodu jest następujące:

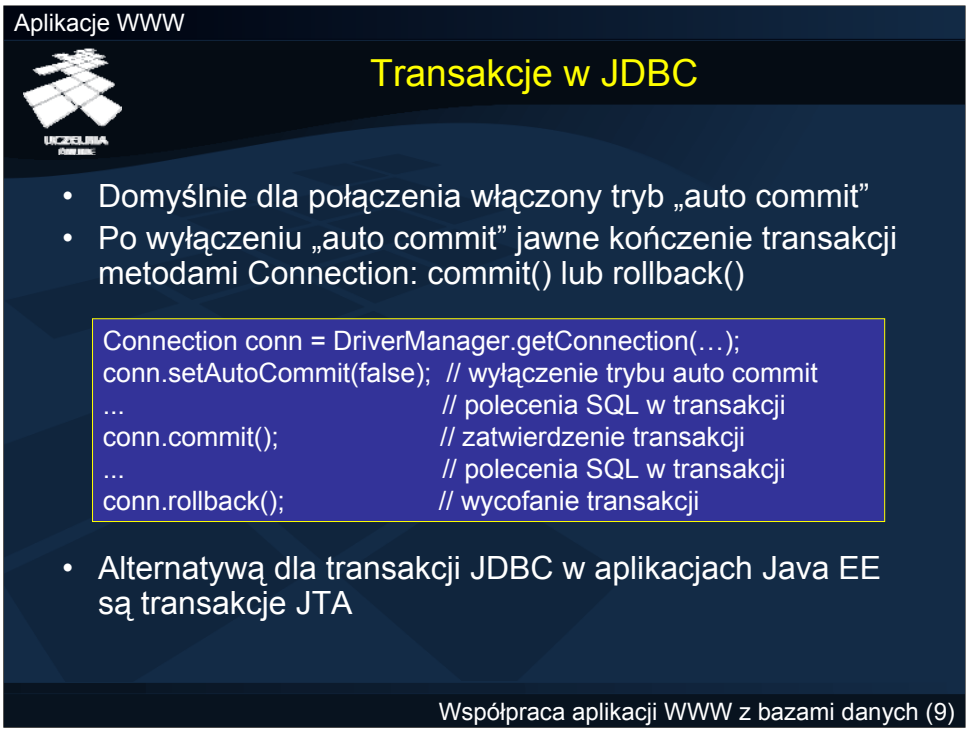
1. Utworzenie sparametryzowanego obiektu PreparedStatement. Parametry są umieszczane w treści polecenia jako znaki zapytania. W przykładzie, parametrami są ustawiana wartość płacy i identyfikator pracownika, którego płaca ma być zmieniona.

2. Ustalenie wartości parametrów dla polecenia. Do tego celu służą metody setXXX() dla poszczególnych typów języka Java. Parametrami są numer parametru w zapytaniu i wartość. Do ustawienia wartości NULL dla parametru służy metoda setNull().

3. Uruchomienie polecenia – metodą executeUpdate(), gdyż polecenie w przykładzie nie jest zapytaniem. Dla zapytania byłaby użyta metoda executeQuery().

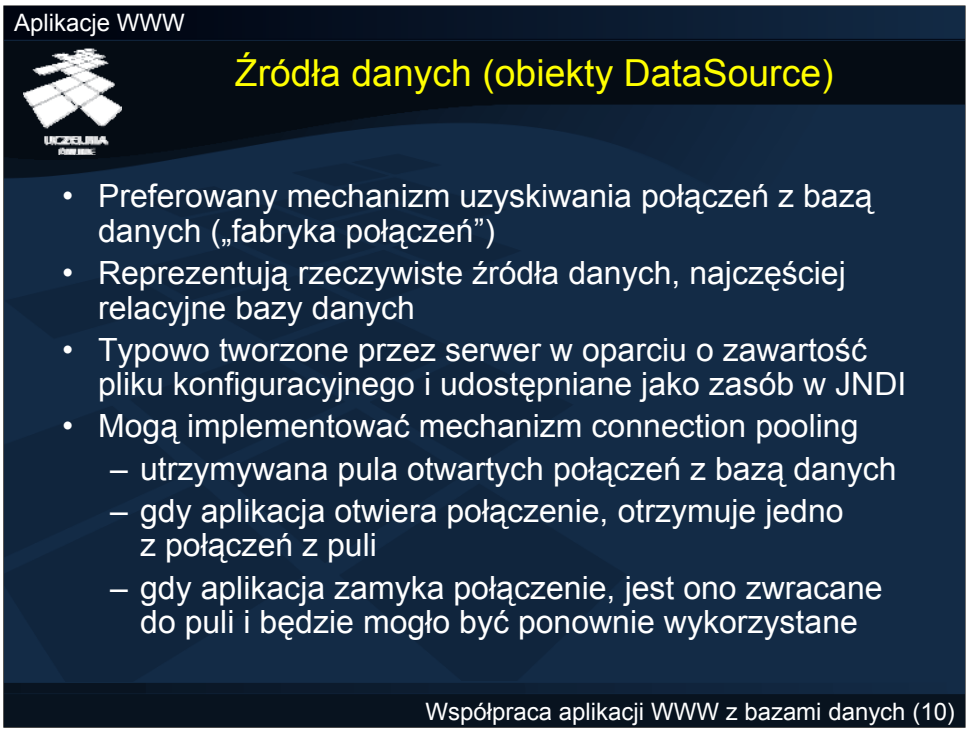
4. i 5. Powtórzenie kroków 2 i 3 dla innych wartości parametrów.

6. Zamknięcie obiektu PreparedStatement.



Domyślnie operacje na bazie danych poprzez obiekt Connection realizowane są w trybie „auto commit”, gdzie każde polecenie stanowi odrębną transakcję i jest automatycznie zatwierdzane po wykonaniu. W celu realizacji transakcji na poziomie JDBC, obejmującej więcej niż jedno polecenie, należy najpierw wyłączyć tryb „auto commit” dla połączenia metodą setAutoCommit(). Po wyłączeniu trybu „auto commit”, każdą transakcję należy jawnie zakończyć poprzez jej zatwierdzenie metodą commit() lub wycofanie metodą rollback() obiektu Connection. Nie ma w JDBC wyróżnionej metody rozpoczynającej transakcję. Pierwsze polecenie SQL wykonane po wyłączeniu trybu „auto commit” lub zakończeniu poprzedniej transakcji rozpoczyna nową transakcję. Realizację transakcji na poziomie JDBC ilustruje fragment kodu przedstawiony na slajdzie.

Alternatywą dla transakcji JDBC w aplikacjach Java EE są transakcje realizowane poprzez interfejs JTA (Java Transaction API). Transakcje JTA będą omówione w trakcie wykładu na temat komponentów EJB w ramach przedmiotu „Zaawansowane aplikacje internetowe”.

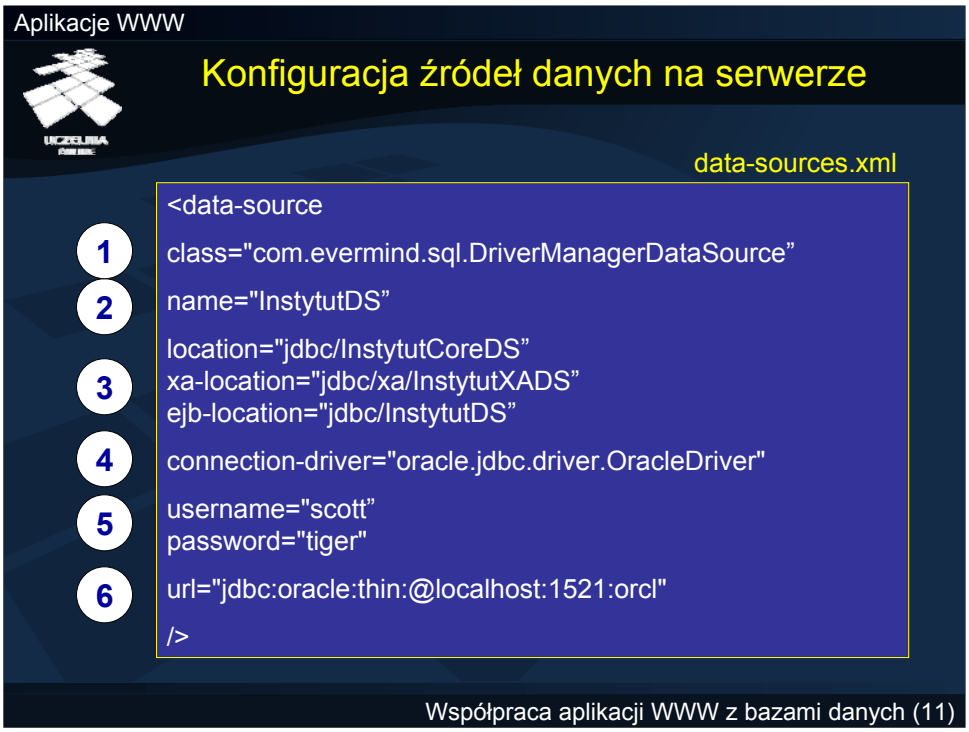


Preferowanym od wersji JDBC 2.0 mechanizmem uzyskiwania połączeń z bazą danych przez aplikacje Java są źródła danych (obiekty DataSource). Obiekt DataSource reprezentuje rzeczywiste źródło danych, najczęściej relacyjną bazę danych i może być postrzegany jako „fabryka połączeń” z bazą danych.

Typowo źródła danych są konfigurowane deklaratywnie przez administratora serwera aplikacji. Serwer tworzy je w oparciu o zawartość pliku konfiguracyjnego i udostępnia aplikacjom poprzez serwis nazw JNDI (Java Naming and Directory Interface). Dzięki takiemu podejściu, fizyczne parametry źródła danych nie są zaszyte w kodzie aplikacji, ale zawarte w pliku konfiguracyjnym serwera.

Implementacje DataSource mogą obsługiwać mechanizm connection pooling, pozwalający zredukować czas poświęcany przez aplikacje na otwieranie połączenia z bazą danych. Mechanizm ten jest szczególnie ważny w aplikacjach WWW, których komponenty typowo wykonują proste operacje na bazie danych. Redukcja czasu spędzanego na otwarcie połączenia może znacząco skrócić czas przeznaczany w nich na komunikację z bazą danych.

Mechanizm connection pooling polega na utrzymywaniu puli otwartych połączeń z bazą danych. Minimalna i maksymalna liczba połączeń w puli to typowo parametry konfiguracyjne źródła danych. Gdy aplikacja otwiera połączenie, otrzymuje jedno z połączeń z puli. Gdy aplikacja zamyka połączenie, jest ono zwracane do puli i będzie mogło być ponownie wykorzystane. Ważne jest aby aplikacja jawnie zamykała połączenie, gdy już go nie potrzebuje, tak aby to samo połączenie było możliwie szybko dostępne dla obsługi innych żądań.



Slajd pokazuje przykład konfiguracji źródła danych w pliku konfiguracyjnym serwera aplikacji (składnia dla serwera OC4J). Typowa nazwa pliku zawierającego konfigurację źródeł danych to data-sources.xml. Znaczenie poszczególnych elementów przykładowej definicji jest następujące:

1. Nazwa klasy źródła danych.

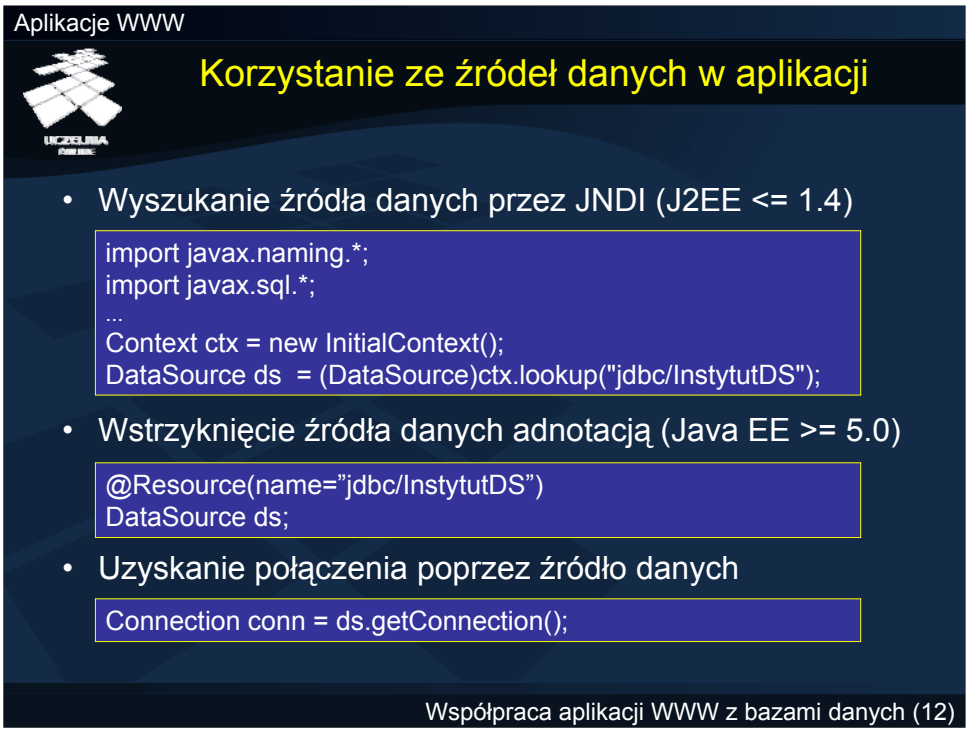
2. Logiczna nazwa źródła na serwerze. Pod tą nazwą źródło będzie widoczne w narzędziach administracyjnych serwera.

3. Nazwy (lokalizacje) JNDI, pod którymi będzie dostępne źródło danych. Źródło DriverManagerDataSource wymaga podania trzech lokalizacji, odpowiadających różnym poziomom funkcjonalności źródła. W praktyce obecnie wykorzystywana jest nazwa ejb-location.

4. Klasa sterownika JDBC, poprzez który źródło będzie łączyć się z bazą danych.

5. Nazwa użytkownika w bazie danych i jego hasło.

6. JDBC URL wskazujący bazę danych.



W celu uzyskania połączenia z bazą danych, aplikacja musi najpierw uzyskać referencję do obiektu źródła danych (DataSource). Do wersji Java EE 1.4 (określanej jeszcze skrótem J2EE) aplikacja wyszukiwała źródło danych poprzez jawną operację lookup w JNDI. W tym celu, aplikacja najpierw musiała uzyskać obiekt kontekstu serwisu nazw, a następnie na jego rzecz wywołać metodę lookup(), przekazując nazwę JNDI źródła danych jako parametr. Klasy kontekstów są zawarte w pakiecie javax.naming, a sam interfejs DataSource w pakiecie javax.sql.

Zalecany obecnie (tj. od wersji Java EE 5.0) sposób przekazania źródła danych aplikacji to wstrzyknięcie go za pomocą adnotacji @Resource. Na slajdzie pokazano jedną z możliwości wstrzyknięcia źródła danych, gdzie jest ono wstrzykiwane do pola w klasie, a nazwa JNDI źródła została podana jawnie w adnotacji.

W celu uzyskania obiektu Connection należy na rzecz obiektu DataSource wywołać metodę getConnection(). Obiekt Connection, reprezentujący połączenie z bazą danych, uzyskany poprzez źródło danych, jest wykorzystywany w aplikacji tak samo jak w przypadku uzyskania go poprzez klasę DriverManager.

Implementacja aplikacji Java pracujących na relacyjnej bazie danych na poziomie interfejsu JDBC jest czasochłonna i uciążliwa. Problem stanowi niski poziom abstrakcji interfejsu JDBC i różnice w organizacji danych między obiektowym językiem Java, a relacyjnymi bazami danych. Lansowana przez specyfikację Java EE do wersji 1.4 jako rozwiązanie tego problemu technologia encyjnych EJB okazała się niepraktyczna i nieefektywna. Jako alternatywę, różne środowiska zaproponowały technologie automatyzujące odwzorowanie obiektów na poziomie programu Java w struktury relacyjne. Technologie te są określane jako technologie odwzorowania obiektowo-relacyjnego (Object-Relational Mapping – w skrócie O/RM). Można z nich korzystać również w celu uzyskania obiektowej reprezentacji danych dla istniejącego schematu relacyjnej bazy danych.

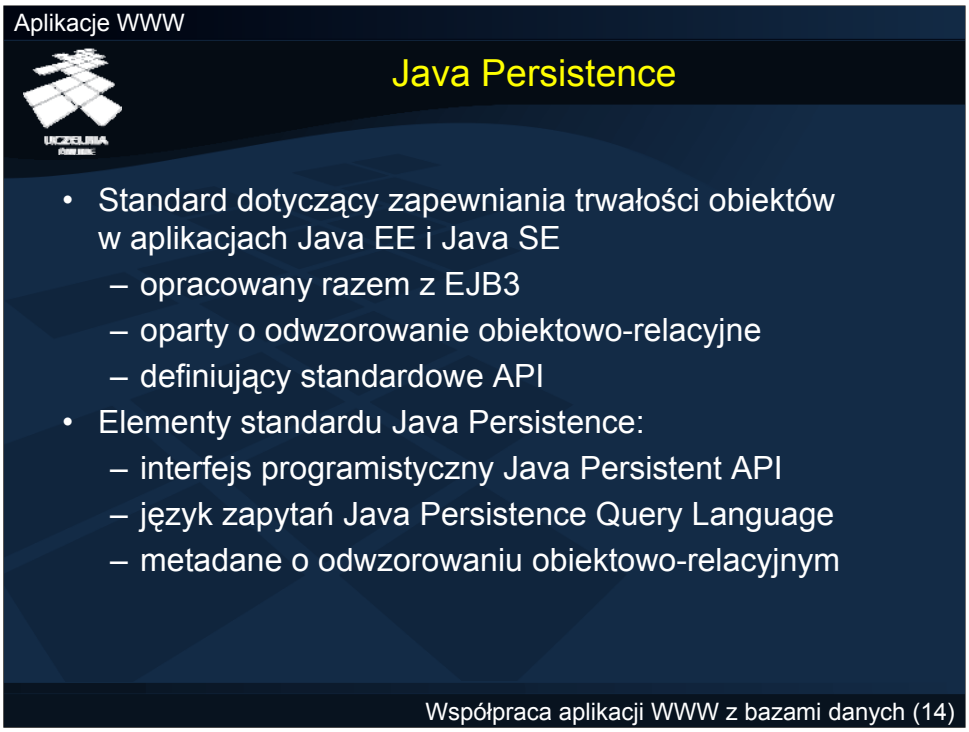
Elementy technologii O/RM to:

1. API do zarządzania trwałością obiektów;

2. Mechanizm specyfikowania metadanych opisujących odwzorowanie klas na relacje w bazach danych;

3. Język lub API do wykonywania zapytań.

Najpopularniejsze implementacje technologii odwzorowania obiektowo-relacyjnego dla aplikacji Java to Hibernate (rozwiązanie Open Source firmy JBoss) i Oracle Toplink (rozwiązanie firmowe firmy Oracle). Mniejszą popularność zyskała technologia JDO (firmy Sun).



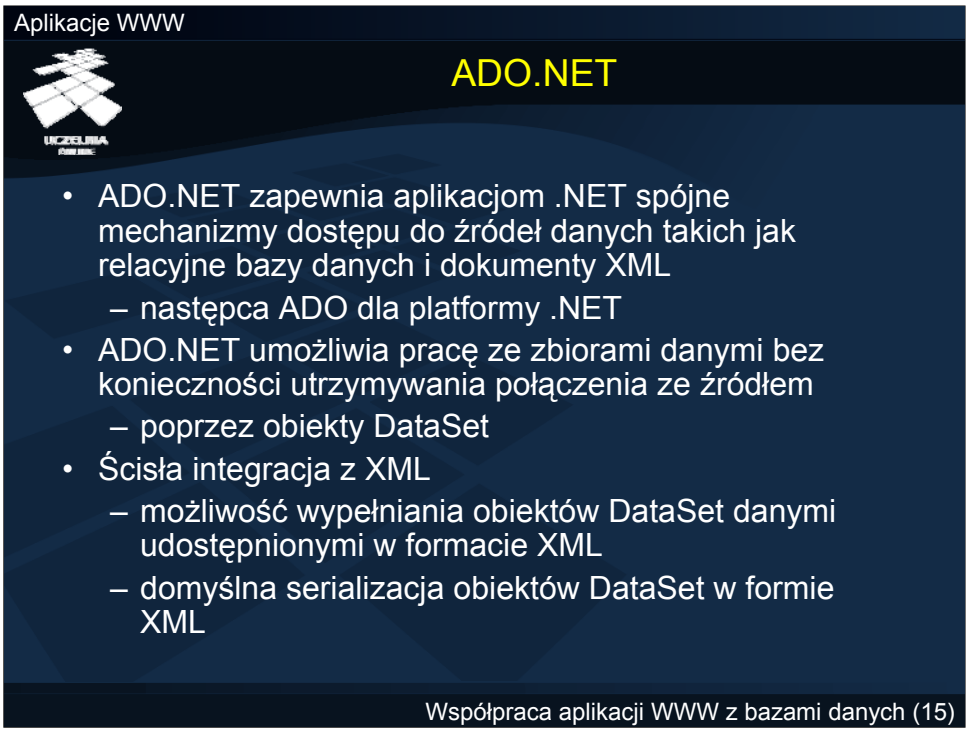
Java Persistence to nowy standard zapewniania trwałości obiektów w aplikacjach Java EE i Java SE, stanowiący część specyfikacji Java EE od wersji 5.0. Został on opracowany razem z EJB 3.0 w odpowiedzi na niepowodzenie lansowanej do tej pory koncepcji encyjnych EJB i niewątpliwy sukces technologii odwzorowania obiektowo-relacyjnego takich jak Hibernate czy Oracle Toplink. Technologie te, mimo że oparte o te same idee, różnią się jeśli chodzi o API. Standard Java Persistence jest oparty o odwzorowanie obiektowo-relacyjne i definiuje standardowe API do obsługi trwałości obiektów.

Elementy standardu Java Persistence to:

1. Interfejs programistyczny Java Persistence API, obejmujący interfejs do zarządcy trwałości EntityManager;

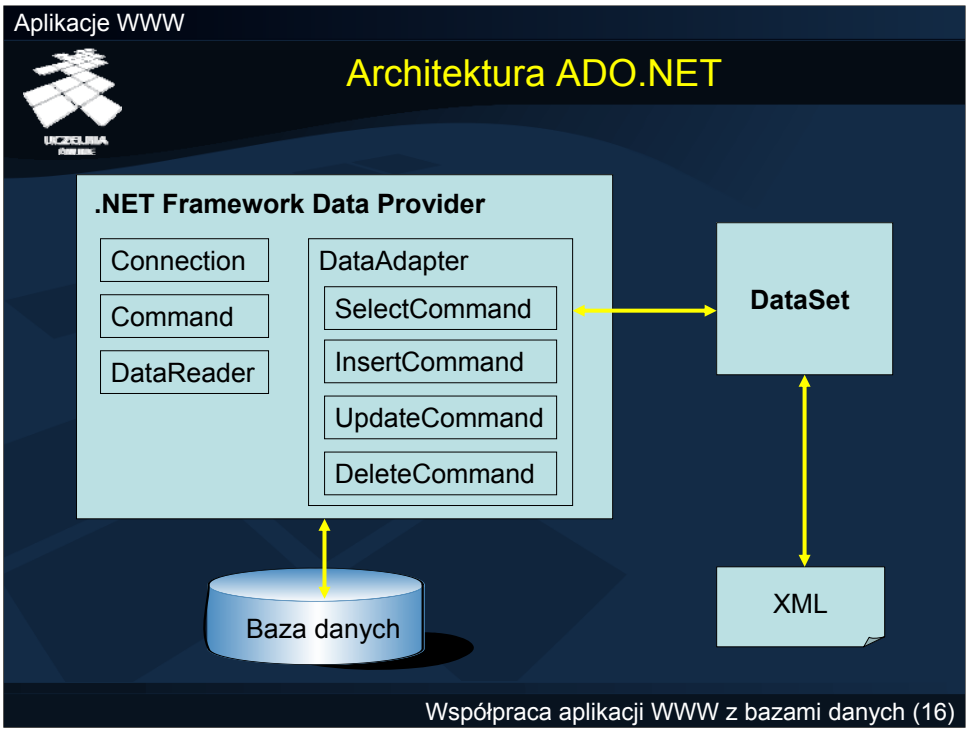
2. Język zapytań Java Persistence Query Language (JPQL), o składni przypominającej SQL, umożliwiający tworzenie przenaszalnych zapytań.

3. Metadane o odwzorowaniu obiektowo-relacyjnym, najczęściej umieszczone w kodzie w formie adnotacji, z możliwością ich nadpisania w środowisku produkcyjnym poprzez XML-owe pliki konfiguracyjne.



Biblioteka ADO.NET zapewnia aplikacjom .NET spójne mechanizmy dostępu do źródeł danych takich jak relacyjne bazy danych i dokumenty XML. ADO.NET jest następcą wcześniejszej technologii Microsoft - ADO (ActiveX Data Objects), dostosowanym do potrzeb aplikacji .NET. ADO.NET jest koncepcyjnie podobne do ADO, stanowiąc warstwę abstrakcji w dostępie do danych. Nową jakością w ADO.NET jest silna orientacja na umożliwienie pracy ze zbiorami danych bez konieczności utrzymywania połączenia ze źródłem danych. Możliwość ta jest kluczowa w internetowych aplikacjach wielowarstwowych, ze względu na charakterystyczną dla aplikacji WWW pracę w trybie żądanie-odpowiedź i migrację zbiorów danych między warstwami aplikacji. Praca w trybie „odłączonym” jest realizowana w ADO.NET poprzez obiekty DataSet, które mogą być wypełnione danymi z bazy danych, odłączone od bazy i przekazane do interfejsu użytkownika, modyfikowane po odłączeniu od bazy danych i w końcu synchronizowane z bazą danych.

ADO.NET jest ściśle zintegrowane z językiem XML. Klasy ADO.NET i klasy do obsługi XML były projektowane wspólnie i są częścią tej samej architektury. XML pełni też kluczową rolę w ADO.NET jako format przesyłania danych. Mimo, że wewnętrznie dane w obiektach DataSet nie są zapisywane w formie XML, to istnieje możliwość wypełnienia ich danymi ze źródeł XML (plikowych lub strumieniowych) i eksportu (serializacji) w formie XML.



Dwa podstawowe składniki architektury ADO.NET to zbiór danych - DataSet i dostawca danych - .NET Framework Data Provider.

DataSet zawiera kolekcję tabel danych i jest niezależny od źródła danych. Najczęściej obiekt DataSet służy do udostępnienia aplikacji danych z bazy danych. Obiekty DataSet mogą być wypełniane danymi ze źródeł XML i przesyłane w formie XML, co ułatwia ich przekazywanie np. w aplikacjach Web Services.

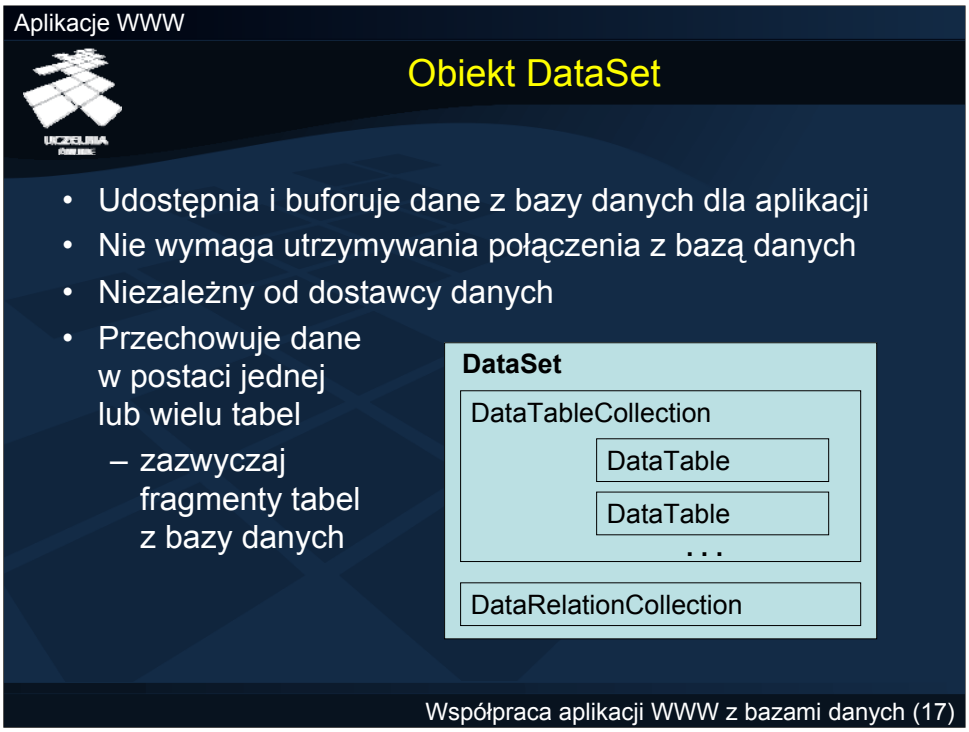
.NET Framework Data Provider to zbiór komponentów (klas) do odczytu i manipulacji danymi dla konkretnego typu źródła danych np. Microsoft SQL Server czy ODBC.

Obiekt Connection reprezentuje połączenie ze źródłem danych i służy do łączenia się ze źródłem danych określonym przez łańcuch połączenia (ang. connection string).

Obiekt Command umożliwia wykonanie w bazie danych poleceń SQL do odczytu danych, manipulacji danymi i uruchamiania procedur składowanych. Polecenia reprezentowane przez obiekty Command mogą być sparametryzowane.

Obiekt DataReader to lekki i szybki obiekt do odczytu danych ze źródła danych, stanowiący alternatywę dla oferującego zaawansowaną funkcjonalność obiektu DataSet. Jeśli dane po odczycie mają być następnie uaktualniane, buforowane lub udostępniane innym warstwom aplikacji np. poprzez Web Service, należy zamiast DataReader wykorzystać DataSet.

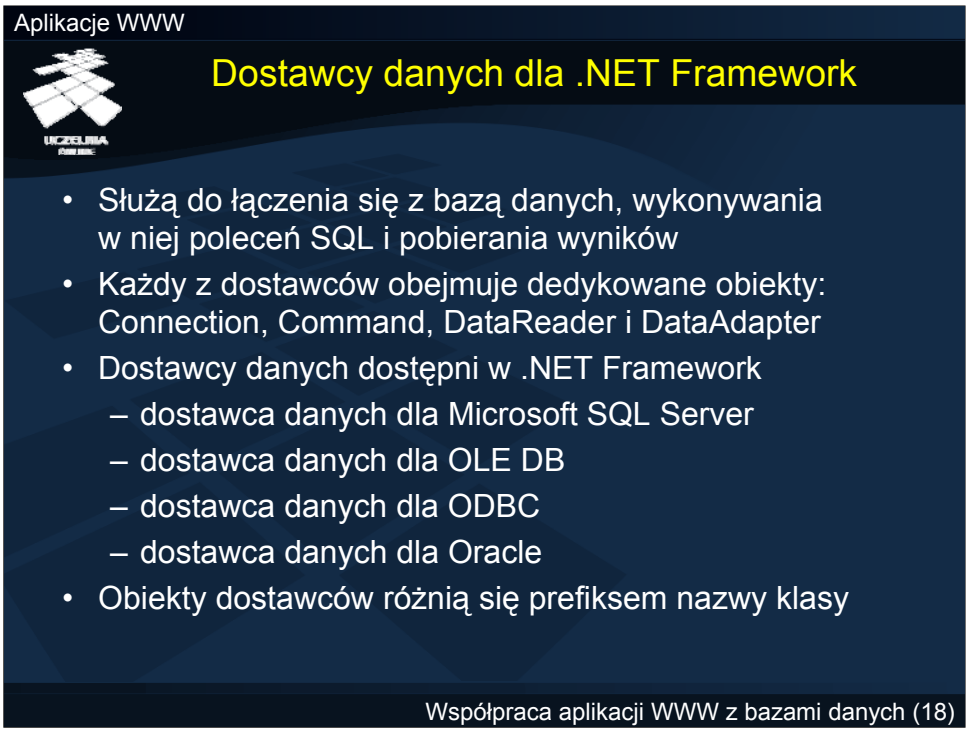
Obiekt DataAdapter stanowi pomost między źródłem danych a zbiorem danych DataSet. DataAdapter wykonuje polecenia w źródle danych poprzez obiekty Command, w celu wypełnienia obiektu DataSet danymi lub przeniesienia zmian dokonanych przez aplikację na obiekcie DataSet do źródła danych.



DataSet to obiekt kluczowy z punktu widzenia architektury ADO.NET, zorientowanej na zapewnienie dostępu do danych w rozproszonych, komponentowych aplikacjach .NET. DataSet udostępnia i buforuje dane z bazy danych dla aplikacji, nie wymagając utrzymywania otwartego połączenia z bazą danych. Praca w trybie „odłączonym" zwiększa skalowalność aplikacji poprzez zmniejszenie liczby otwartych połączeń z serwerem bazy danych na rzecz użytkowników aplikacji WWW. Połączenie z bazą danych jest otwierane na czas odczytu danych z bazy danych i uaktualniania bazy w celu odzwierciedlenia w niej zmian naniesionych w obiekcie DataSet gdy pozostawał on odłączony.

Obiekt DataSet jest niezależny od źródła danych, z którego dane udostępnia. Ta sama implementacja DataSet jest używana dla wszystkich dostawców danych.

Obiekt DataSet przechowuje dane w postaci tabelarycznej, analogicznie do relacyjnych baz danych. DataSet zawiera kolekcję DataTableCollection, obejmującą jedną lub więcej tabel (DataTable), których nazwy i struktura nie muszą wprost odpowiadać tabelom z bazy danych. Zazwyczaj DataSet zawiera fragmenty tabel z bazy danych. Każdy obiekt DataTable w ramach kolekcji tabel zawiera kolekcje wierszy, kolumn i ograniczeń integralnościowych dla tabeli. Oprócz kolekcji DataTableCollection, obiekt DataSet zawiera również kolekcję DataRelationCollection z powiązaniami między wierszami tabel, analogicznymi do powiązań między kolumnami kluczy obcych i kluczy głównych w relacyjnej bazie danych.

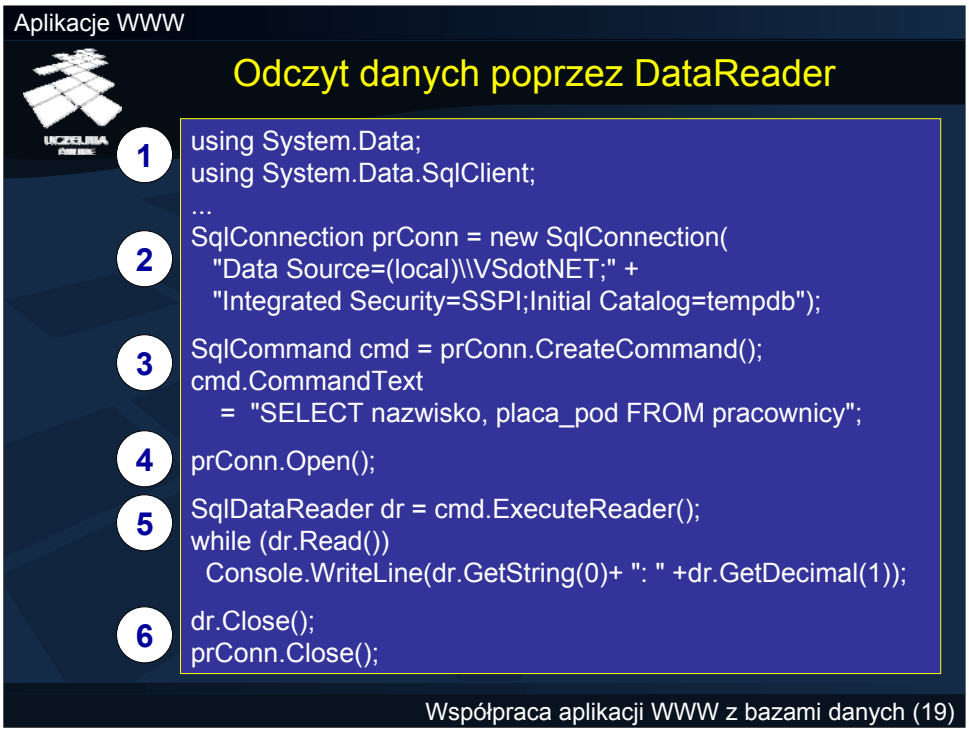


Dostawcy danych dla .NET Framework służą do łączenia się z bazą danych, wykonywania w niej poleceń SQL i pobierania wyników. Wyniki zapytań mogą być przetwarzane bezpośrednio lub buforowane w obiektach DataSet.

Każdy z dostawców obejmuje dedykowane obiekty: Connection, Command, DataReader i DataAdapter.

Dostawcy danych dostępni w .NET Framework to: dostawca danych dla Microsoft SQL Server, dostawca danych dla OLE DB, dostawca danych dla ODBC i dostawca danych dla Oracle. Dostawcy danych dla ODBC i OLE DB są dostawcami uniwersalnymi, umożliwiającymi obsługę każdej bazy danych, dla której jest dostępny sterownik ODBC lub dostawca OLE DB. Większą efektywność oferują dostawcy dedykowani, dostępni w .NET Framework dla SQL Server i Oracle. Dedykowanych dostawców dla innych systemów zarządzania bazą danych udostępniają producenci tych systemów.

Obiekty dostawców różnią się prefiksem nazwy klasy np. poszczególne wersje obiektu Connection dostępne są jako klasy: SqlConnection, OleDbConnection, OdbcConnection, OracleConnection.



Slajd przedstawia przykład odczytu danych z tabeli w bazie danych SQL Server poprzez obiekt DataReader. Znaczenie poszczególnych fragmentów kodu jest następujące:

1. Deklaracje wykorzystywanych przestrzeni nazw: System.Data dla klas ADO.NET niezależnych od dostawcy danych i System.Data.SqlClient dla klas dostawcy danych dla Microsoft SQL Server.

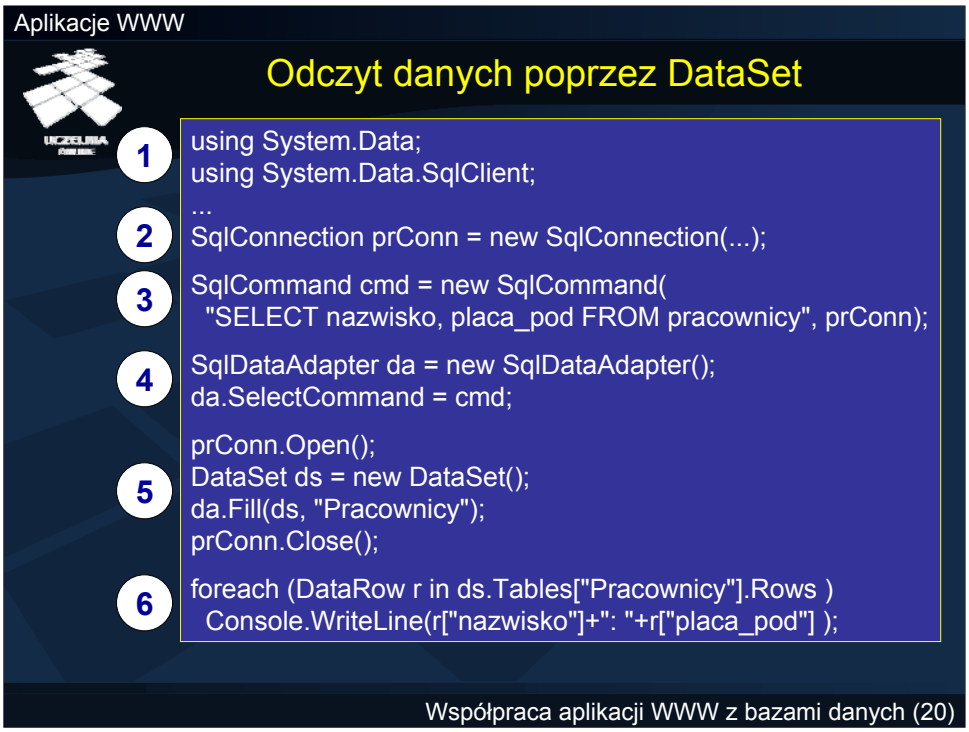
2. Utworzenie obiektu SqlConnection, reprezentującego połączenie z bazą danych SQL Server. Poszczególne składniki łańcucha połączenia mają następujące znaczenie: Data Source wskazuje serwer i instancję SQL Server, Integrated Security oznacza, że do logowania do serwera bazy danych będzie wykorzystana tożsamość z systemu Windows (dlatego łańcuch połączenia nie zawiera nazwy użytkownika i hasła), Initial Catalog wskazuje bazę danych na wybranej instancji SQLServer.

3. Utworzenie obiektu SqlCommand, reprezentującego zapytanie wybierające nazwiska i płace z tabeli Pracownicy.

4. Otwarcie połączenia z bazą danych.

5. Wykonanie zapytania do bazy danych i zwrócenie jego wyniku w formie obiektu DataReader. Następnie odczyt w pętli kolejnych wierszy wyniku zapytania i wyświetlenie dla każdego wiersza nazwiska odczytanego jako obiekt String i płacy jako obiekt Decimal.

6. Zamknięcie obiektu DataReader, a następnie połączenia z bazą danych, w celu zwolnienia zasobów.



Slajd przedstawia przykład odczytu danych z tabeli w bazie danych SQL Server poprzez obiekt DataSet. Znaczenie poszczególnych fragmentów kodu jest następujące:

1. Deklaracje przestrzeni nazw jak w poprzednim przykładzie.

2. Utworzenie obiektu SqlConnection jak w poprzednim przykładzie.

3. Utworzenie obiektu SqlCommand, reprezentującego zapytanie wybierające nazwiska i płace z tabeli Pracownicy, w sposób równoważny temu z poprzedniego przykładu. Tym razem do utworzenia obiektu SqlCommand użyty został konstruktor, pozwalający na utworzenie obiektu SqlCommand w ramach połączenia z bazą danych i związanie z nim treści polecenia SQL w jednej instrukcji programu.

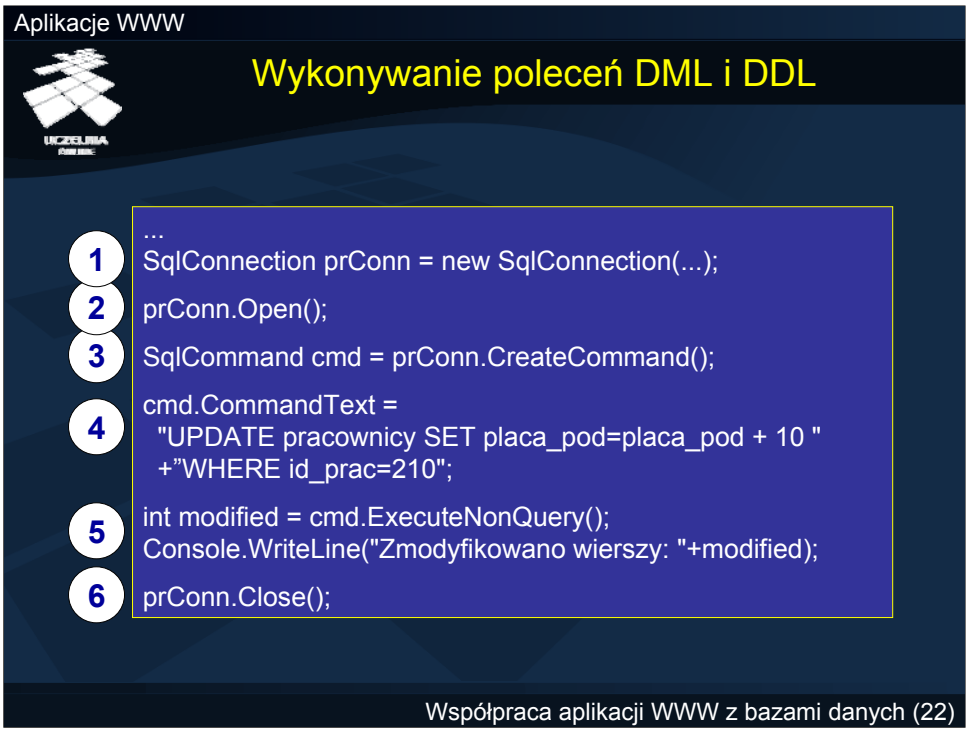
4. Utworzenie obiektu SqlDataAdapter, który będzie wykorzystany do umieszczenia wyników zapytania w obiekcie DataSet. Dla utworzonego obiektu SqlDataAdapter podana jest tylko treść polecenia SELECT, gdyż w przykładzie obiekt ten nie będzie wykorzystany do aktualizacji danych w bazie.

5. Otwarcie połączenia z bazą danych, utworzenie obiektu DataSet, wypełnienie obiektu DataSet danymi poprzez SqlDataAdapter i zamknięcie połączenia z bazą. W wywołaniu metody Fill(), umieszczającej dane w obiekcie DataSet, jako drugi parametr podana została nazwa tabeli, w której odczytane dane mają być zapamiętane w obiekcie DataSet. Połączenie z bazą mogło zostać zamknięte zaraz po wypełnieniu obiektu DataSet danymi, gdyż DataSet umożliwia pracę w trybie „odłączonym”.

6. Odczyt w pętli kolejnych wierszy wyniku zapytania i wyświetlenie dla każdego wiersza nazwiska i płacy. Tabele zawarte w obiekcie DataSet są dostępne poprzez ich nazwy operatorem indeksu na rzecz składowej Tables. Wiersze tabeli są w tabeli zawarte jako kolekcja Rows obiektów DataRow. Obiekt DataRow udostępnia poszczególne atrybuty poprzez ich nazwy operatorem indeksu.

ASP.NET oferuje kilka kontrolek umożliwiających łatwą prezentację danych z bazy danych umieszczonych w obiekcie DataSet. Jedną z nich jest kontrolka DataGrid, prezentująca dane w postaci tabelarycznej. Kontrolkę DataGrid po odpowiedniej konfiguracji można również wykorzystać do aktualizacji danych. Źródłem danych dla kontrolki DataGrid może być również obiekt DataReader, z tą różnicą, że DataReader jest odpowiedni tylko do odczytu danych.

U góry slajdu pokazany został sposób umieszczenia kontrolki DataGrid na stronie ASP.NET. Oczywiście w przypadku tworzenia aplikacji w środowisku Visual Studio, kontrolkę można umieścić na stronie pobierając ją z palety techniką drag-and-drop. U dołu slajdu pokazany został fragment kodu w języku C# wiążący kontrolkę DataGrid z jedną z tabel dostępnych w obiekcie DataSet. Kod ten typowo byłby umieszczony w pliku Code Behind obsługującym stronę np. w metodzie Page_Load(). Wyróżnione linie w kodzie to wskazanie tabeli Pracownicy z obiektu DataSet jako źródła danych dla kontrolki DataGrid na stronie i wywołanie metody wiążącej kontrolkę ze wskazanym wcześniej źródłem danych. Otwarcie połączenia z bazą danych i wypełnienie obiektu DataSet danymi z bazy danych odbywa się jak w przykładzie na poprzednim slajdzie.



Obiekt Command, wykorzystany w jednym z wcześniejszych przykładów do odczytu wyników zapytania SELECT do obiektu DataReader, umożliwia również wykonywanie poleceń DML (INSERT, UPDATE, DELETE) i DDL (np. CREATE TABLE). W celu uruchomienia polecenia DML lub DDL należy na rzecz obiektu Command wywołać metodę ExecuteNonQuery(). Metoda ta dla poleceń DML zwraca liczbę wierszy, których dotyczyło polecenie, a dla innych poleceń zawsze wartość –1.

Na slajdzie pokazano przykład wykorzystania obiektu SqlCommand do zmodyfikowania wiersza w bazie danych SQL Server. Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Utworzenie obiektu połączenia z bazą danych.

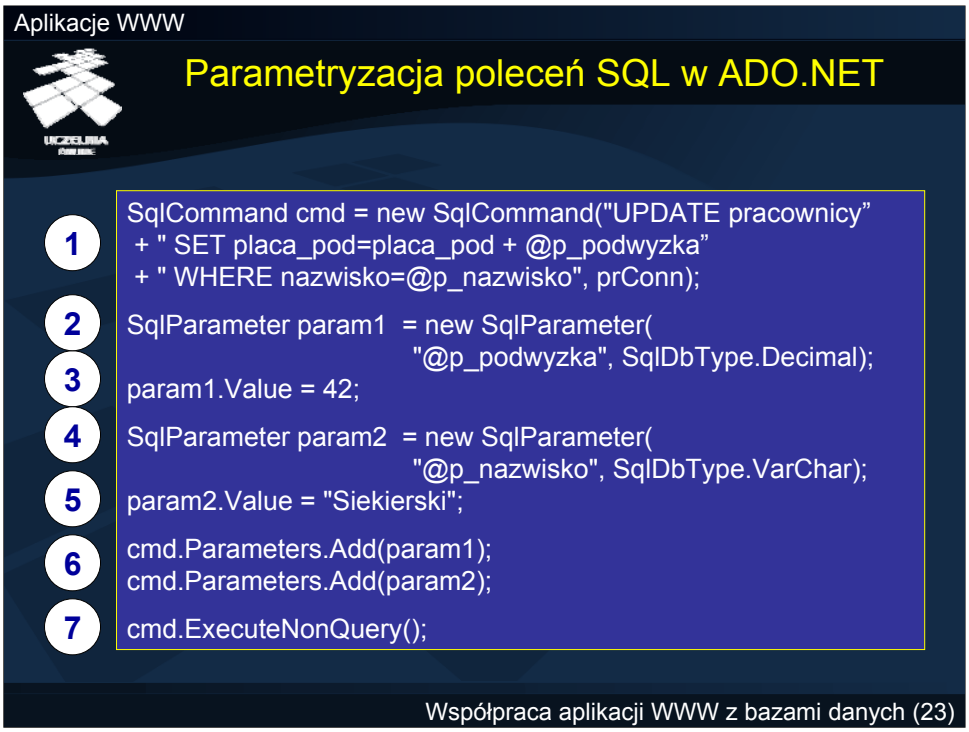
2. Otwarcie połączenia z bazą danych.

3. Utworzenie obiektu SqlCommand w ramach otwartego połączenia.

4. Związanie z obiektem SqlCommand treści polecenia SQL modyfikującego wiersz w tabeli Pracownicy.

5. Uruchomienie polecenia SQL poprzez wywołanie metody ExecuteNonQuery() na rzecz obiektu SqlCommand i zapamiętanie zwróconej liczby zmodyfikowanych wierszy w zmiennej. Następnie wyświetlenie na konsoli komunikatu o liczbie zmodyfikowanych wierszy.

6. Zamknięcie połączenia z bazą danych.

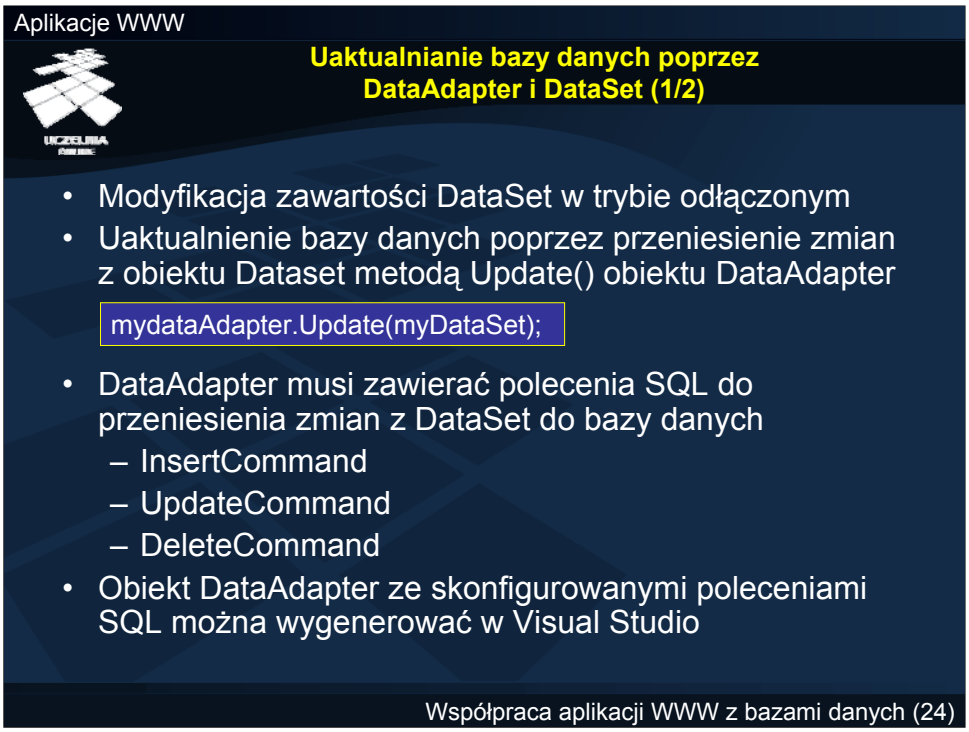


Bezpiecznym sposobem parametryzacji poleceń SQL, chroniącym przed techniką ataku SQL injection, jest w ADO.NET wykorzystanie obiektów Parameter związanych z obiektem Command.

Na slajdzie pokazano przykład wykonania sparametryzowanego polecenia UPDATE podwyższającego pensję wskazanemu pracownikowi. Parametry to kwota podwyżki i nazwisko pracownika. Kolejne kroki przykładowego fragmentu kodu to:

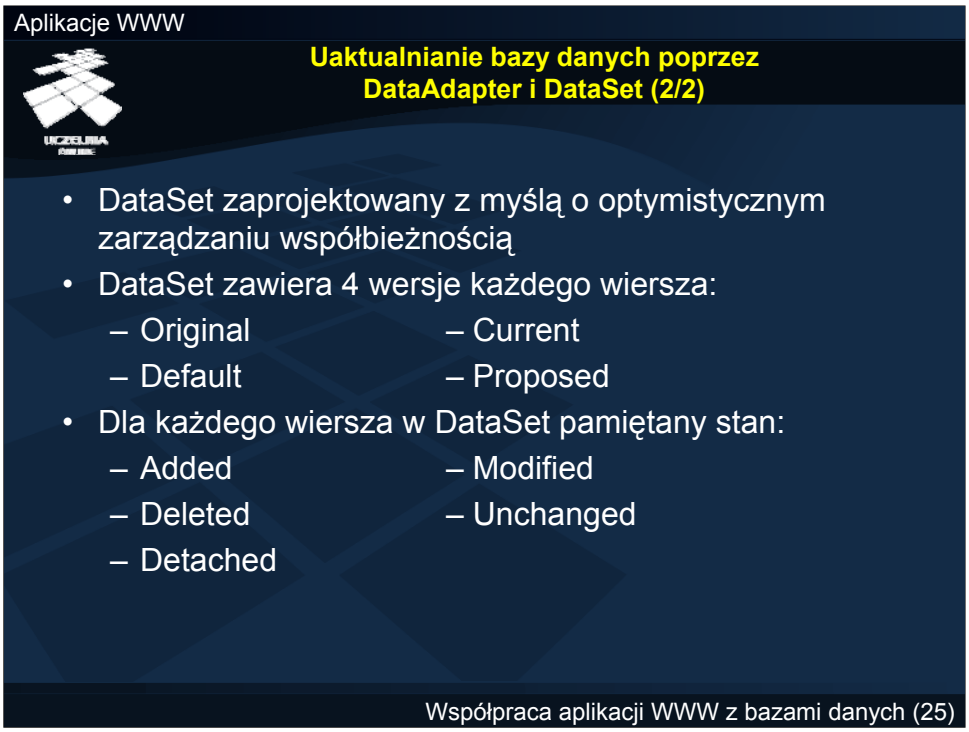
1. Utworzenie obiektu SqlCommand w kontekście wskazanego połączenia. Parametry w treści polecenia SQL zagnieżdżone są za pomocą notacji „@nazwa_parametru”.
2. Utworzenie obiektu SqlParameter dla parametru @p_podwyzka typu Decimal.
3. Przypisanie wartości parametrowi @p_podwyzka.
4. Utworzenie obiektu SqlParameter dla parametru @p_nazwisko typu VarChar.
5. Przypisanie wartości parametrowi @p_nazwisko.
6. Związanie obiektów parametrów z obiektem polecenia.
7. Wykonanie polecenia SQL.

Do wskazania typu danych dla parametru służą wartości typu wyliczeniowego SqlDbType dla systemu SQLServer, a DbType dla pozostałych. Jeśli parametr ma przyjąć wartość NULL, należy mu przypisać DBNull. Przed wykonaniem sparametryzowanego polecenia, należy związać z poleceniem obiekty Parameter dla wszystkich parametrów.



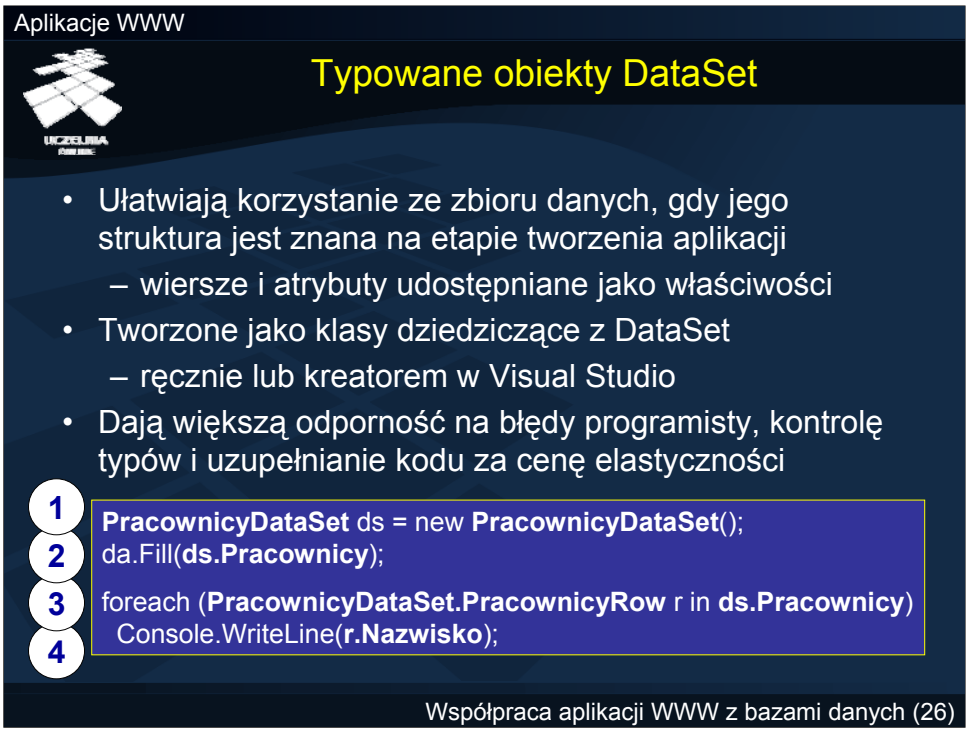
Aplikacja uaktualnia zawartość obiektu DataSet modyfikując, wstawiając i usuwając wiersze w trybie odłączonym – bez utrzymywania otwartego połączenia z bazą danych. Uaktualnienie bazy danych poprzez przeniesienie zmian z obiektu DataSet jest realizowane metodą Update() obiektu DataAdapter.

Obiekt DataAdapter musi zawierać polecenia SQL (w formie obiektów Command i związanych z nimi obiektów Parameter) do przeniesienia zmian z DataSet do bazy danych. Polecenia te są zapamiętane we właściwościach InsertCommand, UpdateCommand i DeleteCommand obiektu DataAdapter. Obiekt DataAdapter ze skonfigurowanymi poleceniami SQL dla operacji SELECT, INSERT, UPDATE, DELETE można wygenerować kreatorem w Visual Studio. Przypomnijmy, że obiekt Command zawierający polecenie SELECT jest zapamiętany we właściwości SelectCommand obiektu DataAdapter i jest wykorzystywany przez metodę Fill(), wypełniającą obiekt DataSet danymi.



Obiekt DataSet, z racji przeznaczenia do pracy w trybie odłączonym, został zaprojektowany z myślą o optymistycznym zarządzaniu współbieżnością. Przypomnijmy, że pesymistyczna strategia zarządzania współbieżnością wiąże się z blokowaniem danych i wymaga utrzymywania połączenia z bazą danych, więc jest nieodpowiednia dla pracy z obiektami DataSet. Strategia optymistyczna dopuszcza ryzyko modyfikacji danych przez innego użytkownika w okresie między wypełnieniem obiektu DataSet danymi metodą Fill(), a uaktualnianiem bazy danych metodą Update() obiektu DataAdapter. Problem ten można minimalizować poprzez odświeżanie zawartości DataSet przez ponowne wywołanie Fill() lub implementację zarządzania współbieżnością w oparciu o etykiety czasowe dodane do wierszy tabel bazy danych.

Metoda Update() przenosząca zmiany w obiekcie DataSet musi wiedzieć, które wiersze zostały zmodyfikowane, usunięte i dodane. W tym celu DataSet zawiera cztery wersje każdego wiersza. Oprócz aktualnej (Current) pamiętana jest odczytana z bazy (Original), a ponadto domyślna (Default) i będąca przedmiotem edycji (Proposed). Ponadto, dla każdego wiersza w DataSet pamiętany jest jego stan, który może przyjąć jedną z pięciu wartości: Added – dodany; Deleted – usunięty; Detached – utworzony, ale niedodany do kolekcji wierszy lub usunięty, ale jeszcze niezniszczony; Modified – zmodyfikowany; Unchanged – niezmodyfikowany.



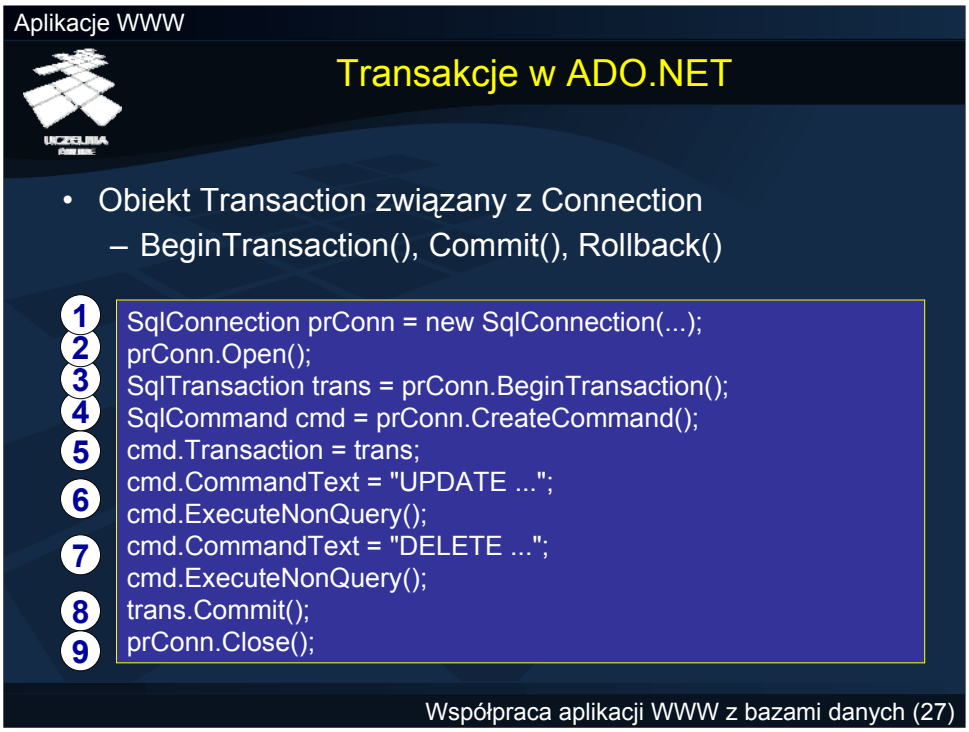
Typowane obiekty DataSet (ang. typed DataSet objects) ułatwiają korzystanie ze zbioru danych w kodzie aplikacji, gdy jego struktura jest znana na etapie tworzenia aplikacji. Należy podkreślić, że jest to scenariusz typowy, gdyż aplikacje zazwyczaj wykonują powtarzalne operacje na określonych tabelach bazy danych. Ułatwienie polega na tym, że typowane obiekty DataSet udostępniają wiersze i atrybuty jako właściwości, a nie elementy indeksowanych kolekcji.

Typowane obiekty DataSet są obiektami klas dziedziczących z DataSet, w pełni zachowują więc oryginalną funkcjonalność. Klasy takie można tworzyć ręcznie lub korzystając z kreatora w Visual Studio.

Zalety typowanych obiektów DataSet to większa odporność na błędy programisty np. literówki w nazwach atrybutów, kontrola typów danych i automatyczne uzupełnianie kodu w odwołaniach do tabel i atrybutów w środowisku Visual Studio. Ceną za uzyskane korzyści jest utrata elastyczności oferowanej przez zwykłe obiekty DataSet, ale ta elastyczność, polegająca na dynamicznym podawaniu nazw tabel i ich atrybutów, w typowych aplikacjach nie jest potrzebna.

U dołu slajdu przedstawiony jest fragment kodu ilustrujący korzystanie z typowanych obiektów DataSet. Klasa typowanego obiektu DataSet do przechowywania danych o pracownikach została nazwana PracownicyDataSet (1).

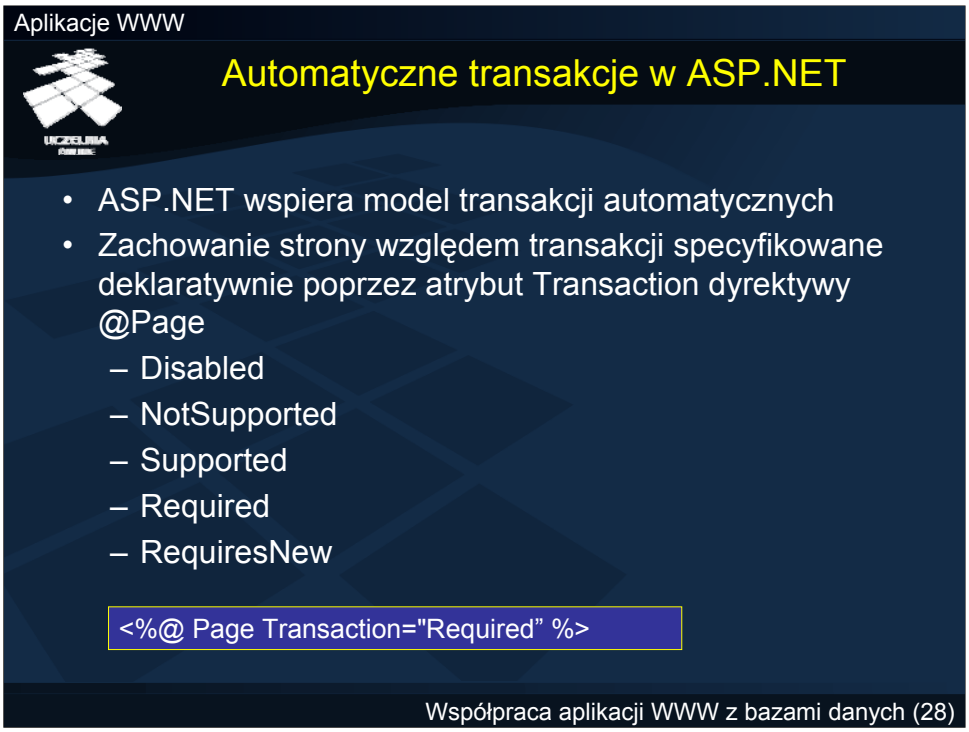
Tabela z danymi pracowników jest dostępna w obiekcie PracownicyDataSet jako właściwość Pracownicy (2). Każdy wiersz tej tabeli jest obiektem klasy zagnieżdżonej PracownicyDataSet.PracownicyRow (3). Klasa ta udostępnia atrybuty wiersza tabeli jako właściwości np. Nazwisko (4).



Do realizacji transakcji w ADO.NET służy obiekt Transaction związany z obiektem Connection reprezentującym połączenie. Obiekt Transaction udostępnia metody: BeginTransaction() do rozpoczęcia transakcji, Commit() do zatwierdzenia transakcji i Rollback() do wycofania transakcji.

Na slajdzie pokazano przykład transakcji obejmującej dwa polecenia SQL (treść poleceń dla oszczędności miejsca nie została w całości pokazana na slajdzie). Znaczenie poszczególnych instrukcji w kodzie jest następujące:

1. Utworzenie obiektu połączenia z bazą danych.
2. Otwarcie połączenia z bazą danych.
3. Utworzenie obiektu transakcji w ramach połączenia z bazą danych.
4. Utworzenie obiektu SqlCommand w ramach otwartego połączenia.
5. Związanie obiektu SqlCommand z transakcją.
6. Związanie z obiektem SqlCommand treści pierwszego polecenia SQL, które ma być wykonane w transakcji, a następnie wykonanie go.
7. Związanie z obiektem SqlCommand treści drugiego polecenia SQL, które ma być wykonane w transakcji, a następnie wykonanie go.
8. Zatwierdzenie transakcji.
9. Zamknięcie połączenia z bazą danych.



ASP.NET wspiera transakcje automatyczne w ramach modelu transakcji rozproszonych Microsoft dla Microsoft Transaction Server, COM+ i wspólnego środowiska uruchomieniowego .NET. W modelu tym komponent oznaczony jako biorący udział w transakcji będzie automatycznie uruchomiony w kontekście transakcyjnym.

W ASP.NET zachowanie strony względem transakcji specyfikowane jest deklaratywnie poprzez atrybut Transaction dyrektywy @Page. Atrybut ten może przyjąć jedną z następujących wartości:

Disabled – ASP.NET zignoruje kontekst transakcyjny (wartość domyślna);

NotSupported – strona będzie uruchomiona poza kontekstem transakcyjnym, niezależnie od tego czy jest w danej chwili aktywna transakcja czy nie;

Supported – strona będzie uruchomiona w kontekście bieżącej transakcji jeśli istnieje aktywna transakcja lub bez transakcji w przeciwnym wypadku;

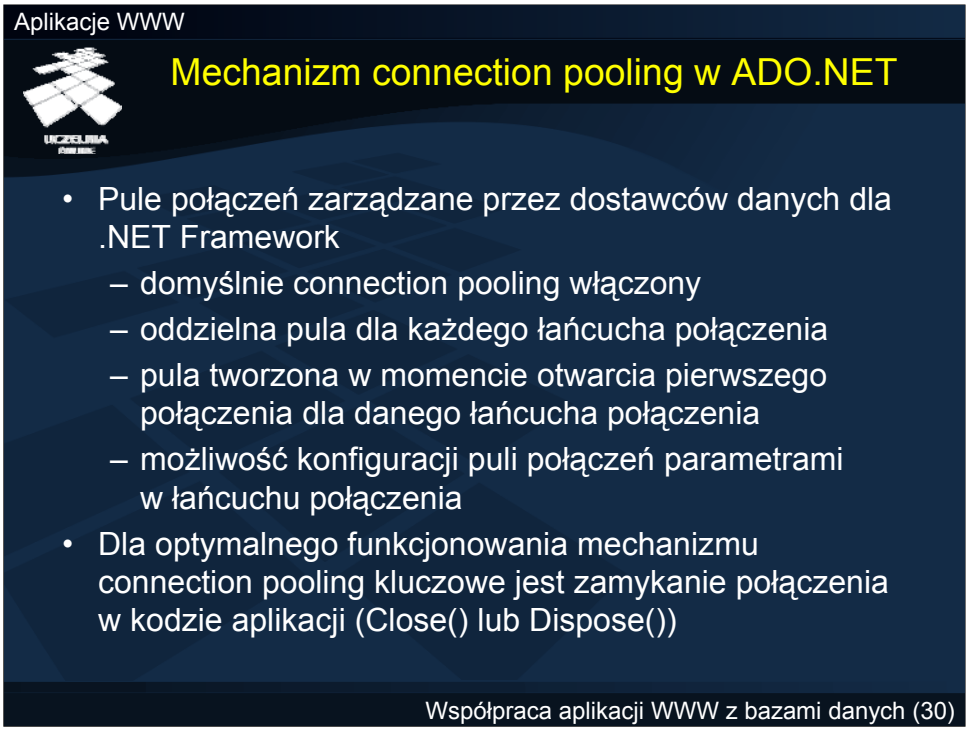
Required - strona będzie uruchomiona w kontekście bieżącej transakcji jeśli istnieje aktywna transakcja lub rozpocznie nową transakcję w przeciwnym wypadku;

RequiresNew – strona rozpocznie nową transakcję dla każdego żądania, niezależnie od tego czy istnieje aktywna transakcja czy nie.

Od wersji 2.0 ASP.NET oferuje kontrolki redukujące ilość kodu obsługującego komunikację z bazą danych. Kontrolki te są zalecane dla prostych aplikacji i umożliwiają deklaratywną specyfikację połączeń z bazą danych, poleceń SQL odczytujących i uaktualniających dane oraz prezentację danych na stronach ASP.NET.

Pierwszą podgrupę kontrolek dostępu do danych stanowią kontrolki źródeł danych, których przykładem jest SqlDataSource reprezentująca połączenie z bazą danych obsługiwaną przez jednego z dostawców danych dla .NET Framework. Drugą grupę stanowią kontrolki do prezentacji danych potrafiące współpracować z kontrolkami źródeł danych np. GridView.

Slajd pokazuje przykładowy fragment strony generujący tabelkę z danymi pracowników pobranymi z bazy danych. Dane prezentuje kontrolka GridView, która jako źródło danych wskazuje źródło o nazwie „ds”. Źródło to definiuje kontrolka SqlDataSource. W jego definicji podana jest treść polecenia odczytującego dane (jako atrybut SelectCommand) i referencja do łańcucha połączenia zdefiniowanego w pliku konfiguracyjnym Web.config. Przykładowy fragment pliku Web.config został pokazany u dołu slajdu. Łańcuchy połączeń są definiowane w sekcji <connectionStrings>. Dla każdego łańcucha oprócz jego nazwy i treści podany jest wykorzystywany dostawca danych (w przykładzie dostawca danych dla SQL Server).

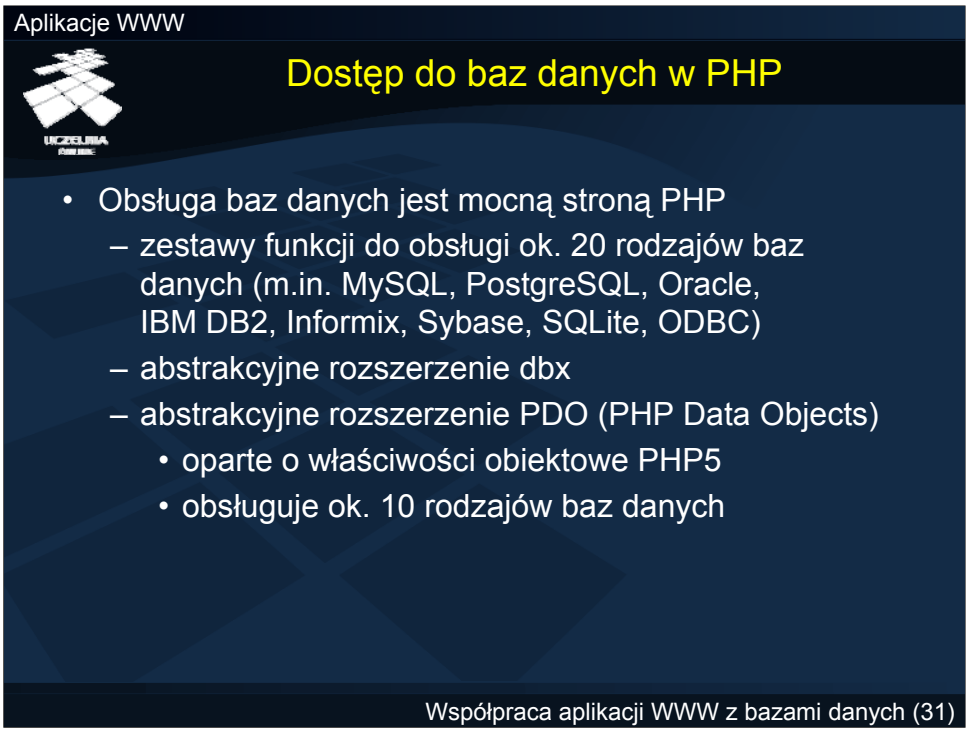


Mechanizm connection pooling pozwala poprawić efektywność aplikacji korzystających z baz danych poprzez utrzymywanie zbioru otwartych połączeń w puli i udostępnianie ich aplikacji gdy wymaga ona otwarcia połączenia z bazą.

Na platformie .NET pule połączeń są zarządzane przez dostawców danych dla .NET Framework. Domyślnie connection pooling jest włączony z sensownie ustawionymi parametrami i nie wymaga konfiguracji.

Dla każdego łańcucha połączenia utrzymywana jest oddzielna pula połączeń. Nowa pula połączeń jest tworzona automatycznie w momencie otwarcia pierwszego połączenia dla danego łańcucha połączenia. Istnieje możliwość konfiguracji puli połączeń poprzez zawarcie w łańcuchu połączenia parametrów określających minimalną i maksymalną liczbę połączeń dla puli. Dostępny jest również parametr łańcucha połączenia wyłączający mechanizm connection pooling dla danego łańcucha połączenia.

Dla optymalnego funkcjonowania mechanizmu connection pooling kluczowe jest zamykanie połączenia w kodzie aplikacji (metodą Close() lub Dispose()), a nie poleganie na mechanizmie garbage collection. Ma to na celu jak najszybsze zwrócenie nieużywanego już połączenia do puli.



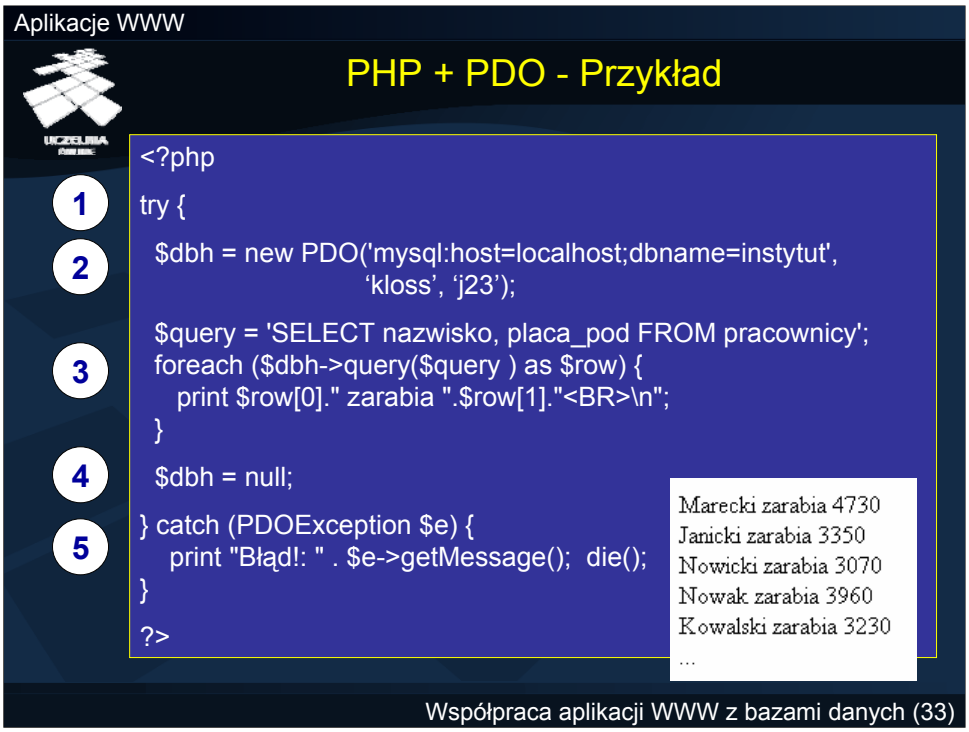
Obsługa baz danych jest uważana za mocną stronę PHP. Jednocześnie tworzenie bazodanowych aplikacji w PHP nie jest skomplikowane. PHP udostępnia dedykowane zestawy funkcji do obsługi około 20 rodzajów baz danych, w tym MySQL, PostgreSQL, Oracle, IBM DB2, Informix, Sybase, SQLite oraz zestaw funkcji do komunikacji z bazami danych poprzez interfejs ODBC. Zestawy funkcji dla poszczególnych rodzajów baz danych są dostępne w PHP, podobnie jak wiele innych specjalistycznych zestawów funkcji, w formie tzw. rozszerzeń (ang. extensions). Oprócz rozszerzeń dedykowanych dla poszczególnych systemów, PHP oferuje dwa abstrakcyjne rozszerzenia stanowiące warstwę abstrakcji, oferującą spójny interfejs do różnych rodzajów baz danych. Te abstrakcyjne rozszerzenia to: starsze dbx i nowsze PDO (PHP Data Objects). Oba współpracują z kilkoma rodzajami baz danych.

PDO wymaga nowych własności obiektowych w PHP5 i nie jest dostępne dla wcześniejszych wersji PHP niż 5.0. PDO obsługuje poszczególne bazy danych za pośrednictwem sterowników PDO. Obecnie dostępne są sterowniki PDO dla około 10 rodzajów baz danych, w tym MySQL, PostgreSQL, Oracle, Microsoft SQL Server, Informix, Sybase, SQLite oraz ODBC.

Na slajdzie zamieszczono kod źródłowy przykładowego dokumentu PHP, wyświetlającego listę pracowników odczytaną z bazy danych MySQL zestawem dedykowanych funkcji dla MySQL. Tradycyjnie, MySQL był zawsze powszechnie wykorzystywany w połączeniu z PHP. W PHP4 obsługa MySQL jest zawsze włączona, bez konieczności instalowania odpowiedniego rozszerzenia jak dla innych rodzajów baz danych.

Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Otwarcie połączenia z lokalnym serwerem MySQL, nasłuchującym na domyślnym dla MySQL porcie, jako użytkownik „kloss” z hasłem „j23”. W przypadku niepowodzenia otwarcia połączenia, skrypt kończy działanie z komunikatem o błędzie.
2. Wybór bazy danych „instytut” spośród baz dostępnych na serwerze MySQL.
3. Wysłanie zapytania do bazy danych i zapamiętanie uchwytu do jego wyniku w zmiennej $result.
4. Odczyt w pętli kolejnych wierszy wyniku zapytania do tablicy.
5. Wyświetlenie danych bieżącego wiersza wyniku zapytania – wersja z dostępem do tablicy przez indeks liczbowy, odpowiadający pozycji wyrażenia w klauzuli SELECT zapytania.
6. Zwolnienie zasobów na serwerze związanych z wynikiem zapytania.
7. Zamknięcie połączenia z serwerem MySQL.



Na slajdzie zamieszczono kod źródłowy przykładowego dokumentu PHP, wyświetlającego listę pracowników odczytaną z bazy danych MySQL poprzez warstwę abstrakcji PDO. Znaczenie wyróżnionych fragmentów kodu jest następujące:

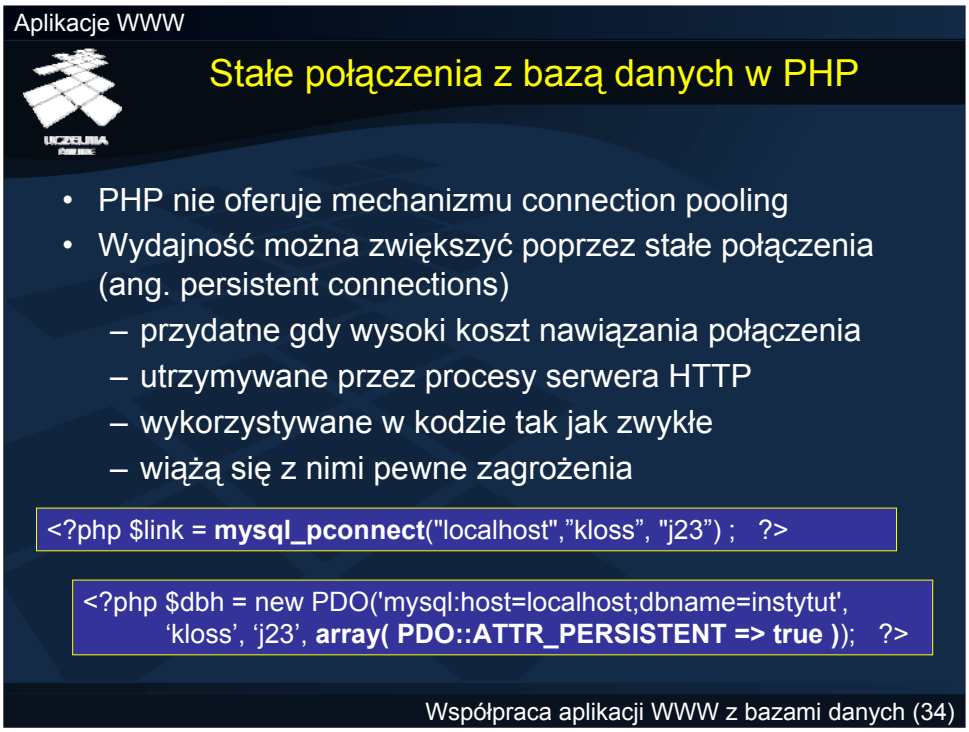
1. Operacje na bazie danych, jako mogące rzucić wyjątek, zawarte są w instrukcji try/catch.

2. Utworzenie połączenia z lokalnym serwerem MySQL, jako użytkownik „kloss” z hasłem „j23”, z wybraniem bazy danych „instytut”.

3. Wysłanie zapytania do bazy danych i przetworzenie zwróconego przez nie zbioru wynikowego w pętli foreach.

4. Zamknięcie połączenia z bazą danych poprzez przypisanie null do jedynej referencji na obiekt reprezentujący połączenie.

5. Obsługa wyjątku PDOException rzucanego przez funkcje PDO. W wypadku przechwycenia wyjątku, wyświetlany jest pobrany z niego komunikat o błędzie, a następnie skrypt kończy działanie.

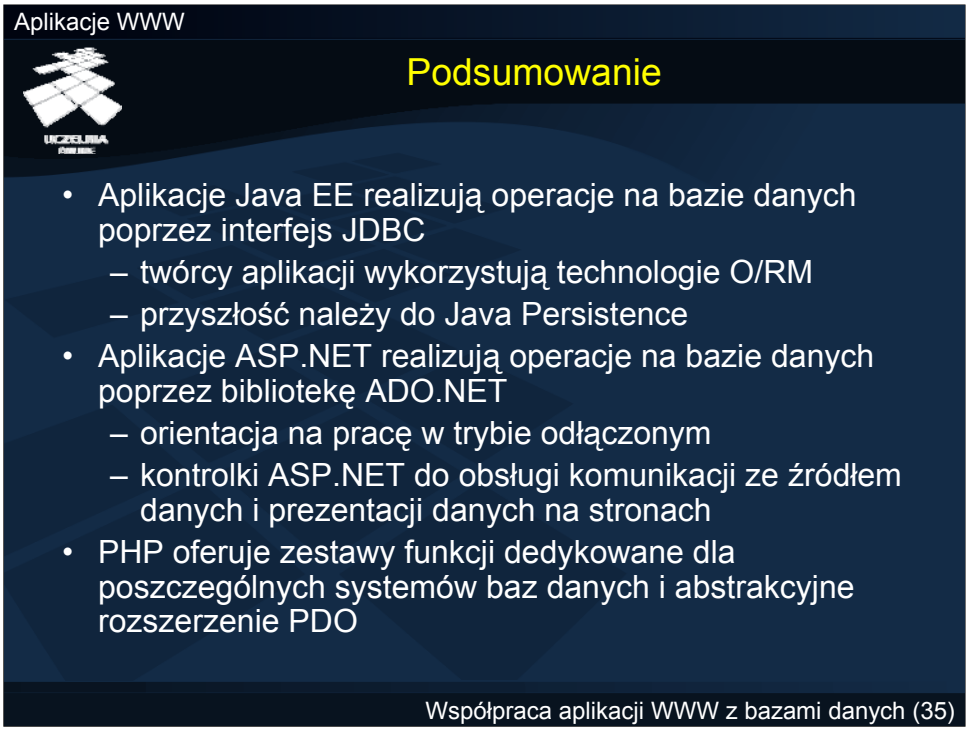


PHP nie oferuje mechanizmu pul połączeń (connection pooling). Wydajność komunikacji z bazą danych poprzez redukcję czasu nawiązywania połączenia można w aplikacjach PHP zwiększyć stosując tzw. stałe połączenia (ang. persistent connections). Rozwiązanie to jest przydatne gdy koszt nawiązania połączenia jest wysoki, co zależy np. od wykorzystywanego systemu zarządzania bazą danych.

Mechanizm stałych połączeń polega na utrzymywaniu połączeń przez procesy serwera HTTP obsługujące żądania. Gdy skrypt próbuje nawiązać stałe połączenie, PHP sprawdza czy proces uruchamiający skrypt posiada już otwarte wcześniej identyczne połączenie (z tym samym serwerem, nazwą użytkownika i hasłem). Jeśli tak – zostanie ono wykorzystane ponownie, jeśli nie - utworzone będzie nowe.

Stałe połączenia pozwalają najczęściej znacząco poprawić wydajność aplikacji. Z ich wykorzystaniem wiążą się jednak pewne zagrożenia. Błędy w oprogramowaniu blokowania danych mogą zablokować dane dla kolejnych wywołań skryptu. Z kolei źle dobrana liczba procesów usługowych serwera może doprowadzić do wyczerpania limitu połączeń z serwerem bazy danych.

Stałe połączenia od momentu ich otwarcia są wykorzystywane w kodzie tak jak zwykłe. Modyfikacja skryptu korzystającego ze zwykłych połączeń tak by wykorzystywał stałe sprowadza się zmiany nazwy funkcji lub dodania parametru wywołania. Na slajdzie pokazano przykłady otwarcia stałego połączenia przy korzystaniu z rozszerzenia dla MySQL (funkcją mysql_pconnect() zamiast mysql_connect()) i uniwersalnego rozszerzenia PDO (poprzez dodanie parametru PDO::ATTR_PERSISTENT w wywołaniu konstruktora połączenia).



Aplikacje Java EE realizują operacje na bazie danych poprzez interfejs JDBC. Ponieważ ręczne kodowanie w JDBC jest uciążliwe twórcy aplikacji Java obecnie najczęściej wykorzystują technologie odwzorowania obiektowo-relacyjnego (O/RM). W najbliższej przyszłości należy się spodziewać migracji ze specyficznych API poszczególnych technologii O/RM do standardowego Java Persistence API.

Aplikacje ASP.NET realizują operacje na bazie danych poprzez bibliotekę ADO.NET. Charakterystyczna dla ADO.NET jest orientacja na pracę z danymi w trybie odłączonym, bez konieczności utrzymywania połączenia z bazą danych, co ułatwia przetwarzanie danych w komponentowych aplikacjach internetowych. Ważną rolę w ASP.NET pełnią kontrolki do obsługi komunikacji ze źródłem danych i prezentacji danych na stronach, pozwalające zredukować ilość kodu obsługującego komunikację z bazą danych w aplikacji.

PHP oferuje zestawy funkcji dedykowane dla poszczególnych systemów baz danych i rozwijane od niedawna abstrakcyjne rozszerzenie PDO oferujące jednolity interfejs do wielu systemów.

Materiały dodatkowe

- JDBC API, http://java.sun.com/javase/technologies/database.jsp
- Hibernate, http://www.hibernate.org/
- The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/
- .NET Framework Developer Center, http://msdn.microsoft.com/netframework/
- ASP.NET Quickstart Tutorial, http://www.asp.net/QuickStart/aspnet/
- PHP: Documentation, http://www.php.net/docs.php

•JDBC API, http://java.sun.com/javase/technologies/database.jsp
•Hibernate, http://www.hibernate.org/
•The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/
•.NET Framework Developer Center, http://msdn.microsoft.com/netframework/
•ASP.NET Quickstart Tutorial, http://www.asp.net/QuickStart/aspnet/
•PHP: Documentation, http://www.php.net/docs.php