University of Regina

CS 455/855
Mobile Computing

Persistent Data

Dr. Orland Hoeber
orland.hoeber@uregina.ca
http://www.cs.uregina.ca/~hoeber/cs455/2018F

Readings

- iOS Developer Library
  - File System Programming Guide (only the iOS bits)
  - Archives and Serializations Programming Guide
- SQLite.swift Library Documentation

- Others as needed:
  - Core Data Programming Guide
  - UIDocument Class Reference
  - iCloud Design Guide
  - Firebase

Persistent Data

- Managing persistent data is different in the mobile environment than in most other programming settings
  - limited storage
  - sandboxing
  - files primarily accessed by the sole app that created them
  - difficult to inspect and debug
- With the iOS API
  - may allow iTunes to access the files
  - may access files/databases over the cloud (iCloud, Dropbox, Firebase etc.)
  - may store data using low-level file system commands, or intermediate data management objects such as UIDocument or Core Data

Big Design Decision

- When designing for persistent storage, you have a big design decision to make:
  - How should we save our data?
    - simple text file
    - binary data file
    - documents
    - local database (SQLite)
    - remote database (MySQL via API, Firebase Database, etc.)
    - cloud-based storage (Dropbox, S3, Firebase Cloud Storage, etc.)
  - When should we save our data?
- Each decision has pros and cons that must be considered

Low-Level File Management

- Each app operates in its own sandbox
  - portion of the file system that is reserved exclusively for the app
  - an app cannot look at the files of another app
    - if you need to share files across apps or to the outside world, you will want to use one of the high-level data or document management frameworks
  - users will not have direct access to this file system
  - built-in directories for each app:
    - Documents
    - Library
    - tmp

Guidelines for Where to Put Data

- There are some guidelines for where to put data:
  - <Application Home>/Documents/
    - user-generated content and data
  - <Application Home>/Documents/Inbox/
    - read-only access to files that your app was asked to open by outside entities (i.e., mail app)
  - <Application Home>/Library/
    - location for application support files that are not user data
  - <Application Home>/Library/Caches/
    - downloaded data
  - <Application Home>/tmp/
    - temporary data
- By obeying these guidelines, we can write code in a generic way for managing low memory situations (e.g., empty the /tmp and /Library/Caches/ directories)

Specifying a Path to a File

- While you can specify a path to a file using a String object, the preferred way is to use an URL object
- Why?
  - if you specify the path in a string format, it is easy to make a mistake
  - the URL object has a large number of helper functions that will ensure that you build the path correctly
- Note that a URL is not just for specifying web-based resources (remember what the U stands for?)

Code for Specifying a File-Based URL

```
let docsDir = FileManager().urls(
        for: .documentDirectory,
        in: .userDomainMask).first!
let fileURL = docsDir.appendingPathComponent("meals")
```

- ".urls" produces an array, use ".first" to get the first item
- "for:" allows us to specify the common directory type
  - enumerated type with many options for common locations of files (documentDirectory, cachesDirectory, libraryDirectory, etc.)
- "in:" allows us to specify which file system domain to search
  - for iOS programming, this will always be .userDomainMask because each app is in its own sandbox, and doesn't have broader filesystem access
- the second statement appends "meals" to the end of the URL, making this a file name

Saving String Data to a File

- The easiest way to write/read data from a file is to use one of the following methods of the String object:
  - write(to: URL, atomically: Bool, encoding: .Encoding)
  - init(contentsOf: URL, encoding: .Encoding)

- These methods also exist for Array and Dictionary objects that contain String objects

- If your data is more complex than just strings, then you need to do more work to write it to a file

Archive Files

- When your model objects are more than just a string, another way to save data in your app is to construct an Archive
  - binary file format
  - encodes the contents of an object and saves this as a file
  - in reverse, it decodes the file and uses this to initialize an object
- Pros:
  - easy to save and restore the state of your model objects
- Cons:
  - if not designed well, there can be lots of overhead for saving incremental changes

Saving Data from Custom Objects

- The general process for saving data from custom objects (model objects) to a file is to:
  - make the class adopt the NSCoding protocol
  - for writing:
    - Use the NSKeyedArchiver object to convert the object to an NSData type, and write it to the file (URL)
  - for reading
    - Use the NSKeyedUnarchiver object to read the file (URL) and convert it back to it’s original type

Conform to NSCoding Protocol

```
class Person: NSObject, NSCoding {
        var name: String
        var date: Date
   …

   func encode (with aCoder: NSCoder) {
        aCoder.encode (name, forKey: "name")
        aCoder.encode (date, forKey: "date")
   }

   required convenience init? (coder aDecoder: NSCoder) {
        let nameData = aDecoder.decodeObject(forKey: "name") as? String
        let dateData = aDecoder.decodeObject(forKey:"date") as? Date
        self.init(name: nameData, date: dateData)
   }
}
```

Save & Load the Data

```
class Person: NSObject, NSCoding {
    …
          let docsDir = FileManager().urls(
                    for: .documentDirectory,
                    in: .userDomainMask).first!
          let fileURL = DocsDir.appendingPathComponent(”person")
    …

    func saveMe() {
          let isSuccessfulSave = NSKeyedArchiver.archiveRootObject(
                    self,
                    toFile: fileURL.path)
    }

    func loadMe() {
          self = NSKeyedUnarchiver.unarchiveObject(
                    withFile: fileURL.path)
    }
}
```

Some Challenges

- The NSKeyedArchiver and NSKeyedUnarchiver are expecting to be encoding Objects
  - some simple data types aren't encoded very easily
    - Bool
    - Enumerated data types
  - trick:
    - within the methods that you add to make your class conform to NSCoding (encode, init(coder)), you need to explicitly convert these to/from an encodable data type
      - String
      - Number

Keyed Archives

- Keyed Archives allow you to take control of storing data in binary files
  - allows you to specify unique keys to the archive file, representing specific objects that are encoded
  - without this, if you want the data for different objects to be stored and loaded separately, you'd have to create separate data files
  - using this method means that you don't have to write absolutely everything out to the file each time

Other Considerations

- There are a number of important considerations that must be made:
  - only one data file: load this in the init method
  - multiple data files: allow the user to select which file to load
    - generate directory listing
    - provide interface to select the file
  - when to save?
    - saving too often wastes resources
    - saving not often enough exposes the user to the risk of their data being lost

Other Persistent Storage Options

- NSUserDefaults
  - key-value pairs, used for default settings
- UIDocument
  - document management architecture
  - support for auto-save
  - thread-safe (data saved in the background)
  - compatible with iCloud
- Core Data
  - memory management (only load a subset of the data)
  - tools for data migration with new versions of your app
  - automatic load, save, undo, etc.
- SQLite
  - file-driven database
  - supports the common SQL commands (select, insert, update, delete)

SQLite

- SQLite is a public-domain, self-contained database management system that can be embedded in other software

- The value of SQLite is that it works very much like any other database
  - this means that we can pre-load data into it and distribute this with our app
  - the app can then manipulate this data using well-known insert, update, and delete commands
  - allows for small updates without having to write everything out to the file

- The problem is that it is written in C, which makes using it in Swift oddly complex

3rd Party Library: SQLite.swift

- https://github.com/stephencelis/SQLite.swift

  - provides a Swift interface to SQLite
  - allows you to build your queries within the Swift data structures
  - query and parameter binding interface
  - proper error-handling
  - well-documented:
    - https://github.com/stephencelis/SQLite.swift/blob/master/Documentation/Index.md#getting-started
  - but, there is some extra overhead in coding to ensure that the data is mapped to proper Swift data structures

Tips for using SQLite/SQLite.swift

- There are a number of different package managers that you could use, but for this class
- For a manual installation
  - explicitly link the project to the library in the following locations (under General tab of the project)
    - Linked Frameworks and Libraries
    - Embedded Binaries
- In the viewDidLoad method (or any other place where you do initialization), you should try to create the table structure if it doesn't already exist

Create the Table

```
do {
   // establish connection to database
   let docsDir = FileManager().urls(for: .documentDirectory, in: .userDomainMask).first!
   let db = try Connection("\(docsDir)/db.sqlite3")

   // table variable
   let users = Table ("users")
   // column variables
   let id = Expression<Int64>("id")
   let email = Expression<String>("email")
   let name = Expression<String?>("name")
   // create table
   try db.run (users.create (ifNotExists: true) { t in // CREATE TABLE "users" (
       t.column (id, primaryKey: .autoincrement)       // "id" INTEGER PRIMARY KEY
AUTO_INCREMENT,
       t.column (email, unique: true)                  // "email" TEXT UNIQUE NOT NULL,
       t.column (name)                                 // "name" TEXT )
   })
} catch {
   os_log("Database creation failure.", log: OSLog.default, type: .error)
}
```

Adding Columns

- When adding columns to a table (using the column method), there are a wide range of parameters that are used to build proper database tables:
  - `t.column(id, primaryKey: true)`
    - "id" INTEGER PRIMARY KEY NOT NULL
  - `t.column(id, primaryKey: .autoincrement)`
    - "id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL
  - `t.column(email, unique: true)`
    - "email" TEXT UNIQUE NOT NULL
  - `t.column(email, check: email.like("%@%"))`
    - "email" TEXT NOT NULL CHECK ("email" LIKE '%@%')
  - `t.column(name, defaultValue: "Anonymous")`
    - "name" TEXT DEFAULT 'Anonymous'

Insert New Item

```
do {
   // establish connection to database
   let docsDir = FileManager().urls(for: .documentDirectory, in: .userDomainMask).first!
   let db = try Connection("\(docsDir)/db.sqlite3")

   // create variables to use in the query
   let users = Table ("users")
   let email = Expression<String>("email")
   let name = Expression<String>(”name”)

   let rowid = try db.run(users.insert(email <- "hoeber@cs.uregina.ca", name <- “Orland”))
   print(”Insert success: row id = \(rowid)")
} catch {
   os_log(”Insert failure: %@", log: OSLog.default,
           type: .error, error.localizedDescription)
}
```

Update Item

- The update command works very similarly to the insert

```
try db.run(counter.update(count <- 0))
// UPDATE counters SET count = 0
```

- Note that there are a number of other operators beyond the assignment (<−) that you can use
  - ++ (increment)
  - −− (decrement)
  - += (add numbers; concatenate strings)
  - and many more

Iterating over the Data in a Table

```
do {
   // establish connection to database
   let docsDir = FileManager().urls(for: .documentDirectory, in: .userDomainMask).first!
   let db = try Connection("\(docsDir)/db.sqlite3")

   // create the variables to use in the query
   let users = Table ("users")
   let id = Expression<Int64>("id")
   let email = Expression<String>("email")
   let name = Expression<String?>("name")

   for user in try db.prepare(users) {       // SELECT * FROM "users”
       print("id: \(user[id]), email: \(user[email]), name: \(user[name])")
      // id: 1, email: alice@mac.com, name: Optional("Alice")
   }

} catch {
   os_log("Database query failure: %@", log: OSLog.default,
           type: .error, error.localizedDescription)
}
```

Building Complex Queries

- If you need to do more than just iterate over all of the data in a table, you can build up complex queries by stringing operators together:
  - select
  - join
  - filter
  - order
  - limit

- There are also operators for aggregation (count, max, min, total, etc.)

Query the Data

```
[in do-catch block]
   // establish connection to database
   let docsDir = FileManager().urls(for: .documentDirectory, in: .userDomainMask).first!
   let db = try Connection("\(docsDir)/db.sqlite3")

   // create the variables to use in the query
   let users = Table ("users")
   let id = Expression<Int64>("id")
   let email = Expression<String>("email")
   let name = Expression<String?>("name")

   let query = users.select(id, email, name)
                    .filter(name != nil)
                    .order(email.desc, name)
                    .limit(5)

   for user in try db.prepare(query) {
        print("id: \(user[id]), email: \(user[email]), name: \(user[name])")
   }
```

Handling DATETIME and BLOB

- Because of the complexity of matching the Date datatype in Swift to the DATETIME data type in SQLite, you have to do some extra work
  - map the Date object to the specific format you want in your database

- A similar issue exists for storing BLOBs (e.g., for storing images within the database)

- See the SQLite.swift documentation for sample code

Support for Transactions

- The SQLite.swift Library includes support for transactions
  - use the transaction and savepoint methods
  - if any single statement within the transaction fails, all changes in the block are rolled back

```
try db.transaction {
    let rowid = try db.run(users.insert(email <- "betty@icloud.com"))
    try db.run(users.insert(email <- "cathy@icloud.com", managerId <- rowid))
}
// BEGIN DEFERRED TRANSACTION
// INSERT INTO "users" ("email") VALUES ('betty@icloud.com')
// INSERT INTO "users" ("email", "manager_id") VALUES ('cathy@icloud.com', 2)
// COMMIT TRANSACTION
```

Firebase

- https://firebase.google.com

- Cloud-based data storage designed specifically for mobile app development
- Support for iOS, Android, JavaScript, and REST API
- Wide range of support features for app development
- Our particular interest for data storage are:
  - Cloud Storage
  - Realtime Database

Realtime Database

- Store and sync data across devices
  - JSON data
  - NoSQL database (like MongoDB, Cassandra, etc.)
  - Cloud-based (no need to maintain your server)
- Key mobile-centric features
  - Realtime – whenever the data changes, the update is pushed to other connected devices automatically
  - Offline mode – data is replicated on each device, allowing operation even if network connectivity is lost; when connectivity is restored, data is merged
  - Security & authentication – can control who has access to what
- Limitation
  - Will only allow operations that can be executed quickly (read, write); no complex queries
  - Store files (e.g., images) elsewhere (Firebase Cloud Storage), and save URL here

Homework

- Next week: Evaluation Methods
- Last things: Project Demos, Exam Review
- CS 855: Position Paper #2
  - due ~~Nov 23~~ Nov 26
- Demos
  - Dec 4/6
- Project Submission
  - due Dec 6