Perl

Autor: Reuven M. Lerner
Tłumaczenie: M. Michalski, S. Dzieniszewski
ISBN: 83-7361-169-X
Tytuł oryginału: Core Perl
Format: B5, stron: 45
Przykłady na ftp: 61 kB

Ten kompletny przewodnik po Perlu szybko i wygodnie dostarcza doświadczonym programistom odpowiednich technik i ilustrujących je praktycznych przykładów kodu. Lektura tej książki pozwoli Ci najpierw rozwinąć swoje dotychczasowe umiejętności, a następnie zgłębić tajniki podstawowych technik programowania w Perlu. Książka rozpoczyna się opisem podstawowej składni języka, przechodzi później do obiektów, struktur danych i zasad przetwarzania tekstu. Następnie wyjaśnia, jak korzystać z dostarczonych przez Perl narzędzi umożliwiających pracę z plikami, działanie programów w sieci czy współpracę z relacyjnymi bazami danych. Na koniec pokazuje, jak wykorzystać pełnię możliwości Perla podczas tworzenia aplikacji WWW — zarówno prostych skryptów CGI, jak i w pełni zaawansowanych narzędzi obsługujących witryny WWW.

Opisano między innymi:

- Typy danych i podstawowe funkcje Perla
- Zasady pisania i korzystania z modułów Perla
- Sposoby korzystania z plików tekstowych i plików DBM
- Interfejs DBI pozwalający na korzystanie z baz danych i integrowanie baz danych ze stronami WWW
- Programy CGI, moduły mod_perl dla serwera Apache, cookie protokołu HTTP oraz szablony HTML/Perla
- Mason: oparty na Perlu system tworzenia zaawansowanych witryn WWW
- Wykrywanie i usuwanie błędów, optymalizacja kodu oraz sprawy związane z bezpieczeństwem

Od wielu już lat programiści na całym świecie doceniają Perla za jego prostotę, wygodę i uniwersalną zdolność do rozwiązywania szerokiego zakresu problemów; począwszy od przetwarzania tekstu i administrowania systemem operacyjnym po komunikację z bazami danych i tworzenie stron WWW. Książka „Perl" dostarcza programistom wiedzy niezbędnej do tworzenia wszechstronnych, przejrzystych i bardzo wydajnych programów — niezależnie jakie będą zadania tworzonych przez niego aplikacji.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

12

Tworzenie aplikacji dla baz danych

W tym rozdziale:

- Projektowanie bazy danych
- Tworzenie aplikacji dla baz danych
- Wykrywanie błędów w programach `DBI`

Poprzedni rozdział był wprowadzeniem do zagadnień związanych z bazami danych, a w szczególności do języka SQL i modułu `DBI`. W tym rozdziale wykorzystamy te narzędzia do przygotowania zestawu aplikacji obsługujących bazę danych.

Po drodze przyjrzymy się projektowaniu bazy danych, co — podobnie jak w przypadku projektowania programu — jest kluczową, ale często niedocenianą częścią procesu programowania. W szczególności omówimy proces *normalizacji*, który poprawia efektywność bazy danych i czyni ją bardziej elastyczną, redukując jednocześnie możliwość pojawienia się błędów.

Aplikacje, które stworzymy, będą wykorzystywać zarówno techniki opisane w poprzednim rozdziale, jak i nowe typy danych SQL oraz metody interfejsu `DBI`. Przyjrzymy się bliżej metodzie `trace`, która ułatwia wykrywanie błędów w zapytaniach SQL.

Rozdział ten dostarcza podstawowej wiedzy na temat procesu pisania programów współdziałających z bazą danych, włączając w to projektowanie tabel i wykrywanie błędów w przygotowanych programach.

12.1. Projektowanie bazy danych

Nasze zadanie polega na skomputeryzowaniu katalogu książek sprzedawanych w pewnej księgarni. Zbudujemy bazę danych księgarni w systemie PostgreSQL i napiszemy w Perlu aplikacje, które pozwolą nam na obsługę danych zawartych w bazie.

Pierwszym krokiem podczas tworzenia aplikacji bazy danych jest zawsze stworzenie tabel. Opiszę tutaj proces tworzenia prostego projektu bazy danych, omawiając pokrótce problemy pojawiające się w większości aplikacji baz danych.

12.1.1. Tabela Books

Ponieważ księgarni potrzebny jest katalog książek oferowanych do sprzedaży, trzeba zacząć od przygotowania tabeli `Books`, której wiersze będą przechowywać dane o poszczególnych książkach (identyfikator książki, kod ISBN, tytuł, autora, wydawcę, liczbę stron, datę wydania oraz cenę książki):

```
CREATE TABLE Books
(
   book_id         SERIAL       NOT NULL,
   isbn            TEXT         NOT NULL,
   title           TEXT         NOT NULL,
   author          TEXT         NOT NULL,
   publisher       TEXT         NOT NULL,
   num_pages       NUMERIC(5,0) NOT NULL,
   pub_date        DATE         NOT NULL,
   us_dollar_price NUMERIC(6,2) NOT NULL,

   PRIMARY KEY(book_id),
   UNIQUE(isbn)
)
;
```

Nasza tabela `Books` wygląda podobnie jak tabele, które prezentowałem w rozdziale 11., niemniej pojawia się parę typów danych oraz ograniczeń, których jeszcze nie omawialiśmy:

- kolumna daty wydania `pub_date` jest typu `DATE` (data). Jest to typ bardzo podobny do typu `TIMESTAMP` (znacznik czasu), ale zawiera samą datę, bez informacji o godzinie;
- kolumna ceny `us_dollar_price` została zdefiniowana jako kolumna typu `NUMERIC(6,2)`, co oznacza, że może zawierać sześciocyfrowe liczby posiadające dwie cyfry po przecinku. Tak więc możemy sprzedawać książki o cenie do 9999,99 dolarów;
- kolumna `isbn` przechowuje niepowtarzalny i niezmienny kod ISBN (ang. *International Standard Book Number*), który mógłby pełnić funkcję klucza głównego. Niemniej kody ISBN są dość długie, a zazwyczaj dobrze jest, aby klucz główny był tak krótki, jak to tylko możliwe. Aby upewnić się, że wartości w kolumnie `isbn` będą niepowtarzalne (choć nie są one kluczem głównym), możemy dodać do definicji tabeli ograniczenie `UNIQUE`, które automatycznie utworzy indeks dla podanej kolumny.

Z technicznego punktu widzenia zaprezentowana tutaj definicja tabeli `Books` jest w pełni poprawna. Niemniej w momencie, gdy zaczniemy korzystać z tej tabeli, ujawnią się pewne istotne problemy.

Załóżmy, że księgarnia chciałaby przygotować katalog książek wydawnictwa Prentice-Hall. Przedstawione tu zapytanie powinno pobrać nazwy wszystkich książek wydanych przez Prentice-Hall:

```
SELECT title
FROM Books
WHERE publisher = 'Prentice-Hall'
ORDER BY title
;
```

Co się jednak stanie, jeśli jedna z książek będzie miała w kolumnie `publisher` wydawcę podanego jako `Prentice Hall` zamiast `Prentice-Hall`? Taka książka oczywiście nie będzie spełniać wymogów stawianych przez nasze zapytanie i w związku z tym nie trafi do katalogu.

Podobne problemy pojawiają się również podczas wyszukiwania książek napisanych przez określonego autora. Jeśli osoba wpisująca książkę do bazy przez pomyłkę źle wpisze nazwisko autora, to książka ta nie pojawi się w odpowiedzi na zapytanie `SELECT` pobierające autora (`author`) o określonym nazwisku.

12.3.2. Normalizacja tabeli Books

Rozwiązaniem tego problemu jest *normalizacja* danych, która pozwala upewnić się, że dane wpisywane będą do bazy w jeden tylko sposób. Znormalizowane bazy danych cechuje znacznie mniejsze ryzyko pojawienia się uszkodzonych lub niezsynchronizowanych danych. Bazy takie są również znacznie wydajniejsze niż bazy, które nie zostały znormalizowane.

Aby znormalizować tabelę `Books`, przygotujemy dwie nowe tabele: `Publishers` (wydawcy) i `Authors` (autorzy). Tabele te będą pozwalały na wpisanie każdego autora i każdego wydawcy do bazy danych tylko raz. Dodatkowo wykorzystamy mechanizm kluczy obcych wskazujących na te wartości (patrz: sekcja 11.3.6):

```
CREATE TABLE Authors
(
   author_id    SERIAL   NOT NULL,
   author_name  TEXT     NOT NULL,

   PRIMARY KEY(author_id)
)
;

CREATE TABLE Publishers
(
   publisher_id   SERIAL   NOT NULL,
   publisher_name TEXT     NOT NULL,

   PRIMARY KEY(publisher_id),
   UNIQUE(publisher_name)
)
;

CREATE TABLE Books
(
   book_id         SERIAL       NOT NULL,
```

```
    isbn            TEXT         NOT NULL,
    title           TEXT         NOT NULL,
    author_id       INTEGER      NOT NULL  REFERENCES Authors,
    publisher_id    INTEGER      NOT NULL  REFERENCES Publishers,
    num_pages       NUMERIC(5,0) NOT NULL,
    pub_date        DATE         NOT NULL,
    us_dollar_price NUMERIC(6,2) NOT NULL,

    PRIMARY KEY(book_id),
    UNIQUE(isbn)
)
;
```

Warto zauważyć, że tabele te nakładają ograniczenie unikatowości na nazwę wydawnictwa, ale nie nakładają go na nazwiska autorów. Może się w końcu zdarzyć kilku autorów o nazwisku John Smith, ale raczej nie powinno istnieć więcej niż jedno wydawnictwo o nazwie Prentice Hall.

Znormalizowane tabele gwarantują, że nazwiska autorów i nazwy wydawnictw będą występowały w tej samej formie w całej bazie danych. Co więcej, zmiana nazwy wydawnictwa wymagać będzie teraz aktualizacji tylko jednego wiersza w tabeli `Publishers` zamiast wielu wierszy w tabeli `Books`. Jakby tych korzyści było jeszcze mało, tabela `Books` zmniejszy znacznie swoje rozmiary dzięki zastąpieniu dwóch kolumn typu `TEXT` kolumnami typu `INTEGER`. Znormalizowane tabele zajmować będą mniej miejsca na dysku i w pamięci, co przyspieszy działanie bazy danych.

Obecnie nasze zapytanie pobierające książki wydawnictwa Prentice-Hall wymaga wykonania złączenia:

```
SELECT title
FROM Books B, Publishers P
WHERE P.publisher_name = 'Prentice-Hall'
  AND P.publisher_id = B.publisher_id
ORDER BY title
;
```

Złączenie razem trzech tablic pozwoli na zdobycie listy wszystkich książek napisanych przez określonego autora, takiego jak np. Paul Krugman:

```
SELECT B.isbn, B.title, P.publisher_name,
       B.pub_date, B.us_dollar_price
FROM Books B, Publishers P, Authors A
WHERE A.author_name = 'Paul Krugman'
  AND B.author_id = A.author_id
  AND B.publisher_id = P.publisher_id
ORDER BY title
;
```

12.1.3. Głębsza normalizacja

Przedstawiona przed chwilą wersja tabeli `Books` była niewątpliwie krokiem w dobrym kierunku. Co jednak, jeśli dana książka ma więcej niż jednego autora? Dotychczasowa definicja tabeli nie dopuszcza takiej sytuacji.

Jednym z możliwych rozwiązań tego problemu jest dodanie do tabeli Books kilku nowych kolumn. Można na przykład zmienić kolumnę author_id na author1_id i dodać trzy kolumny dopuszczające wartość NULL (author2_id, author3_id oraz author4_id) przeznaczone dla dodatkowych autorów.

To rozwiązanie jednak sztucznie ogranicza liczbę autorów, których może posiadać książka, i prowadzi do niepotrzebnego marnotrawienia przestrzeni na dysku w przypadku każdej książki, która będzie miała mniej autorów. Dodatkowo nadmiernie komplikuje wszystkie zapytania — rozważmy na przykład listę książek napisanych przez Paula Krugmana:

```
SELECT B.isbn, B.title, P.publisher_name,
       B.pub_date, B.us_dollar_price
FROM Books B, Publishers P, Authors A
WHERE A.author_name = 'Paul Krugman'
  AND (  B.author1_id = A.author_id
      OR B.author2_id = A.author_id
      OR B.author3_id = A.author_id
      OR B.author4_id = A.author_id )
  AND B.publisher_id = P.publisher_id
ORDER BY title
;
```

Lepszym rozwiązaniem jest zupełne usunięcie informacji o autorze z tabeli Books. Zamiast tego przygotujemy tabelę BookAuthors (autorzy książki) posiadającą dwie kolumny — jedną będzie klucz obcy do tabeli Books, a drugą klucz obcy do tabeli Authors. A oto poprawiona definicja tabeli Books wraz z tabelą BookAuthors:

```
CREATE TABLE Books
(
   book_id         SERIAL       NOT NULL,
   isbn            TEXT         NOT NULL,
   title           TEXT         NOT NULL,
   publisher_id    INTEGER      NOT NULL  REFERENCES Publishers,
   num_pages       NUMERIC(5,0) NOT NULL,
   pub_date        DATE         NOT NULL,
   us_dollar_price NUMERIC(6,2) NOT NULL,

   PRIMARY KEY(book_id),
   UNIQUE(isbn)
)
;

CREATE TABLE BookAuthors
(
   book_id   INTEGER  NOT NULL  REFERENCES Books,
   author_id INTEGER  NOT NULL  REFERENCES Authors,

   UNIQUE(book_id, author_id)
)
;
```

Zarówno identyfikator książki (book_id), jak i identyfikator autora (author_id) mogą pojawić się w tabeli wielokrotnie. Powinniśmy jednak zagwarantować, żeby każdy autor w odniesieniu do konkretnej książki pojawiał się tylko raz. Robimy to, stosując ograniczenie UNIQUE łącznie na dwóch kolumnach zamiast na pojedynczej kolumnie.

Po dodaniu tabeli BookAuthors zapytanie odszukujące wszystkie książki napisane przez Paula Krugmana wymagać będzie wykonania złączenia czterech tabel:

```
SELECT B.isbn, B.title, P.publisher_name,
       B.pub_date, B.us_dollar_price
FROM Books B, Publishers P, Authors A, BookAuthors BA
WHERE A.author_name = 'Paul Krugman'
  AND A.author_id = BA.author_id
  AND BA.book_id = B.book_id
  AND B.publisher_id = P.publisher_id
ORDER BY title
;
```

Z kolei następne zapytanie zwraca książkę o numerze ISBN 123-456-789, zwracając jeden wiersz dla każdego autora książki:

```
SELECT B.isbn, B.title, P.publisher_name,
       B.pub_date, B.us_dollar_price
FROM Books B, Publishers P, Authors A, BookAuthors BA
WHERE B.isbn = '123-456-789'
  AND B.book_id = BA.book_id
  AND BA.author_id = A.author_id
  AND B.publisher_id = P.publisher_id
ORDER BY title
;
```

Jak widać, prawidłowo zaprojektowana baza danych pozwala na uzyskanie odpowiedzi na wiele różnych pytań — nawet jeśli nie uwzględnialiśmy ich podczas projektowania bazy.

12.1.4. Indeksy

Jak pisałem w podrozdziale 11.6, należy dodawać indeks do każdej kolumny, która może pojawiać się w klauzuli WHERE. Klucze główne i kolumny oznaczone jako UNIQUE są indeksowane automatycznie, co oszczędza nam trochę pracy. W naszych tabelach jest jednak jeszcze kilka kolumn, na których prawdopodobnie również wykonywane będą zapytania.

```
CREATE INDEX author_name_index ON Authors (author_name);
CREATE INDEX publisher_name_index ON Publishers (publisher_name);

CREATE INDEX book_title_index ON Books (title);
CREATE INDEX book_publisher_id_index ON Books (publisher_id);
CREATE INDEX book_pubdate_index ON Books (pub_date);
CREATE INDEX book_us_dollar_price_index ON Books (us_dollar_price);

CREATE INDEX author_id_index ON BookAuthors (author_id);
```

Przygotowanie tych indeksów sprawi, że nawet jeśli księgarnia oferować będzie kilka milionów książek, łatwo i szybko je odnajdziemy, poszukując książek według tytułu, autora, daty publikacji czy ceny.

12.1.5. Integralność powiązań

Nasze obecne definicje tabel zabraniają wstawiania wartości NULL do kolumn tytułów książek i kodów ISBN. Może się jednak zdarzyć, że przez przypadek do tej lub innej kolumny wstawimy pusty łańcuch.

Ponieważ tworzona baza danych jest centralnym miejscem zbierania informacji o książkach dostępnych w księgarni, przechowywane w niej dane muszą być na tyle wiarygodne, na ile to tylko możliwe. Słowo kluczowe CHECK bazy PostgreSQL pozwala na nałożenie na kolumny dodatkowych ograniczeń, w wyniku których wprowadzane dane przechodzić będą dodatkowe testy. Na przykład tabelę Books można przedefiniować w następujący sposób:

```
CREATE TABLE Books
(
   book_id         SERIAL       NOT NULL,
   isbn            TEXT         NOT NULL  CHECK (isbn <> ''),
   title           TEXT         NOT NULL  CHECK (title ~* '[A-Z0-9]'),
   publisher_id    INTEGER      NOT NULL  REFERENCES Publishers,
   num_pages       NUMERIC(5,0) NOT NULL  CHECK (num_pages > 0),
   pub_date        DATE         NOT NULL,
   us_dollar_price NUMERIC(6,2) NOT NULL  CHECK (us_dollar_price > 0),

   PRIMARY KEY(book_id),
   UNIQUE(isbn)
)
;
```

Taka poprawiona definicja tabeli Books uniemożliwia wpisanie do kolumny isbn pustego łańcucha, gwarantuje, że kolumna title zawierać będzie przynajmniej jedną literę lub cyfrę (korzystając w tym celu z niewrażliwego na rozmiar liter operatora ~* PostgreSQL), i wymaga, żeby wartości w kolumnach num_pages i us_dollar_price były liczbami dodatnimi.

PostgreSQL będzie od tej pory odmawiać dodawania do bazy książek, które łamać będą jedno lub więcej z tak zdefiniowanych ograniczeń. Jeśli na przykład spróbujemy dodać książkę nieposiadającą tytułu:

```
INSERT INTO Books
   (isbn, title, publisher_id, num_pages, pub_date, us_dollar_price)
VALUES
   ('12345-6789', '', 1, 500, '2001-Dec-01', 50)
;
```

— PostgreSQL zwróci następujący komunikat o błędzie:

```
ERROR: ExecAppend: reject due to CHECK constraint books_title
```

Tego rodzaju ograniczenia mogą się wydawać nonsensowne lub zbyteczne, niemniej wcale tak nie jest. Bez nich moglibyśmy przypadkowo wprowadzić do bazy danych nieprawidłowe dane. Naprawianie bazy danych zawierającej niepoprawne lub niespójne dane jest skomplikowanym i pracochłonnym zadaniem, szczególnie jeśli baza danych musi być cały czas dostępna. Ograniczenie CHECK pozwala na uniknięcie takich sytuacji, wymuszając poprawność informacji zapisywanych w bazie danych.

12.1.6. Kategorie książek

Chcielibyśmy także przypisać każdą książkę do jednej lub więcej kategorii. Najprostszy sposób polega na przygotowaniu pary tabel podobnych do tych, które przygotowaliśmy dla autorów — tabeli `Categories` z nazwami kategorii i tabeli `CategoryBooks`, która powiąże te kategorie z książkami:

```
CREATE TABLE Categories
(
   category_id    SERIAL  NOT NULL,
   category_name  TEXT    NOT NULL,

   PRIMARY KEY(category_id),
   UNIQUE(category_name)
)
;

CREATE TABLE BookCategories
(
   book_id      INTEGER  NOT NULL REFERENCES Publishers,
   category_id  INTEGER  NOT NULL REFERENCES Categories,

   UNIQUE(book_id, category_id)
)
;

CREATE INDEX category_id_index ON BookCategories (category_id)
;
```

12.2. Pisanie aplikacji

Gdy baza danych jest gotowa, można przygotować programy Perla, które będą z nią współpracować. Jeden zestaw programów pozwoli personelowi księgarni wprowadzać, aktualizować i usuwać informacje na temat książek, autorów i wydawców. Drugi przeznaczony będzie dla klientów księgarni, żeby z jego pomocą mogli odnaleźć w bazie danych interesujące ich książki.

12.2.1. Wprowadzanie informacji o autorach

Kusić nas może, aby zacząć od przygotowania aplikacji pozwalającej na wprowadzanie do bazy danych informacji o książkach. Jednak musimy pamiętać, że kolumna `publisher_id` w tabeli `Books` jest kluczem obcym tabeli `Publishers`, co oznacza, ze przed zapełnieniem tabeli `Books` trzeba najpierw wypełnić tabelę `Publishers`.

Pierwszy z przedstawionych programów, *insert-author.pl*, łączy się z serwerem PostgreSQL, pobiera od użytkownika nazwisko autora książki i wykonuje odpowiednie zapytanie `INSERT` wstawiające autora do bazy. Jeśli coś pójdzie nie tak, zakończy pracę, sygnalizując krytyczny błąd.

Program ten wykonuje prosty test integralności danych na poziomie aplikacji, aby upewnić się, że nazwisko autora zawiera przynajmniej jeden znak. Większość aplikacji współpracujących z bazami danych sprawdza wprowadzane dane dwukrotnie: na poziomie definicji tabeli (ograniczenie CHECK) i ponownie na poziomie aplikacji. Pierwszy test sprawdza, czy baza danych nie zawiera niespójnych lub uszkodzonych danych, a drugi pozwala na przechwytywanie błędów i przygotowywanie czytelnych komunikatów o błędach dla użytkownika.

```
#!/usr/bin/perl
# nazwa pliku: insert-author.pl

use strict;
use warnings;
use DBI;

# Połącz się z bazą danych
my $username = 'reuven';
my $password = '';
my $dbh = DBI->connect("DBI:Pg:dbname=coreperl", $username, $password,
                       {'AutoCommit' => 1, 'PrintError' => 1}) ||
    die "Błąd łączenia z bazą: '$DBI::errstr' ";

# Pobierz od użytkownika nazwisko nowego autora
print "Podaj nazwisko autora: ";
my $author_name = <>;
chomp $author_name;

# Nazwisko autora musi zawierać przynajmniej jeden znak
if ($author_name !~ /\w/)
{
    print "Nazwisko autora musi zawierać ";
    print "co najmniej jeden znak.  Kończę pracę.\n";
}

# Jeśli nazwisko autora jest OK, wstaw je
else
{
    # Usuń niepotrzebne spacje z nazwiska autora
    $author_name =~ s/^\s+//g;
    $author_name =~ s/\s+$//g;

    # Przygotuj zapytanie
    my $sql = "INSERT INTO Authors (author_name) VALUES (?) ";

    my $success = $dbh->do($sql, undef, $author_name);


    if ($success)
    {
        print "Wstawiłem do bazy autora:  '$author_name'.\n";
    }
    else
    {
        die "Błąd wywołania zapytania SQL '$sql': '$DBI::errstr' ";
    }

}

# Zamknij połączenie z bazą danych
$dbh->disconnect;
```

Warto przyjrzeć się odwołaniu do metody $dbh->do() w programie *insert-author.pl*. Metoda do interfejsu DBI przeznaczona jest dla tych zapytań INSERT i UPDATE, które nie muszą pobierać wyników zapytania za pośrednictwem uchwytu $sth. Pierwszy argument metody $dbh->do() jest łańcuchem zawierającym instrukcję SQL, a drugi odwołaniem do tablicy asocjacyjnej zawierającej definicje atrybutów (lub wartość undef). Pozostałe argumenty są wartościami, jakie interfejs DBI powinien przypisać symbolom zastępczym w instrukcji SQL. Metoda $dbh->do() zwraca liczbę zmienionych przez nią wierszy w tabeli.

12.2.2. Wprowadzanie kategorii

Program *insert-category.pl* wstawiający nową kategorię do tabeli Categories jest bardzo podobny do programu *insert-author.pl*. Niemniej tym razem istnieje pewna pułapka: kolumna category_name została zdefiniowana jako UNIQUE. Nasza aplikacja mogłaby ignorować ten wymóg, próbując na ślepo wstawiać nowe nazwy kategorii do tabeli Categories. Gdyby baza PostgreSQL odmówiła dodania nowej kategorii, moglibyśmy przechwycić błąd i zaraportować go użytkownikowi.

Tutaj jednak zastosowaliśmy bardziej wyszukane rozwiązanie, które wymaga sprawdzenia bazy danych w celu ustalenia, czy taka nazwa kategorii nie została już wcześniej wprowadzona. To jednak pociąga za sobą konieczność wykonania zapytania SELECT (aby pobrać nazwy już istniejących kategorii), a dopiero potem zapytania INSERT (aby wstawić nową kategorię). Co jednak, jeśli ktoś inny spróbuje wykonać między tymi dwoma zapytaniami podobne zapytanie INSERT?

Rozwiązanie, tak jak to zostało wspomniane w sekcji 11.7.4 w poprzednim rozdziale, polega na połączeniu zapytań SELECT i INSERT w pojedynczej transakcji. W ten sposób upewniamy się, że nikt inny nie zmodyfikuje bazy danych między obydwoma naszymi zapytaniami. Obsługę transakcji można włączyć, przypisując atrybutowi AutoCommit wartość FALSE. Wygodniej jest jednak skorzystać z metody $dbh->begin_work(), która wyłączy atrybut AutoCommit na czas pojedynczej transakcji — do momentu wywołania metody $dbh->commit() zatwierdzającej transakcję lub metody $dbh->rollback() wycofującej transakcję.

Kod programu będzie więc najpierw pytał użytkownika o nazwę nowej kategorii, następnie za pomocą zapytania SELECT sprawdzi, czy takiej nazwy już nie ma, a jeśli taka kategoria już istnieje — przerwie wykonywanie transakcji, przywołując metodę $dbh->rollback(). Wycofanie transakcji nie jest w tym przypadku tak naprawdę konieczne, ponieważ nie zmodyfikowaliśmy żadnej tabeli. Ja jednak zawsze staram się postępować przezornie i przerywam zapytanie za pomocą metody $dbh->rollback(). Gdy już dane przejdą wszystkie testy na poziomie aplikacji, wykonywane jest zapytanie INSERT i wywoływana metoda $dbh->commit() zatwierdzająca transakcję.

Warto zauważyć, że program *insert-category.pl* działa z włączonym atrybutem RaiseError. Oznacza to, że nie ma potrzeby sprawdzania wartości zwracanych przez procedury prepare i execute, ponieważ program zakończy działanie, sygnalizując krytyczny błąd, za każdym razem, gdy natknie się na błąd bazy danych.

```
#!/usr/bin/perl
# nazwa pliku: insert-category.pl
```

```
use strict;
use warnings;
use DBI;

# Połącz się z bazą danych
my $username = 'reuven';
my $password = '';
my $dbh = DBI->connect("DBI:Pg:dbname=coreperl", $username, $password,
                       { 'AutoCommit' => 1, 'RaiseError' => 1}) ||
    die "Błąd łączenia z bazą: '$DBI::errstr' ";

# Pobierz od użytkownika nazwę nowej kategorii
print "Podaj nazwę nowej kategorii: ";
my $category_name = <>;
chomp $category_name;

# Nazwa kategorii musi zawierać przynajmniej jeden znak
if ($category_name !~ /\w/)
{
    print "Nazwa kategorii musi zawierać ";
    print "co najmniej jeden znak.  Exiting.\n";
}

# Jeśli nazwa kategorii jest OK, spróbuj ją wstawić
else
{
    # Usuń niepotrzebne spacje
    $category_name =~ s/^\s+//g;
    $category_name =~ s/\s+$//g;

    # Wyłącz AutoCommit na czas jednej transakcji. Pozostanie on wyłączony
    # do czasu wywołania metody $dbh->commit() lub $dbh->rollback()

    $dbh->begin_work();

    # ------------------------------------------------------------
    # Czy taka nazwa kategorii już jest w bazie danych?
    # Jeśli tak, wycofaj transakcję, poinformuj użytkownika o problemie
    # i zakończ pracę

    # Utwórz zapytanie
    my $select_sql = "SELECT COUNT(category_id) ";
    $select_sql .= "FROM Categories where category_name = ?";

    # Przygotuj instrukcję
    my $sth = $dbh->prepare($select_sql);

    # Wykonaj instrukcję
    $sth->execute($category_name);

    # Pobierz wynik
    my ($category_already_exists) = $sth->fetchrow_array;

    # Zakończ to zapytanie
    $sth->finish;

    if ($category_already_exists)
```

```
    {
        print "Kategoria '$category_name' była już ";
        print "dodana do bazy.\nSpróbuj wpisać inną nazwę.\n";

        # Transakcja wycofana.
        $dbh->rollback();
    }

    else
    {
        # Przygotuj zapytanie INSERT
        my $sql = "INSERT INTO Categories (category_name) VALUES (?) ";

        # Przygotuj instrukcję
        my $successfully_inserted = $dbh->do($sql, undef, $category_name);

        if ($successfully_inserted)
        {
            print "Dodałem do bazy kategorię '$category_name'.\n";
            $dbh->commit();
        }
        else
        {
            print "Błąd wstawiania: '$DBI::errstr'.\n";
            $dbh->rollback();
        }
    }
}

# Zakończ połączenie z bazą danych
$dbh->disconnect;
```

Jak łatwo przewidzieć, program *insert-publisher.pl* będzie wyglądał bardzo podobnie do programu *insert-category.pl*.

12.2.3. Modyfikowanie już istniejących wartości

Program *insert-category.pl* działa całkiem sprawnie — co jednak, jeśli ktoś wprowadzi do bazy złą nazwę kategorii? Baza danych księgarni potrzebuje jakiegoś sposobu modyfikowania już istniejących nazw kategorii (to samo dotyczy wstawionych nazw autorów i wydawców, ale tutaj omawiać będziemy tylko przypadek kategorii).

Program *update-category.pl* będzie wymagał podania przez użytkownika dwóch wartości: starej nazwy kategorii `$old_category_name` i nowej kategorii `$new_category_name`. Może się jednak zdarzyć, że użytkownik popełni błąd przy wpisywaniu starej nazwy kategorii, podając nazwę, której nie ma w bazie danych, i wówczas klauzula `WHERE` zapytania SQL nie będzie pasowała do żadnego wiersza tabeli `Categories`.

Moduł `Term::Complete` dostępny w sieci `CPAN` (patrz: podrozdział 1.6) pozwala na ograniczenie liczby przyjmowanych od użytkownika danych wejściowych. Moduł `Term::Complete` automatycznie eksportuje do przestrzeni nazw kodu przywołującego procedurę `Complete`.

Procedura `Complete` wymaga przesłania jej dwóch argumentów. Pierwszy z nich to tekst komunikatu proszącego o wprowadzenie danych, który zostanie wyświetlony użytkownikowi, a drugi jest listą lub też odwołaniem do tablicy, które zawierać będą dopuszczalne odpowiedzi użytkownika. Użytkownik może w każdej chwili wcisnąć klawisze *CTL+D*, by wyświetlić listę dopuszczalnych odpowiedzi. Jeśli użytkownik wprowadzi wystarczającą liczbę znaków, aby było możliwe jednoznaczne zidentyfikowanie danego elementu listy, wciśnięcie klawisza *TAB* pozwoli uzupełnić resztę.

Program *update-category.pl* pobierze listę bieżących kategorii (która przesłana zostanie procedurze `Complete`), korzystając z zapytania `SELECT`, a następnie będzie modyfikować bazę za pomocą zapytania `UPDATE`. Oba zapytania zostaną połączone w jedną transakcję za pomocą metody `$dbh->begin_work`, podobnie jak to było w programie *insert-category.pl*.

Lista kategorii pobierana jest z tabeli `Categories`. Zamiast jednak umieszczać nazwy kategorii w tablicy, uczynimy je kluczami tablicy asocjacyjnej `%categories`. Sprawdzenie, czy dany klucz istnieje w tablicy asocjacyjnej, zajmuje bowiem znacznie mniej czasu niż odszukanie elementu w tablicy, tak więc program będzie dzięki temu działał szybciej.

A oto jedna z możliwych implementacji programu *update-category.pl*:

```
#!/usr/bin/perl
# nazwa pliku: update-category.pl

use strict;
use warnings;
use DBI;
use Term::Complete;

# Połącz się z bazą danych
my $username = 'reuven';
my $password = '';
my $dbh = DBI->connect("DBI:Pg:dbname=coreperl", $username, $password,
                       { 'AutoCommit' => 1, 'RaiseError' => 1}) ||
    die "Błąd łączenia z bazą: '$DBI::errstr' ";

my %categories = ();

# ------------------------------------------------------------
# Rozpocznij transakcję, która będzie trwała do czasu
# wywołania metody $dbh->commit() lub $dbh->rollback()
$dbh->begin_work();

# Pobierz nazwy kategorii i uczyń je kluczami %categories
my $sql = "SELECT category_name from Categories";
my $sth = $dbh->prepare($sql);
$sth->execute;

while (my $rowref = $sth->fetchrow_arrayref)
{
    # Dodaj nowy klucz do %categories, z wartością 1
    $categories{$$rowref[0]} = 1;
}

# Zakończ instrukcję
$sth->finish();
```

```
my $old_category_name = "";

# Pobierz istniejącą kategorię, używając Term::Complete. Gdy przerwiemy
# tę pętlę, zmienna $old_category_name będzie zawierać nazwę
# istniejącej kategorii
until (defined $categories{$old_category_name})
{
    $old_category_name =
        Complete("Wprowadź nazwę istniejącej kategorii: ", keys %categories);
}

# ------------------------------------------------------------
# Pobierz nową nazwę kategorii

print "Zmień '$old_category_name' na: ";
my $new_category_name = <>;
chomp $new_category_name;

# Nazwa kategorii musi zawierać przynajmniej jeden znak
if ($new_category_name !~ /\w/)
{
    print "Błąd: Nazwa kategorii musi zawierać co najmniej jeden znak!\n";
}

# Jeśli nazwy kategorii są OK, sprawdź, czy nie ma już jej
# w bazie danych
else
{
    # Usuń z nazwy niepotrzebne spacje
    $new_category_name =~ s/^\s+//g;
    $new_category_name =~ s/\s+$//g;

    # Jeśli $new_category_name jest w tabeli %categories, musimy
    # przerwać, aby nazwy się nie powtarzały
    if (defined $categories{$new_category_name})
    {
        # Zasygnalizuj błąd użytkownikowi i wycofaj transakcję
        print "Błąd: Kategoria '$new_category_name' już istnieje w bazie danych!\n";
        $dbh->rollback();
        $dbh->disconnect();
        exit;
    }

    # Utwórz zapytanie
    my $sql = "UPDATE Categories ";
    $sql .= "SET category_name = ? ";
    $sql .= "WHERE category_name = ? ";

    # Przygotuj instrukcję
    my $number_of_affected_rows =
        $dbh->do($sql, undef, $new_category_name, $old_category_name);

    if ($number_of_affected_rows == 1)
    {
        # Raportuj wynik użytkownikowi i zatwierdź transakcję
        print "Zmieniłem nazwę '$old_category_name' na '$new_category_name'.\n";
        $dbh->commit();
    }
```

```
    else
    {
        # Raportuj błąd użytkownikowi i wycofaj transakcję
        print "Zmiana nazwy nie została wykonana.\n";
        $dbh->rollback();
    }
}

# Zakończ połączenie z bazą danych
$dbh->disconnect();
```

12.2.4. Dodawanie nowych książek

Po dodaniu kilku pozycji do tabel `Authors`, `Publishers` i `Categories` można przystąpić do wprowadzania do bazy danych księgarni nowych książek. Dodanie nowej książki oznacza konieczność wstawienia jednego wiersza do tabeli `Books` i jednego lub więcej wierszy do tabeli `BookAuthors`.

Kolejny program, *insert-book.pl*, wykorzystuje większość technik, które opisywałem wcześniej w tym rozdziale: wykonuje zapytanie pobierające z bazy danych listę już wprowadzonych wydawców i autorów, wykorzystuje moduł `Term::Complete`, dzięki któremu użytkownik podaje odpowiednią nazwę wydawnictwa i nazwisko autora, i wreszcie wstawia wiersze do tabel `Books` i `BookAuthors`. Co więcej, przeprowadza to wszystko w ramach jednej transakcji, dzięki czemu opisana sekwencja transakcji zostaje wykonana łącznie i niepowodzenie jednej oznaczać będzie niepowodzenie wszystkich.

Warto również zauważyć, że korzystamy tutaj z właściwego tylko bazie PostgreSQL atrybutu `pg_oid_status`, który zwraca identyfikator (OID) ostatnio wstawionego obiektu.

Identyfikatory obiektów są elementem specyficznym dla PostgreSQL. Można postrzegać je jako niewidoczny, globalny klucz główny każdego wiersza w bazie danych. Identyfikator OID ostatnio wprowadzonego wiersza pozwala na pobranie klucza głównego dla ostatnio wykonanej operacji `INSERT`.

Na koniec warto przyjrzeć się, w jaki sposób wiersze wstawiane są do tabeli `BookAuthors` za pomocą pojedynczego odwołania do metody `prepare` i kilkakrotnego przywoływania metody `execute`. W ten sposób odciążamy program Perla i przyśpieszamy znacznie wykonywanie zapytania, jeśli baza, z której korzystamy, przechowuje zapytania w pamięci podręcznej.

Program *insert-book.pl* jest jak dotąd najdłuższym programem związanym z bazami danych, które były omawiane w tej książce. Ponieważ jednak wszystkie jego elementy pojawiały się już w poprzednio prezentowanych programach, zrozumienie go nie powinno być zbyt trudne.

```
#!/usr/bin/perl
# nazwa pliku: insert-book.pl

use strict;
use warnings;
use DBI;
```

```
use Term::Complete;

print "Witamy w programie 'Wstaw książkę'\n\n";

# Połącz się z bazą danych
my $username = 'reuven';
my $password = '';
my $dbh = DBI->connect("DBI:Pg:dbname=coreperl", $username, $password,
                       {
                        'AutoCommit' => 1, 'RaiseError' => 1}) ||
    die "Błąd łączenia z bazą: '$DBI::errstr' ";

# Zdefiniuj parę często używanych zmiennych
my ($sql, $sth);

# ------------------------------------------------------------
# Zacznij nową transakcję, którą zakończy wywołanie $dbh->commit()
# lub $dbh->rollback().
$dbh->begin_work();

# ------------------------------------------------------------
# Pobierz autorów i uczyń ich kluczami tablicy asocjacyjnej
my %authors = ();

$sql  = "SELECT author_name, author_id ";
$sql .= "FROM Authors";

# Przygotuj instrukcję
$sth = $dbh->prepare($sql);

# Wykonaj instrukcję
$sth->execute;

while (my $rowref = $sth->fetchrow_arrayref)
{
    my ($author_name, $author_id) = @{$rowref};
    $authors{$author_name} = $author_id;
}

# ------------------------------------------------------------
# Pobierz wydawców i uczyń ich kluczami tablicy asocjacyjnej
my %publishers = ();

$sql  = "SELECT publisher_name, publisher_id ";
$sql .= "FROM Publishers";

# Przygotuj instrukcję
$sth = $dbh->prepare($sql);

# Wykonaj instrukcję
$sth->execute;

while (my $rowref = $sth->fetchrow_arrayref)
{
    my ($publisher_name, $publisher_id) = @{$rowref};
    $publishers{$publisher_name} = $publisher_id;
}
```

```
# ------------------------------------------------------------
# Pobierz informację od użytkownika
my $isbn = get_info("ISBN");
my $title = get_info("tytuł");

# Pobierz wielu autorów
my @authors;
while (my $author = get_info("autora", keys %authors))
{
    last unless $author;
    push @authors, $author;
}

my $publisher = get_info("wydawcę", keys %publishers);
my $pages = get_info("liczbę stron");
my $pub_date = get_info("datę wydania (YYYY-MM-DD)");
my $us_dollar_price = get_info("cenę w dolarach");

# ------------------------------------------------------------
# Wstaw do bazy nową książkę
$sql  = "INSERT INTO Books ";
$sql .= " (isbn, title, publisher_id, num_pages, ";
$sql .= "  pub_date, us_dollar_price) ";
$sql .= "VALUES (?, ?, ?, ?, ?, ?) ";

$sth = $dbh->prepare($sql);

$sth->execute($isbn, $title, $publishers{$publisher}, $pages,
              $pub_date, $us_dollar_price);

my $new_book_oid = $sth->{pg_oid_status};

$sth->finish();

# ------------------------------------------------------------
# Użyj identyfikatora OID nowej książki do pobrania book_id

$sql  = "SELECT book_id ";
$sql .= "FROM Books ";
$sql .= "WHERE oid = ?";

# Przygotuj instrukcję
$sth = $dbh->prepare($sql);

# Wykonaj instrukcję
$sth->execute($new_book_oid);

# Pobierz book_id z wiersza o identyfikatorze OID
my ($new_book_id) = $sth->fetchrow_array;

$sth->finish();

# ------------------------------------------------------------
# Wstaw powiązanie książki z autorem
$sql  = "INSERT INTO BookAuthors (book_id, author_id) VALUES (?, ?) ";

$sth = $dbh->prepare($sql);
```

```
foreach my $author (@authors)
{
    $sth->execute($new_book_id, $authors{$author});
    print "Gotowe.\n";
}

# Instrukcja zakończona
$sth->finish();

# Jeśli wszystko poszło dobrze, zatwierdzamy transakcję
$dbh->commit();

# Zamykamy połączenie z bazą danych
$dbh->disconnect();

# ------------------------------------------------------------
# Procedura pobierająca informacje od użytkownika
sub get_info
{
    my $question = shift;
    my @completions = @_;
    my %completions_hash = map {($_, 1)} @completions;

    my $input = "";

    # Używamy Term::Complete, jeśli trzeba uzupełnić dane
    if (@completions)
    {
        $input =
            Complete("Podaj $question książki: ", keys %completions_hash);
    }

    # Jeśli nie ma takiej potrzeby, rozpoczynamy standardowo
    else
    {
        print "Podaj $question książki: ";
        $input = <>;            # Pobierz dane od użytkownika
        chomp $input;           # Usuń z końca znak nowego wiersza

        $input =~ s/^\s+//g;    # Usuń spacje z początku
        $input =~ s/\s+$//g;    # Usuń spacje z końca
    }

    return $input;
}
```

Mimo iż program *insert-book.pl* nie obsługuje kategorii, nietrudno domyślić się, jak powinien wyglądać odpowiedni kod, który należałoby w tym celu dodać do programu.

12.2.5. Pobieranie informacji o książkach

Teraz, gdy baza danych księgarni jest gotowa, pora przygotować prostą aplikację wykonującą zapytania, które umożliwią użytkownikom zdobycie informacji na temat konkretnych książek. Napiszemy przykładowy program, który będzie zwracał książki o tytułach zawierających tekst wprowadzony przez użytkownika.

Warto zauważyć, że najpierw wczytujemy wszystkie informacje o książce do tablicy asocjacyjnej `%book`, a dopiero później je wyświetlamy. Dzieje się tak, ponieważ baza PostgreSQL zwróci po jednym wierszu dla każdego autora książki. Gdybyśmy po prostu wyświetlili wyniki, tak jak są one zwracane przez bazę danych, książki mające dwóch autorów byłyby wyświetlane dwukrotnie, a książki mające pięciu autorów — aż pięć razy! Unikniemy takich powtórzeń, wczytując wszystkie zwrócone książki do tablicy asocjacyjnej tablic asocjacyjnych, którą następnie posortujemy i wyświetlimy to, co nam pozostanie.

Porządek, w jakim będą zwracane wyniki, może wydawać się nieistotny, skoro i tak zostaną one uporządkowane w pętli `foreach`. Nie jest to jednak do końca prawda: autorzy są dodawani do tablicy asocjacyjnej `%book` w takim samym porządku, w jakim są pobierani. Dlatego właśnie klauzula `ORDER BY` upewnia się, że zwracane wiersze będą uporządkowane według nazwisk autorów.

Aby uniknąć pomyłek, które mogą być spowodowane różnicami w wielkości liter, wykorzystywana jest funkcja `LOWER` języka SQL, która powoduje zamianę wszystkich wielkich znaków ciągów, które są jej argumentami, na małe:

```
#!/usr/bin/perl
# nazwa pliku: query-title.pl

use strict;
use warnings;
use DBI;

print qq{Witamy w programie "Zapytania".\n\n};

# Połącz się z bazą danych
my $username = 'reuven';
my $password = '';
my $dbh = DBI->connect("DBI:Pg:dbname=coreperl", $username, $password,
                       { 'AutoCommit' => 1, 'RaiseError' => 1}) ||
    die "Błąd łączenia z bazą: '$DBI::errstr' ";

# Pobierz tytuł od użytkownika
print "Podaj poszukiwany napis: ";
my $target = <>;
chomp $target;

# Weź w cudzysłów metaznaki SQL
$target =~ s|%|\\\%|g;
$target =~ s|_|\\\_|g;

# Przekształć łańcuch w wyrażenie regularne SQL
$target = "%$target%";

# Utwórz zapytanie
my $sql = "SELECT B.isbn, B.title, P.publisher_name, B.num_pages, ";
$sql .= "      B.pub_date, B.us_dollar_price, A.author_name ";
$sql .= "FROM Books B, Publishers P, Authors A, BookAuthors BA ";
$sql .= "WHERE LOWER(B.title) LIKE LOWER(?) ";
$sql .= "  AND BA.author_id = A.author_id ";
$sql .= "  AND B.book_id = BA.book_id ";
$sql .= "  AND B.publisher_id = P.publisher_id ";
$sql .= "ORDER BY A.author_name ";
```

```
# Przygotuj instrukcję
my $sth = $dbh->prepare($sql);

# Wykonaj instrukcję
my $success = $sth->execute($target);

if ($success)
{
    my %book = ();

    while (my $rowref = $sth->fetchrow_arrayref)
    {
        my ($isbn, $title, $publisher, $pages,
            $date, $price, $author) = @$rowref;

        # Zachowaj informację o książce w %book
        $book{$isbn}->{title} = $title;
        $book{$isbn}->{publisher} = $publisher;
        $book{$isbn}->{pages} = $pages;
        $book{$isbn}->{date} = $date;
        $book{$isbn}->{price} = $price;

        # Zajmij się autorami
        if ($book{$isbn}->{author})
        {
            $book{$isbn}->{author} .= ", $author";
        }
        else
        {
            $book{$isbn}->{author} = "$author";
        }
    }

    # Przejrzyj książki w pętli i wyświetl informacje o nich
    foreach my $isbn
        (sort {$book{$a}->{title} cmp $book{$b}->{title}} keys %book)
        {
            print "\n";
            print "$book{$isbn}->{title}\n";
            print "-" x length($book{$isbn}->{title}), "\n";

            print "Autor: $book{$isbn}->{author}\n";
            print "ISBN: $isbn\tWydawca: $book{$isbn}->{publisher}\n";
            print "Data wydania: $book{$isbn}->{date}\t";
            print "Cena: \$$book{$isbn}->{price}\n";
            print "Liczba stron: $book{$isbn}->{pages}\n";
        }

    if ($sth->rows == 0)
    {
        print "Brak książek pasujących do wzorca.\n";
    }

}
else
{
    print "Błąd przygotowania zapytania: $DBI::errstr";
}

$sth->finish();
$dbh->disconnect();
```

12.3. Usuwanie błędów w programach korzystających z DBI

Perl jest językiem programowania, w którym błędy wykrywa się bardzo łatwo. Wielu programistów zamiast korzystać z wbudowanego debuggera (patrz: sekcja 13.4.2), po prostu wstawia do kodu kilka instrukcji `print`. Tego rodzaju techniki nie zawsze jednak wystarczają, gdy korzystamy z interfejsu `DBI`.

Jeśli na przykład program zakończy pracę w wyniku błędu bazy danych po próbie wykonania instrukcji `INSERT` wstawiającej wiersz do bazy danych `BookAuthors`, dobrze by było wiedzieć, jak dokładnie wyglądało ostateczne zapytanie SQL, zanim zostało wysłane do bazy danych. Jeśli korzystamy w zapytaniu z symboli zastępczych, okaże się, że uzyskanie tego rodzaju informacji jest bardzo trudne.

Rozwiązaniem jest metoda `$dbh->trace()`, która pomaga w usuwaniu błędów z programów korzystających z interfejsu `DBI`. Wymaga ona jednego obowiązkowego argumentu, który powinien być liczbą całkowitą z przedziału od 0 (zupełny brak wykrywania błędów) do 9 (poziom wykrywania błędów oferujący znacznie więcej informacji niż prawdopodobnie kiedykolwiek będziesz potrzebować). Drugi argument jest opcjonalny i podaje nazwę pliku na dysku, do którego mają być zapisywane informacje debugowania. Domyślnie wszystkie informacje dostarczane przez metodę `$dbh->trace()` trafiają do standardowego strumienia błędów `STDERR`.

Po aktywacji metoda `$dbh->trace()` działać będzie przez cały czas pracy programu lub do momentu, gdy zostanie wyłączona. Bardzo często metodę `$dbh-trace()` wykorzystuje się w trakcie tworzenia programu, by wyłączyć ją, kiedy program zaczyna normalne działanie.

Można również wywołać metodę `trace` jako metodę klasy wprost z modułu `DBI`, wpisując `DBI->trace()`. Pozwoli to na śledzenie działania metody `DBI->connect()`, która zwraca uchwyt bazy danych `$dbh`.

Żeby pokazać, jakie informacje zwraca metoda `$dbh->trace()`, zaprezentuję tutaj zapis działania programu *insert-book.pl*, dla którego metoda `$dbh->trace()` została włączona na pierwszym poziomie debugowania. Do kodu programu *insert-book.pl* dodałem tylko jeden wiersz, zaraz po odwołaniu do metody `DBI->connect()`:

```
$dbh->trace(1, '/tmp/insert-book-trace.txt');
```

Oto, co pojawiło się na ekranie podczas działania programu *insert-book.pl*:

```
Podaj ISBN książki: 123-456-789
Podaj tytuł książki: I like Perl
Podaj autora książki: Reuven M. Lerner
Podaj autora książki:
Podaj wydawcę książki: Prentice-Hall
Podaj liczbę stron książki: 200
Podaj datę wydania (YYYY-MM-DD) książki: 2001-01-01
Podaj cenę w dolarach książki: 20
Gotowe.
```

W tym czasie za kulisami moduł DBI zebrał następujące informacje o tej sesji:

```
    DBI:db=HASH(0x8237fc0) trace level set to 1 in DBI 1.20-nothread
dbd_db_FETCH
2   <- FETCH= 1 at DBI.pm line 1051
dbd_db_STORE
2   <- STORE('AutoCommit' 0 ...)= 1 at DBI.pm line 1053
dbd_db_STORE
2   <- STORE('BegunWork' 1 ...)= 1 at DBI.pm line 1054
    <- begin_work= 1 at insert-book.pl line 27
dbd_st_prepare: statement = >SELECT author_name, author_id FROM Authors<
dbd_st_preparse: statement = >SELECT author_name, author_id FROM Authors<
    <- prepare:('SELECT author_name, author_id FROM Authors' CODE)=
      DBI::st=HASH(0x8238140) at insert-book.pl line 37
dbd_st_execute
    <- execute(CODE)= 2 at insert-book.pl line 40
dbd_st_fetch
    <- fetchrow_arrayref= [ 'Foo Bar' '1' ] row1 at insert-book.pl line 42
dbd_st_fetch
dbd_st_fetch
    <- fecthrow_arrayref= undef row2 at insert-book.pl line 42
dbd_st_prepare: statement = >SELECT publisher_name, publisher_id FROM Publishers<
dbd_st_preparse: statement = >SELECT publisher_name, publisher_id FROM Publishers<
    <- prepare:('SELECT publisher_name, publisher_id FROM Publishers'
       CODE)= DBI::st=HASH(0x82381dc) at insert-book.pl line 56
dbd_st_destroy
    <- DESTROY= undef at insert-book.pl line 59
dbd_st_execute
    <- execute(CODE)= 3 at insert-book.pl line 59
dbd_st_fetch
    <- fetchrow_arrayref= ['Prentice-Hall' '1' ] row1 at insert-book.pl line 61
dbd_st_fetch
dbd_st_fetch
dbd_st_fetch
    <- fetchrow_arrayref= undef row 3 at insert-book.pl line 61
dbd_st_prepare: statement = >INSERT INTO Books
    (isbn, title, publisher_id, num_pages,  pub_date, us_dollar_price)
    VALUES (?, ?, ?, ?, ?, ?) <
dbd_st_preparse: statement = >INSERT INTO Books
    (isbn, title, publisher_id, num_pages,
     pub_date, us_dollar_price) VALUES (?, ?, ?, ?, ?, ?) <
    <- prepare('INSERT INTO Books
    (isbn, title, publisher_id, num_pages,
     pub_date, us_dollar_price) VALUES (?, ?, ?, ?, ?, ?) '
     CODE)= DBI::st=HASH(0x823f2e4) at insert-book.pl line 92
dbd_st_destroy
    <- DESTROY= undef at insert-book.pl line 94
dbd_bind_ph
dbd_st_rebind
dbd_bind_ph
dbd_st_rebind
dbd_bind_ph
dbd_st_rebind
dbd_bind_ph
dbd_st_rebind
```

```
dbd_bind_ph
dbd_st_rebind
dbd_bind_ph
dbd_st_rebind
dbd_st_execute
    <- execute('123-456-789' 'I like Perl' ...)= 1 at insert-book.pl line 94
dbd_st_FETCH
    <- FETCH= '121271' at insert-book.pl line 97
    <- finish= 1 at insert-book.pl line 99
dbd_st_prepare: statement = >SELECT book_id FROM Books WHERE oid = ?<
dbd_st_preparse: statement = >SELECT book_id FROM Books WHERE oid = ?<
    <- prepare('SELECT book_id FROM Books WHERE oid = ?'
CODE)= DBI::st=HASH(0x818d458) at insert-book.pl line 109
dbd_st_destroy
    <- DESTROY= undef at insert-book.pl line 112
dbd_bind_ph
dbd_st_rebind
dbd_st_execute
    <- execute('121271' CODE) at insert-book.pl line 112
dbd_st_fetch
    <- fetchrow_array= ( '19' ) [1 items] row 1 at insert-book.pl line 115
dbd_st_finish
    <- finish= 1 at insert-book.pl line 117
dbd_st_prepare: statement = >INSERT INTO BookAuthors
    (book_id, author_id) VALUES (?, ?) <
dbd_st_preparse: statment = >INSERT INTO BookAuthors
    (book_id, author_id) VALUES (?, ?) <
    <- prepare('INSERT INTO BookAuthors (book_id, author_id)
    VALUES (?, ?) ' CODE)= DBI::st=HASH(0x82381e8) at insert-book.pl line 123
dbd_st_destroy
    <- DESTROY= undef at insert-book.pl line 125
dbd_bind_ph
dbd_st_rebind
dbd_bind_ph
dbd_st_rebind
dbd_st_execute
    <-execute('19' '2' ...)= 1 at insert-book.pl line 127
    <-finish= 1 at insert-book.pl line 132
dbd_db_commit
    <- commit= 1 at insert-book.pl line 135
dbd_db_STORE
2   <- STORE('Autocommit' 1 ...) = 1 at insert-book.pl line 135
dbd_db_disconnect
    <- disconnect= 1 at insert-book.pl line 138
dbd_st_destroy
    <- DESTROY= undef
dbd_st_destroy
    <- DESTROY= undef
```

Jak widać, metoda `$dbh->trace(1)` dostarcza kompletnego dziennika rejestrującego każdą zmianę atrybutu oraz każdą wywołaną metodę `prepare`, `execute` lub `$dbh->commit()`.

12.4. Podsumowanie

W tym rozdziale poszerzyliśmy wiedzę na temat teorii baz danych. Poznaliśmy również kilka standardowych rozwiązań z zakresu projektowania baz danych, które niezmiernie ułatwiają tworzenie prawdziwych aplikacji współpracujących z bazami. Przyjrzeliśmy się dokładnie przykładowym programom obsługującym katalog książek oferowanych przez pewną księgarnię. W szczególności omawialiśmy:

- znaczenie normalizacji danych, która czyni bazy danych wydajniejszymi i łatwiejszymi w obsłudze,
- sposoby projektowania bazy danych, tak by była jak najwszechstronniejsza i pozostawała wydajna, niezależnie od tego, ile informacji będzie przechowywać,
- wykorzystanie modułu `Term::Complete` do narzucania ograniczeń na dane wprowadzane przez użytkownika,
- korzystanie z metody `$dbh->trace()` ułatwiającej wyszukiwanie błędów w aplikacjach dla baz danych.