University of Washington
iOS Development

Ted Neward
Neward & Associates
http://www.tedneward.com | ted@tedneward.com

12 Nov 2015

Data

Goals

Today we're going to...

... talk about data on iOS

- user defaults
- property lists
- object archives
- SQLite
- CoreData

User Defaults

Setting user preferences

User Defaults

Applications often need to store "settings"

- these are data elements about how the app functions
- usually too few, too small or too simple to need a database
- prefer to hold them locally
- fast access required

User Defaults

These user settings are displayed in the Settings app

- consistent user interface
- pre-developed UI and functionality
- requires firing up a separate app

User Defaults

Settings

- collection of key/value pairs
- values can be one of several types:
  - Boolean
  - Data
  - Date
  - Number
  - String
  - Arrays
  - Dictionaries
- requires specific format
- fortunately, "Settings Bundles" do that
  - edit Root.plist
  - this is a property list

User Defaults

Settings

- Root.plist is the Settings property list
- collection of name/value pairs interpreted by Settings app
- conventions are important here
  - "type": what this item is
    - for example, "Group", which is a header
    - "Multi-Value" supports a range of choices
  - "title": name to display
  - "identifier": programmatic key for access
  - "text field is secure": password-style input
  - autocorrect, autocapitalize
  - ... and a few others

User Defaults

NSUserDefaults

- access instance from standardUserDefaults()
- read using "X"forKey()

  where "X" is bool, float, int, etc
- set using set"X"

  again, "X" is Bool, Int, Float, etc
- key passed must match "identifier" in plist
- synchronize() to sync settings with disk

  expensive operation (writes to disk)

User Defaults

Quick way to open Settings from your app

- a URL of scheme "app-settings:" opens the Settings app
- so call UIApplication.sharedApplication().openURL()

  passing NSURL(string:UIApplicationOpenSettingsURLString)!)

iOS App Filesystem Basics

Poking your head around a little

Filesystem

Each application gets its own directory

- this is your application's "sandbox"

  no other app has access to it

- sandbox has several directories of interest

  - **/Documents: read/write access; synced to iTunes**
  - **/Library: files not shared with the user**
  - **/tmp: temp directory space**

    not automatically cleaned, beware!

Filesystem

- Get hold of directories via NSSearthPathDirectory
  - holds a constant for the DocumentDirectory
  - use NSSearchPathDomainMask.UserDomainMask

    sort of a holdover from OSX
- Get hold of tmp directory with NSTemporaryDirectory()
  - returns /tmp directory
- Use NSFileManager for various utility methods
  - fileExistsAtPath(), for example

Property Lists

Simple key/value data in iOS

Property Lists

Common form of simple storage in iOS/OSX

- also known as "plists"
- frequently used to store resource data from XCode
  - often stored in "resource bundles"
- stored in one of three formats
  - •binary
  - •old-school ASCI OpenStep-inherited format
  - •XML (preferred)

Property Lists

Property lists: arrays/dictionaries of data

- no specific "property list" type
- only the following can be plist'ed
  - arrays
  - dictionaries
  - strings
  - numbers
  - dates
- anything else will generate an error

  meaning, no custom data types

Property Lists

- Reading/Writing property lists
  - NSPropertyListSerialization to store/read

    allows specifying the format
- Writing property lists
  - writePropertyList:toStream:format:options:

    serializes to a specified output stream
  - dataWithPropertyList:format:options:

    serializes to an NSData (in-memory format)
- Reading property lists
  - propertyListWithData:options:format:

    deserializes from an NSData
  - propertyListWithStream:options:format:

    deserializes from an input stream

Object Archives

Serialization in iOS

Serialization

NSArrays and NSDictionaries can write themselves to disk

- effectively they become really lightweight databases
- depending on the format (XML), they also are portable

  meaning, some other platform/language could read them
- if they are to serialize successfully, store only:
  - **arrays: Swift array/NSArray or NSMutableArray**
  - **dictionaries: Swift dictionary/NSDictionary or NSMutableDictionary**
  - **NSData, NSMutableData**
  - **Swift string/NSString or NSMutableString**
  - **NSNumber**
  - **NSDate**

Serialization

NSArray/NSDictionary

- write: call writeToFile:atomically

  atomically: either/or success/fail, no intermediate state
- read: use initializer expecting "contentsOfFile" parameter

  atomic by nature
- these will store as property lists by default

Serialization

Custom Objects

- it's possible to "extend" serialization to custom types
- this is the role of the NSCoding protocol

  indiciates participation and how to serialize
- also a good idea to conform to NSCopying protocol

  this allows for easy "cloning" of objects

Serialization

NSCoding protocol

- encodeWithCoder: do the encoding
    - utilizing the passed-in NSCoder object
    - "encodeX" where X is one of a variety of types
    - if you inherit, make sure to call up the chain
- required initializer taking an NSCoder parameter
    - again utilizes the NSCoder
    - "decodeX" where X is one of a variety of types
    - also call up (AFTER decoding) the chain

Serialization

NSCopying protocol

- copyWithZone()
    - performs a "deep copy" of self
    - ignore the NSZone parameter (ancient history)

Serialization

NSKeyedArchiver

- coordinates the serialization
- wraps around an NSData or file
- writeToFile:atomically writes to file
- archiveDataWithRootObject returns an NSData

NSKeyedUnarchiver

- coordinates the deserialization
- wraps around an NSData or file
- use NSData(contentsOfFile) initializer

SQLite

What it is, what it isn't

SQLite

SQLite is a small-scale footprint RDBMS

- ".. a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine."

  http://www.sqlite.org
- Mostly SQL-92 compliant
- ACID Transactions supported
- Stored as a single file stored locally
- Self-contained
- Available as a single C file if necessary

The SQLite SQL dialect

What's it got?

SQLite SQL

(Almost) Full SQL-92 compliance

- classic CRUD ops (SELECT, INSERT, DELETE, UPDATE)
- classic DDL ops (CREATE, ALTER, etc)
- numerous functions (aggregates, date/times, etc)
- transaction support
- indexes, views, triggers, virtual tables

SQLite SQL

SQLite offers this as its list of missing features:

- RIGHT and FULL OUTER JOIN

  LEFT OUTER JOIN is implemented, but not RIGHT OUTER JOIN or FULL OUTER JOIN.

SQLite SQL

SQLite offers this as its list of missing features:

- Complete ALTER TABLE support

  Only the RENAME TABLE and ADD COLUMN variants of the ALTER TABLE command are supported. Other kinds of ALTER TABLE operations such as DROP COLUMN, ALTER COLUMN, ADD CONSTRAINT, and so forth are omitted.

SQLite SQL

SQLite offers this as its list of missing features:

- Complete trigger support

  FOR EACH ROW triggers are supported but not FOR EACH STATEMENT triggers.

SQLite SQL

SQLite offers this as its list of missing features:

- Writing to VIEWs

  VIEWs in SQLite are read-only. You may not execute a DELETE, INSERT, or UPDATE statement on a view. But you can create a trigger that fires on an attempt to DELETE, INSERT, or UPDATE a view and do what you need in the body of the trigger.

SQLite SQL

SQLite offers this as its list of missing features:

- GRANT and REVOKE

  Since SQLite reads and writes an ordinary disk file, the only access permissions that can be applied are the normal file access permissions of the underlying operating system. The GRANT and REVOKE commands commonly found on client/server RDBMSes are not implemented because they would be meaningless for an embedded database engine.

SQLite API

Accessing SQLite programmatically

SQLite API

C-based API

- complete with pointers
- "object-oriented C"
  - **key types being "sqlite3*" and "sqlite_stmt*"**
- all methods will be "sqlite3_"-prefixed
- error objects passed in by reference from caller
- callback (function pointer) specified for results

SQLite API

Some key SQLite API calls

- sqlite3_open(const char* filename, sqlite3** ppdb)

  opens or creates the SQLite database

- sqlite3_close(sqlite3* db)

  - closes the database
  - will fail (SQL_BUSY) if database is not ready to be closed

SQLite API

Some key SQLite API calls

- sqlite3_exec(sqlite3*, const char* sql, (callback), void* arg, char** errmsg)
    - sqlite3* is the database object
    - sql
    - callback is a "int (*)(void*, int, char**, char**)" function
    - arg is the firt parameter to the callback
    - errmsg is the error message allocated by sqlite3
    - make sure to sqlite3_free() the error message (if there is one)

SQLite API

Some key SQLite API calls

- sqlite3_exec is actually a wrapper around...
  - sqlite3_prepare_v2()
  - sqlite3_step()
  - sqlite3_finalize()
- ... executed all in one fell swoop

SQLite on iOS

Using SQLite in iOS apps

Using SQLite on iOS

SQLite remains a popular RDBMS for iOS

- but it's very low-level
- requires Swift/C integration

Using SQLite on iOS

Opening a SQLite database

```
var database: COpaquePointer = nil
let result = sqlite3_open("/databases/file", &database)

// ...

let createSQL = "CREATE TABLE IF NOT EXISTS PEOPLE" +
          "(ID INTEGER PRIMARY KEY AUTOINCREMENT, " +
          "FIELD_DATA TEXT)"
var errMsg: UnsafeMutablePointer<Int8> = nil
let result = sqlite3_exec(database, createSQL, nil, nil, &errMsg)
```

Using SQLite on iOS

Must link the SQLite 3 library

- "libsqlite3.dylib"
- right-click root node in Project Navigator
- select Build Phases
- expand Link Binary With Libraries
- add ("+") and find libsqlite3.tbd

Using SQLite on iOS

Must import SQLite3 declarations

- create a "bridging header" to import sqlite3.h
- look under Build Settings for "Bridging Header"
- if none is specified, create one
  - convention says it is {{ProjectName}}-BridgingHeader.h
  - now create the header file itself
  - inside the contents, add "import <sqlite3.h">"

Core Data

Doing data access the iOS way

Core Data

Core Data is the iOS Object/Relational Mapper (ORM)

- this opens up a whole can of worms
- very easy way to avoid SQL and focus on objects
- maps well with SQLite3
- lots of XCode IDE assistance in building out the models

Final App

Homework: Prepare your application pitches

- you will form small (3-5 person) groups in order to build a non-trivial iOS application as your final project

Final App

Goals:

- something remotely useful

  no "kitchen sink UI apps", for example
- 5 to 10 activities/fragments

  major "screens" the user sees/displays
- some kind of communication component

  either HTTP/XML or HTTP/JSON to/from a server
  and/or ad-hoc P2P WiFi
  and/or phone or SMS

Final App

Rules:

- groups of 3-5 persons
- any outside collaboration must be documented

  if you have somebody else design your UI or create art
- principal use of the iOS SDK

  don't go build a game using Unity, for example

FAQs

- yes, work on your Capstone is OK

  but you will be expected to be at the upper end of the work range
- yes, you can use mocked-up data

  but it should be delivered from a remote location (where reasonable)
- yes, you can use third-party SDKs

  but they shouldn't be the focus of your application

Final App

Deliverables (immediately):

- group info (members, team name, etc)

Final App

Deliverables (on pitch day):

- group info (members, team name, etc): immediately!
- pitch deck (PDF): before day of pitch!
  - **MUST include list of "core user stories" that are committing to**
  - **MAY include list of "optional user stories" that you'd like to hit**
  - **YOU WILL BE GRADED ON HOW WELL YOU HIT THE CORE USER STORIES**

    put some deep thought into your estimates!

Final App

Deliverables (on delivery day):

- TWO devices that are running your app

  one for you to use when demoing, one for the judges to look at
- "summation deck": pitch deck AS-IS, plus your experience/results
- URL to the Github repo containing your code
- email to me (privately) about your experience working with your teammates

  work breakdown, one thing you liked, one thing you'd want improved

Final App

Grading:

- 25% on your pitch

  are you clear in what you're proposing?
  do you have clear user stories established?
  NOT grading on the utility or business value of the app; if you want to build the world's most amazing fart app, that's fine, so long as it meets the criteria laid out above
- 60% on the final demo in three weeks

  does it do what you proposed?
  did you get all the user stories complete?
  does it run without crashing?
- 15% on group self-assessment

  you will evaluate your peers; I will assign points accordingly
  this is designed to help ensure everybody works
  this is also designed to get you used to workplace peer reviews