Lekkie i mobilne bazy danych

dr inż. Sebastian Ernst

Katedra Informatyki Stosowanej
Akademia Górniczo-Hutnicza

Materiały do przedmiotu
Zaawansowane Technologie Bazodanowe

Plan wykładu

Literatura

Wykład oparto na materiałach z:

- dokumentacja SQLite 3,
- Techtopia, Working with iOS 5 iPhone Databases using Core Data,
- Apple Inc., Learning Objective-C: A Primer,
- Reto Meier, *Professional Android 2 Application Development*, Wiley, 2010,
- Lars Vogel, Android SQLite Database and ContentProvider - Tutorial

Plan wykładu

Czym jest SQLite?

- Wolnostojąca, samowystarczalna, bezserwerowa relacyjna baza danych.
- Nie wymaga konfiguracji.
- Niewielka: 350 KB (w wersji 32-bitowej).
- Często wybierana przez programistów jako „format pliku” danych aplikacji.
- Wbudowana m.in. w: Firefox, Mac OS X, iOS, PHP, Skype, Symbian, McAfee AntiVirus, Android, Solaris 10.
- Szacowane 500 milionów kopii, vs. 100 milionów dla „dużych” silników SQL.

Cechy SQLite

- Cała baza (z tabelami, indeksami, wyzwalaczami, widokami, itd.) znajduje się w jednym, przenośnym pliku.
- Cały silnik jest napisany w języku C i korzysta tylko z podstawowych funkcji bibliotecznych.
- Kod SQLite dostępny jest jako jeden duży plik (tzw. *amalgamation*) `sqlite3.c` (z plikiem nagłówkowym `sqlite3.h`) – wystarczy dodać go do projektu aby zacząć korzystać z funkcji SQLite.
- W odróżnieniu od DBMS w architekturze klient/serwer, SQLite jest bezserwerowa. Jest to jedyna baza bezserwerowa z wsparciem dla wielodostępu.
- Posiada wsparcie dla prawie całego standardu SQL-92 (szczegóły później).

Transakcje w SQLite

- Od transakcyjnej bazy danych wymagamy, aby wszystkie zapytania były ACID (Atomic, Consistent, Isolated, Durable).
- SQLite wspiera transakcje na poziomie *serializable* (najwyższy poziom izolacji transakcji).
- ACID zachowany jest nawet w przypadku przerwania operacji zapisu przez:
  - awarię aplikacji,
  - awarię systemu operacyjnego,
  - przerwę w zasilaniu i niespodziewane wyłączenie maszyny.

Wydajność SQLite

- Przyjmuje się, że wydajność SQLite jest *porównywalna* do popularnych DBMS typu klient-serwer.
- Testy przeprowadzane w różnych scenariuszach i dla różnych wersji tych systemów różnią się wynikami.

Według autorów

„SQLite należy traktować jako zamiennik dla *fopen()*, a nie jako zamiennik dla Oracle.”

Standardowe interfejsy SQLite

- Podstawowym narzędziem jest klient shellowy `sqlite3`. Obowiązkowo przyjmuje co najmniej jeden argument, którym jest nazwa pliku z bazą.
- SQLite standardowo udostępnia interfejsy programistyczne dla C, C++ oraz TCL:
  - interfejs C/C++ zawiera wszystkie funkcje konieczne do skorzystania z SQLite i jest w pełni udokumentowany,
  - interfejs TCL oparty jest o polecenie `sqlite3` i także posiada pełną specyfikację.

Wsparcie dla standardów

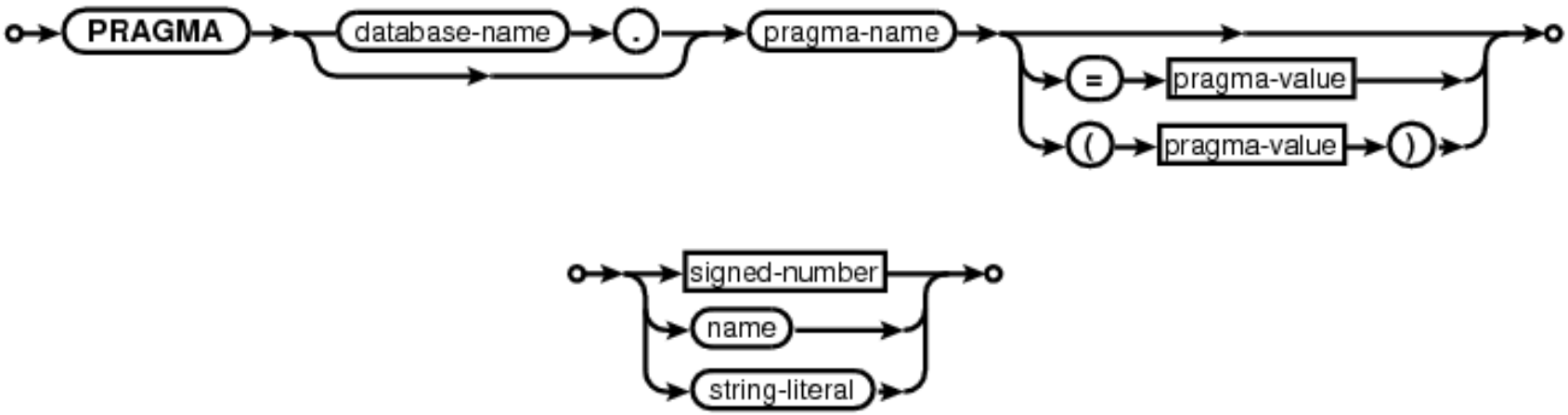
SQLite wspiera standard SQL-92, z następującymi wyjątkami:

- złączenia zewnętrzne prawe i pełne (`RIGHT` oraz `FULL OUTER JOIN`),
- wsparcie dla `ALTER TABLE` ograniczone jest do `RENAME TABLE` oraz `ADD COLUMN`; brakuje wsparcia dla `DROP COLUMN`, `ALTER COLUMN`, `ADD CONSTRAINT`, itd.,
- wspierane są tylko wyzwalacze `FOR EACH ROW` (brak wsparcia dla `FOR EACH STATEMENT`),
- widoki (perspektywy) są *read-only*,
- prawa dostępu kontrolowane są na poziomie systemu plików, polecenia `GRANT` i `REVOKE` nie miałyby więc sensu.

Polecenia PRAGMA

Polecenia PRAGMA są specyficznym dla SQLite rozszerzeniem języka SQL i służą do:

- zmiany konfiguracji SQLite,
- odczytu danych wewnętrznych.



Przykłady poleceń PRAGMA I

- `auto_vacuum` (NONE, FULL, INCREMENTAL) – automatyczne wywoływanie operacji VACUUM,
- `cache_size` – rozmiar pamięci podręcznej,
- `database_list` – aktualnie otwarte bazy,
- `encoding` – określa kodowania (np. UTF-8, UTF-16),
- `foreign_keys` – czy mają być sprawdzane ograniczenia kluczy obcych,
- `locking_mode` – tryb blokowania dostępu,
- `synchronous` – czy czekać, aż dane zostaną zapisane fizycznie na dysk,
- `table_info` – wyświetla informacje o tabeli.

Klient shellowy

```
$ sqlite3 demo.db
SQLite version 3.7.7 2011-06-25 16:35:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Demo...

Interfejs C/C++ – najważniejsze funkcje

- `sqlite3_open` – otwiera bazę,
- `sqlite3_prepare` – przygotowuje zapytanie,
- `sqlite3_step` – wykonuje zapytanie aż do uzyskania pierwszego/kolejnego wiersza wyników,
- `sqlite3_column` – zwraca jedną kolumnę z wiersza wyników,
- `sqlite3_finalize` – niszczy wcześniej przygotowane zapytanie,
- `sqlite3_close` – zamyka połączenie z bazą,
- `sqlite3_exec` – skrót `prepare`, `step`, `column` i `finalize` wykorzystujący funkcję callback,
- `sqlite3_table` – j.w., ale buforuje wyniki zamiast wywoływać callback.

Klient w C

```c
#include <stdio.h>
#include <sqlite3.h>

static int callback(void *nic, int argc, char **argv, char **kol) {
  int i;
  for(i=0; i<argc; i++)
    printf("%s = %s\n", kol[i], argv[i] ? argv[i] : "NULL");
  printf("\n");
  return 0;
}

int main(int argc, char **argv) {
  sqlite3 *db;
  char *zErrMsg = 0;
  int rc;
  rc = sqlite3_open("../demo.db", &db);
  rc = sqlite3_exec(db, "SELECT * FROM pracownicy", callback,
                    0, NULL);
  sqlite3_close(db);
  return 0;
}
```

SQLite3 w PHP – klasa SQLite3

Dostęp do baz SQLite3 uzyskujemy przy pomocy klasy `SQLite3`.[1]
Najważniejsze metody:

- `open` – otwiera bazę,
- `prepare` – przygotowuje zapytanie, tworzy obiekt `SQLite3Stmt`,
- `query` – wykonuje zapytanie bezpośrednio, tworzy obiekt `SQLite3Result`,
- `exec` – dla zapytań nie zwracających wyniku,
- `changes` – liczba wierszy zmienionych ostatnim zapytaniem,
- `lastInsertRowID` – identyfikator ostatnio dodanego wiersza,
- `close` – zamyka bazę.

[1] Funkcje `sqlite_*` działały do wersji 2 włącznie.

SQLite3 w PHP – klasa SQLite3Result

Obiekty tej klasy przechowują wyniki zapytań. Najważniejsze metody:

- `columnName` – zwraca nazwę n-tej kolumny,
- `columnType` – zwraca typ n-tej kolumny,
- `fetchArray` – pobiera kolejny wiersza jako tablicę zwykłą lub asocjacyjną,
- `finalize` – zamyka wyniki zapytania,
- `numColumns` – liczba kolumn w wynikach,
- reset – powrót do pierwszego wiersza.

SQLite3 w PHP – prosty przykład

```php
$db = new SQLite3('../demo.db');

$result = $db->query('SELECT * FROM pracownicy');

while ($row = $result->fetchArray(SQLITE3_ASSOC)) {
    print_r($row);
}

$db->close();
```

SQLite3 w PHP – przykład z argumentem

```
$db = new SQLite3('../demo.db');

$query = "SELECT * FROM pracownicy WHERE nazwisko='".
                                              $argv[1]."'";
$result = $db->query($query);

while ($row = $result->fetchArray(SQLITE3_ASSOC)) {
  print_r($row);
}

$db->close();
```

SQLite3 w PHP – klasa SQLite3Stmt

Obiekt tej klasy tworzony jest po wywołaniu `SQLite3::Prepare`.
Najważniejsze metody:

- `bindParam` – przypisuje parametr do zmiennej wiążącej,
- `bindValue` – przypisuje wartość parametru do zmiennej wiążącej,
- `clear` – czyści wszystkie aktualnie przypisane parametry,
- close – zamyka przygotowane zapytanie,
- `execute` – wykonuje zapytanie,
- `paramCount` – zwraca liczbę parametrów,
- `reset` – zeruje zapytanie.

SQLite3 w PHP – przykład z SQLite3Stmt

```
$db = new SQLite3('../demo.db');

$query = 'SELECT * FROM pracownicy WHERE nazwisko=:nazwisko';
$stmt = $db->prepare($query);
$stmt->bindValue(':nazwisko', $argv[1], SQLITE3_TEXT);

$result = $stmt->execute();

while ($row = $result->fetchArray(SQLITE3_ASSOC)) {
  print_r($row);
}

$db->close();
```

SQLite3 w PHP – PDO, baza w pamięci

Błędy z danymi połączenia – należy przechwycić wyjątek.

```
try {
  $dbh = new PDO('sqlite::memory:', null, null);
  $dbh->query('CREATE TABLE FOO (a INTEGER)');
  $dbh->query('INSERT INTO FOO VALUES (1)');
  $dbh->query('INSERT INTO FOO VALUES (5)');
  foreach($dbh->query('SELECT * from FOO') as $row) {
    print_r($row);
  }
  $dbh = null;
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage() . "<br/>";
    die();
}
```

Wyjście:

```
Array ( [a] => 1 [0] => 1 ) Array ( [a] => 5 [0] => 5 )
```

SQLite3 w PHP – PDO, baza w pliku

```
try {
  $dbh = new PDO('sqlite:../demo.db',
                 null, null);
  $dbh->query('CREATE TABLE FOO (a INTEGER)');
  print_r($dbh->errorCode());
  print_r($dbh->errorInfo());
  $dbh->query('INSERT INTO FOO VALUES (1)');
  $dbh->query('INSERT INTO FOO VALUES (5)');
  foreach($dbh->query('SELECT * from FOO') as $row) {
    print_r($row);
  }
  $dbh = null;
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage() . "<br/>";
    die();
}
```

Rezultat

Za pierwszym razem:

```
00000
Array ( [0] => 00000 )
Array ( [a] => 1 [0] => 1 )
Array ( [a] => 5 [0] => 5 )
```

Za drugim razem:

```
HY000
Array ( [0] => HY000 [1] => 1
        [2] => table FOO already exists )
Array ( [a] => 1 [0] => 1 )
Array ( [a] => 5 [0] => 5 )
Array ( [a] => 1 [0] => 1 )
Array ( [a] => 5 [0] => 5 )
```

Plan wykładu

Podstawy programowania na iOS

- „Natywnym” językiem programowania dla platformy jest język **Objective-C**.
- Język jest nadzbiorem ANSI C i zachowuje podobną do C strukturę plików.
- Główne różnice pojawiają się w nazwach plików z kodem źródłowym oraz w składni używanej do definicji klas i odwoływania się do obiektów.
- Wprowadzenie: Learning Objective-C: A Primer.
- Każda aplikacja ma swój katalog z dokumentami – w nim można przechowywać m.in. bazy SQLite.

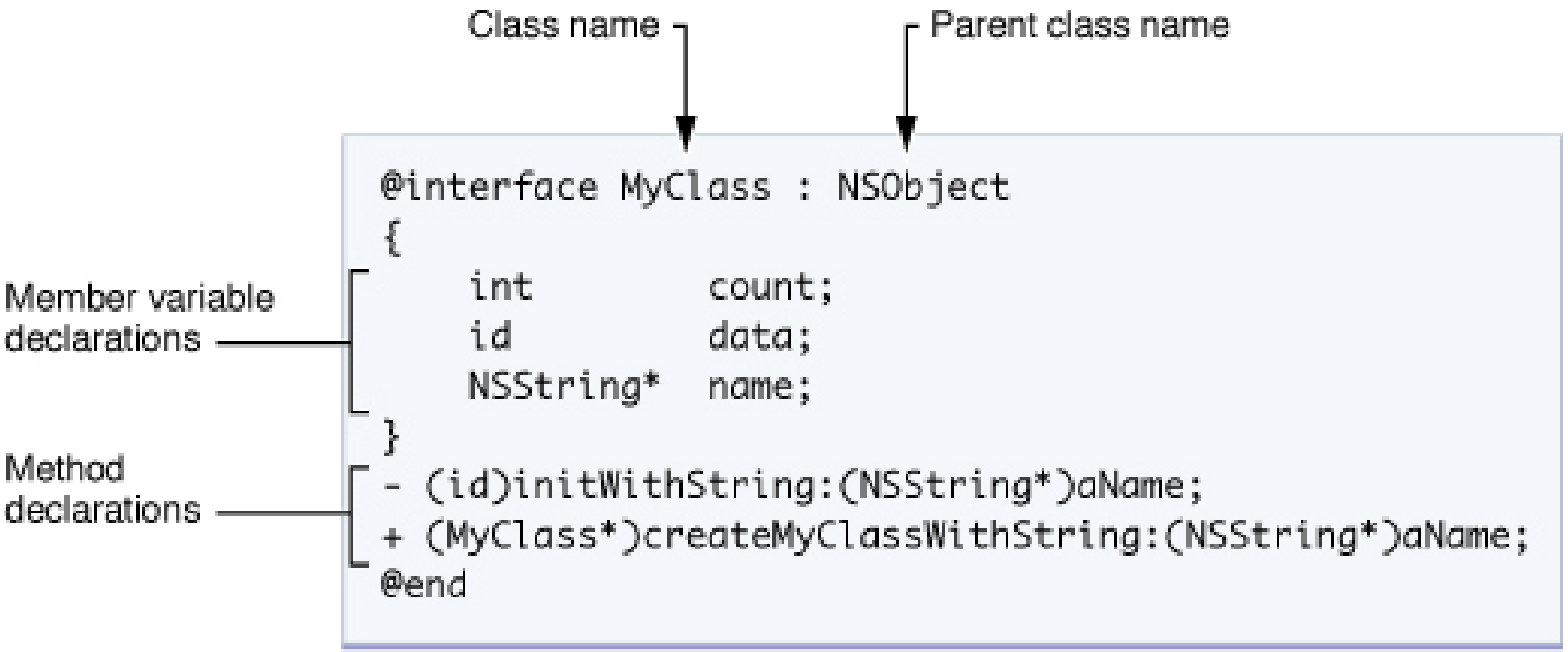
Struktura kodu Objective-C

Pliki źródłowe Objective-C mają następujące rozszerzenia:

- `.h` – pliki nagłówkowe, zawierają deklaracje klas, typów, funkcji i stałych,
- `.m` – pliki źródłowe, mogą zawierać kod C oraz Objective-C,
- `.mm` – j.w., ale mogą zawierać też kod C++.

Deklarowanie klas w Objective-C

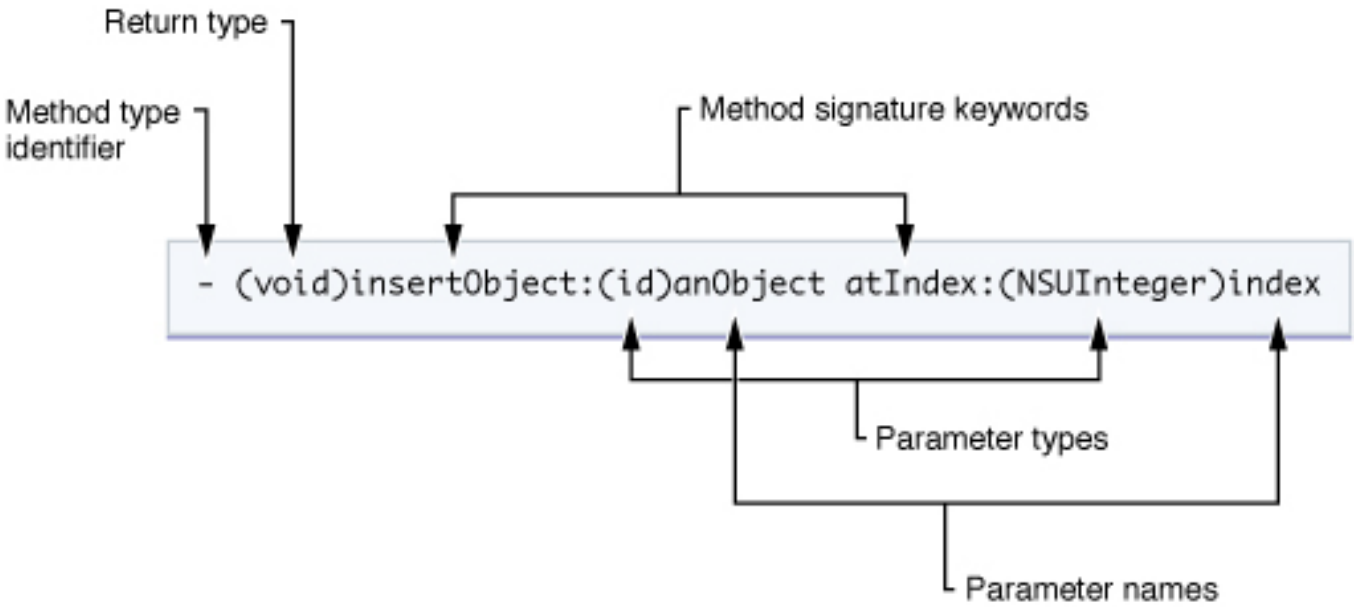
Objective-C umożliwia definiowanie składni w oparciu o następującą składnię:



Źródło: Apple Inc., *Learning Objective-C: A Primer*

Deklarowanie metod w Objective-C

Metody deklaruje się przy pomocy następującej składni:



Źródło: Apple Inc., *Learning Objective-C: A Primer*

Przykład wywołania powyższej metody:

```
[myArray insertObject:anObject atIndex:0];
```

Implementacja klasy

Przykład implementacji wcześniej zadeklarowanej klasy:

```
@implementation MyClass

- (id)initWithString:(NSString *)aName
{
    self = [super init];
    if (self) {
        name = [aName copy];
    }
    return self;
}

+ (MyClass *)createMyClassWithString: (NSString *)aName
{
    return [[[self alloc] initWithString:aName] autorelease];
}
@end
```

SQLite w iOS

- W aplikacjach iOS można łatwo skorzystać z SQLite, korzystając z pokazanego wcześniej API C/C++.
- Integracja SQLite w aplikacji jest prosta i wymaga:
  1. Dodania biblioteki SQLite3 (`libsqlite3.dylib`) na etapie konsolidacji (linkowania).
  2. Dodania pliku nagłówkowego `sqlite3.h`.

SQLite w iOS: proste operacje

Wskaźnik deklarujemy analogicznie jak w „zwykłym” C:

```
sqlite3 *contactDB;
```

Podobnie również otwieramy bazę:

```
int sqlite3_open(const char *filename, sqlite3 **database);
```

Funkcja `sqlite3_open` spodziewa się ciągu znaków UTF-8 (a nie obiektu `NSString`), trzeba więc dokonać konweresji:

```
sqlite3 *contactDB;

const char *dbpath = [databasePath UTF8String];

if (sqlite3_open(dbpath, &contactDB) == SQLITE_OK)
{
// sukces
} else {
// brak sukcesu...
}
```

SQLite w iOS: wykonanie zapytania

Przygotowanie zapytania:

```
sqlite3_stmt *statement;

NSString *querySQL = @"SELECT address, phone FROM contacts";

const char *query_stmt = [querySQL UTF8String];

if (sqlite3_prepare_v2(contactDB, query_stmt, -1,
        &statement, NULL) == SQLITE_OK)
{ ...
} else { ...
}
```

Wykonanie zapytania:

```
sqlite3_step(statement);
sqlite3_finalize(statement);
```

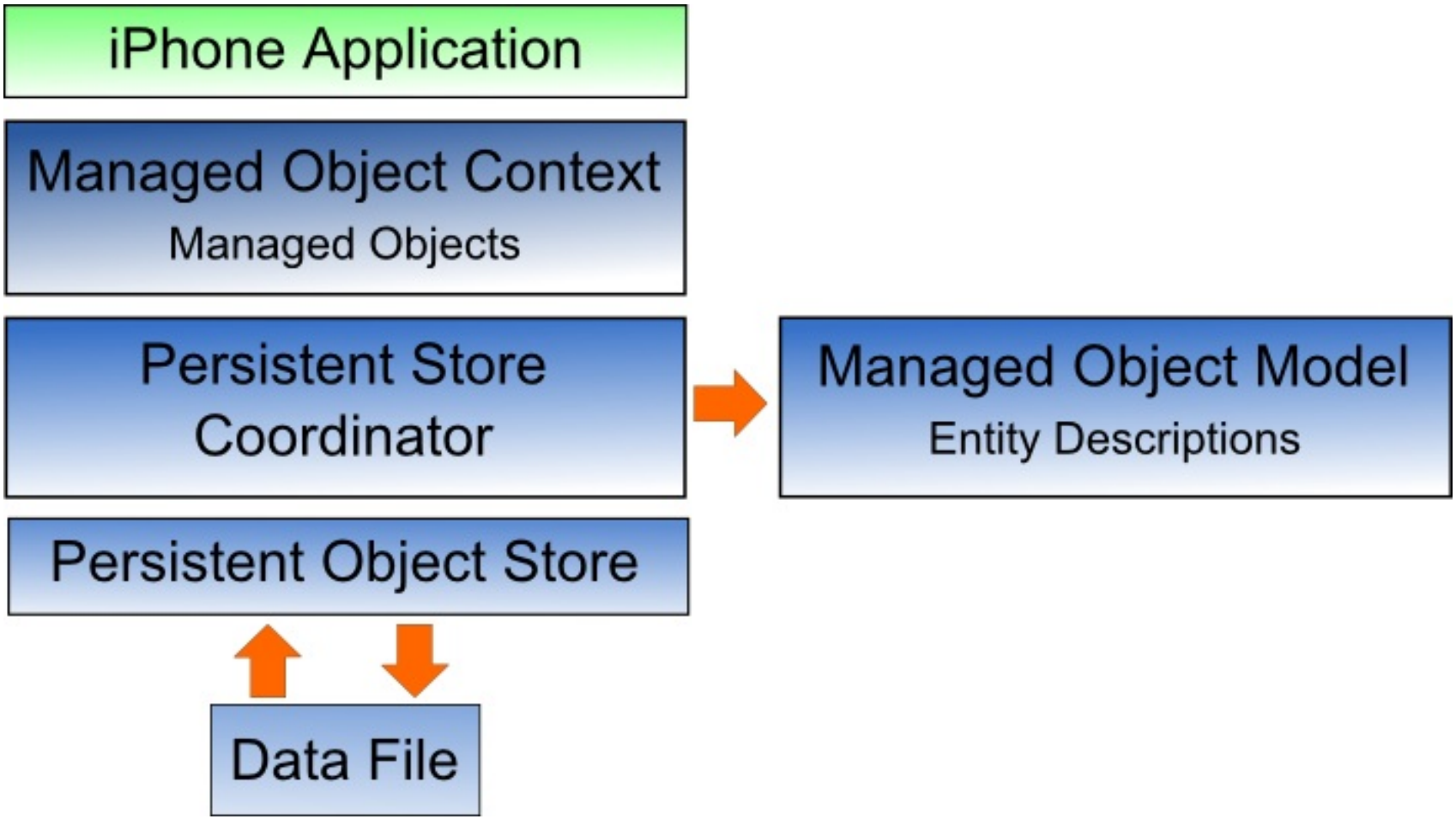
SQLite w iOS: Demo

Demo...

Core Data

- Bezpośrednie wykorzystanie bibliotek C/C++ SQLite3 to jeden ze sposobów na zarządzanie bazą danych przez aplikację iOS.
- Podejście dobre w wielu przypadkach, ale:
  - wymaga znajomości SQL,
  - wykorzystuje nie-obiektową biblitoekę,
  - zarządzanie strukturą bazy danych SQLite jest niewygodne.
- Odpowiedzią na te niedociągnięcia jest framework **Core Data**.

Core Data: architektura



źródło: Techtopia, *iPhone iOS 5 Development Essentials*

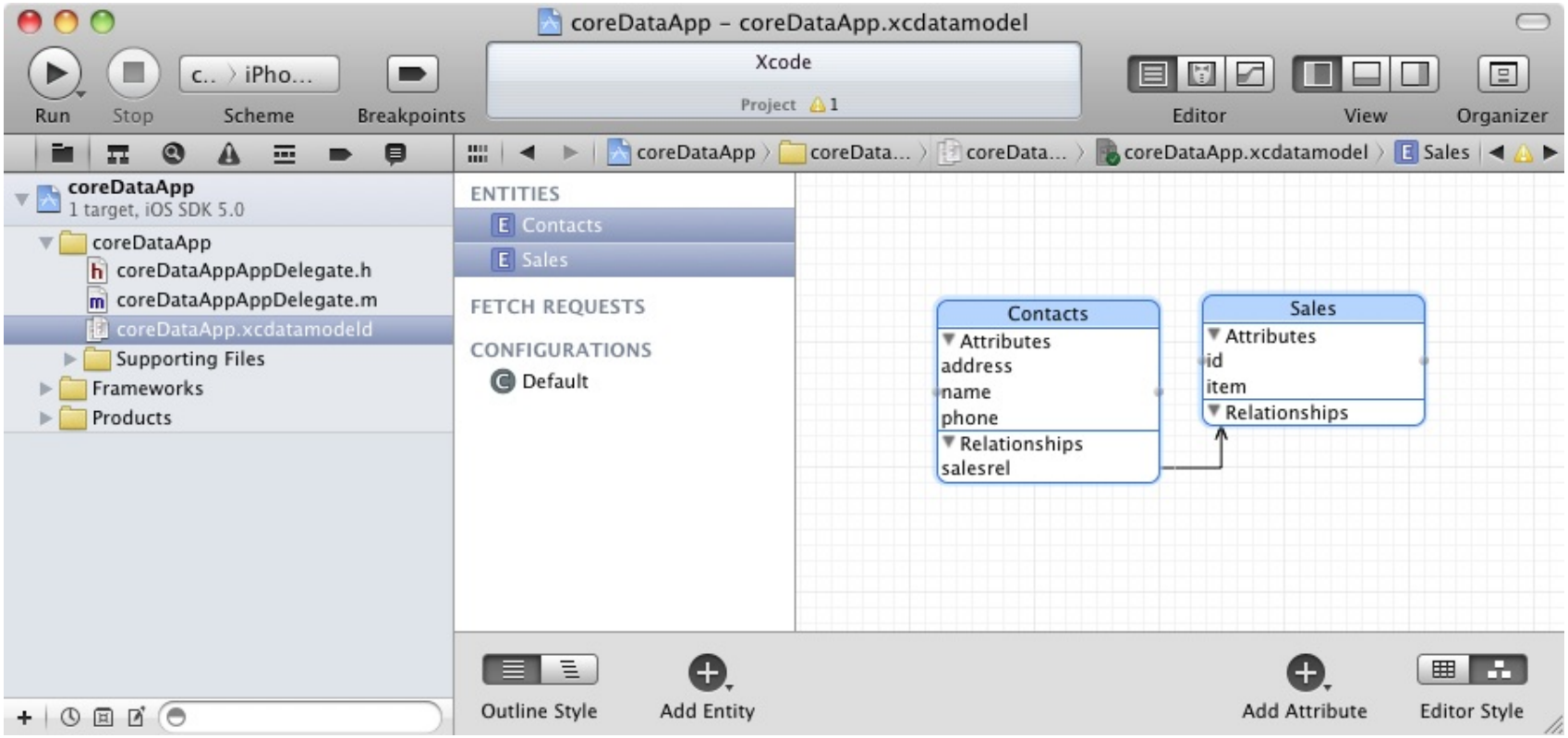
Core Data: koncepcje I

- **Obiekt zarządzany (managed object):** obiekt przechowujący dane, jak wiersz w tabeli bazy relacyjnej,
- **Kontekst obiektu zarządzanego (managed object context):** przechowuje stan obiektów w odniesieniu do fizycznego magazynku danych,
- **Model obiektów zarządzanych (managed object model):** model przechowywania danych; oprócz *atrybutów*, *encje* mogą posiadać również:
  - *relacje* – 1:1, 1:N; N:M,
  - *własności pobierane (fetched properties)* – własności jednego obiektu mogą być odczytywane przez inny obiekt; jak relacje, ale „słabsze”,

Core Data: koncepcje II

  - *żądania pobierania (fetch requests)* – predefiniowane zapytania służące do pobierania danych (np. wszystkie kontakty o nazwisku „Kowalki”).
- **Koordynator magazynu trwałego (persistent store coordinator):** odpowiada za skoordynowanie dostępu do wielu magazynów trwałych,
- **trwały magazyn obiektów (persistent object store):** fizyczny mechanizm utrwalania danych; do wyboru mamy XML, SQLite oraz magazyn binarny.

Core Data: definiowanie encji



źródło: Techtopia, *iPhone iOS 5 Development Essentials*

Core Data: przykłady I

Pobieranie kontekstu obiektu zarządzanego

```
coreDataAppDelegate *appDelegate =
  [[UIApplication sharedApplication] delegate];
NSManagedObjectContext *context =
  [appDelegate managedObjectContext];
```

Pobieranie opisu encji

```
NSEntityDescription *entityDesc = [NSEntityDescription
    entityForName:@"Contacts"
    inManagedObjectContext:context];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:entityDesc];
```

Core Data: przykłady II

Tworzenie obiektu zarządzanego

```
NSManagedObject *newContact;
newContact = [NSEntityDescription
      insertNewObjectForEntityForName:@"Contacts"
      inManagedObjectContext:context];
NSError *error;
[context save:&error];
```

Pobieranie i ustawianie atrybutów obiektu

```
[newContact setValue: @ John  Smith  forKey:@"name"];
[newContact setValue: @ 123  The  Street  forKey:@"address"];
[newContact setValue: @ 555 -123-1234    forKey:@"phone"];
NSString *contactname = [newContact valueForKey:  @  name  ];
```

Core Data: przykłady III

Pobieranie obiektów zarządzanych

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:entityDesc];
NSError *error;
NSArray *matching_objects =
    [context executeFetchRequest:request error:&error];
```

Core Data: przykłady IV

Wyszukiwanie obiektów zarządzanych

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];

[request setEntity:entityDesc];
NSPredicate *pred =
  [NSPredicate predicateWithFormat:@"(name = %@)",   John   Sm
[request setPredicate:pred];
NSError *error;
NSArray *matching_objects =
  [context executeFetchRequest:request error:&error]; ;
```

Plan wykładu

Sposoby przechowywania danych

W systemie Android wyróżniamy dwa podstawowe sposoby przechowywania treści:

- bazy danych SQLite – do przechowywania strukturalizowanych, zarządzanych danych,
- dostawcy treści (*content providers*) – generyczny interfejs do wykorzystania i współdzielenia danych.

Dostęp do danych

- Domyślnie, dostęp do danych ograniczony jest do aplikacji do której należą.
- Dostawcy treści udostępniają standardowy interfejs pozwalający na współdzielenie danych z innymi aplikacjami.

SQLite w Android

- Dostępna jest standardowa funkcjonalność bazy SQLite.
- Domyślnia ścieżka: `/data/data/<package_name>/databases`
- Zapytania zwracają kursory (obiekty `Cursor`).
- Dobrą praktyką jest utworzenie pomocniczej klasy, pośredniczącej w interakcji z bazą.
- Pakiety: `android.database`, `android.database.sqlite`.

Klasa SQLiteOpenHelper

- Po tej klasie zazwyczaj dziedziczy nasza klasa „pomocnicza”.
- Wymagane jest zdefiniowanie metod:
  - `onCreate()` – wołana przez framework jeżeli baza nie istnieje,
  - `onUpgrade()` – wołana, jeżeli wersja bazy danych jest zwiększona w kodzie aplikacji.
- Dobrą praktyką jest tworzenie osobnych klas dla każdej tabeli.

Klasa SQLiteDatabase

- Podstawowa klasa do pracy z SQLite w systemie Android.
- Udostępnia metody do otwierania i zamykania bazy oraz wykonywania zapytań.
- Udostępnia też metodę `execSQL()` do bezpośredniego wykonywania zapytań.
- Zapytania mogą być wykonywane przy pomocy metod `rawQuery()` oraz `query()`.

rawQuery() i query()

Przykład rawQuery()

```
Cursor cursor = getReadableDatabase().
  rawQuery("select * from todo where _id = ?",
  new String[] { id });
```

Przykład query()

```
return database.query(DATABASE_TABLE,
  new String[] { KEY_ROWID, KEY_CATEGORY,
    KEY_SUMMARY, KEY_DESCRIPTION },
  null, null, null, null, null);
```

Dostawcy treści

- Baza SQLite jest prywatna dla aplikacji, która ją utworzyła
- Aby współdzielić dane można wykorzystać dostawców treści (`ContentProvider`).
- Dostawca treści zazwyczaj jest używany jako interfejs do danych z bazy SQLite.
- Dostawca treści może być używany przez samą aplikację, lub do udostępnienia danych innym aplikacjom.
- Dostęp do dostawcy treści następuje poprzez URI.

Definicja dostawcy treści

Dostawców treści definiujemy w `AndroidManifest.xml`:

```
<provider
        android:authorities="de.vogella.android.todos.contentpr
        android:name=".contentprovider.MyTodoContentProvider" >
</provider>
```

Dostawca treści powinien obsługiwać m.in. metody `query`, `insert`, `update`, `delete`, `getType` i `onCreate`. W przeciwnym wypadku należy rzucić `UnsupportedOperationException`.

Dostawcy treści – bezpieczeństwo i współbieżność

- Domyślnie, `ContentProvider` będzie dostępny dla innych aplikacji. Aby wykorzystać go tylko prywatnie, należy dodać `android:exported=false` do definicji.
- Mogą pojawić się problemy ze współbieżnością, więc dostęp powinien być implementowany w sposób bezpieczny dla wątków (*thread-safe*).

Android, SQLite i dostawcy treści – więcej informacji

Więcej informacji:

- L. Vogel, Android SQLite Database and ContentProvider
- wersja na Kindle:
  `http://www.amazon.com/dp/B006YUWEFE`