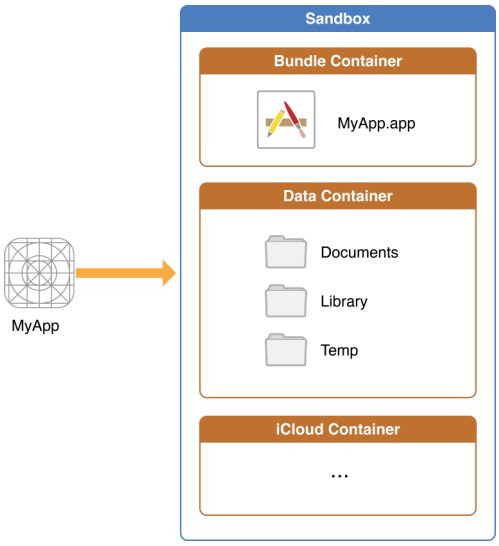
Chapter VIII

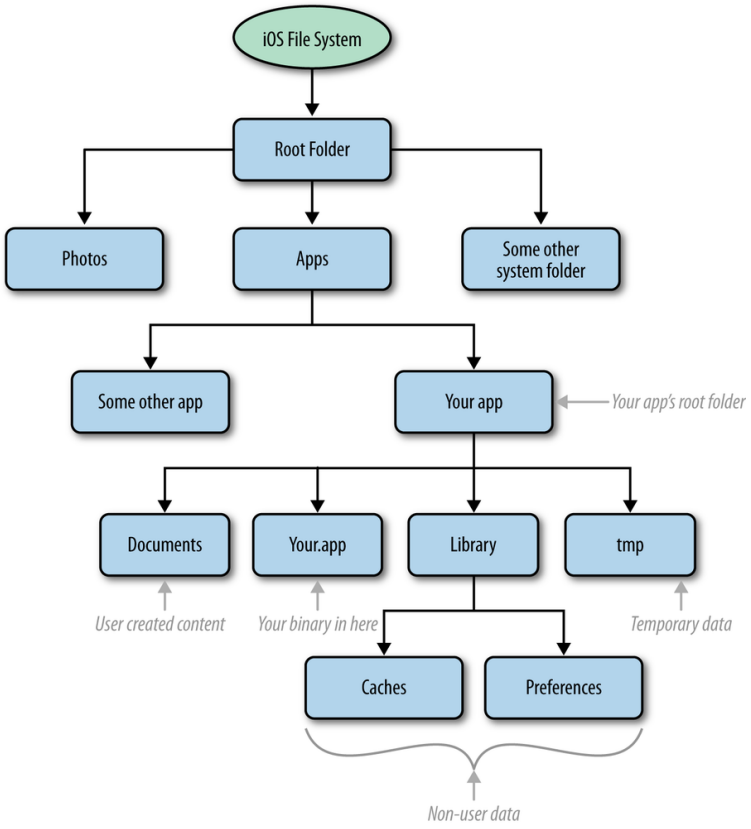
Persistence in iOS

VII.Persistance in iOS

1. iOS File System

The file system in iOS is based on the UNIX file system.The iOS file system is geared toward apps running on their own and an iOS app's interactions with the file system are limited mostly to the directories inside the app's sandbox.





1. Accessing Files and Directories

Typical File Structure

-AppName.app

-Documents/

-Library/

|-Caches

|-Application Support

|-Preferences

-tmp/

1. Using NSFileManager

Using NSFileManager object lets you examine the contents of the file system and make changes: to locate, create, copy, and move files and directories. When specifying the location of files, you can use either NSURL or NSString objects. The use of the NSURL class is generally preferred for specifying file-system items because they can convert path information to a more efficient representation internally.

eg. Using URLsForDirectory Methods of NSFileManager

```
var docPath =
FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)

var cachePath =
FileManager.default.urls(for: .cachesDirectory,
in: .userDomainMask)

var appsupportPath =
FileManager.default.urls(for: .applicationSupportDirectory,
in: .userDomainMask)
```

Note:Dot directories and files. Any file or directory whose name starts with a period (.) character is hidden automatically.When using a file manager object with a delegate, it is recommended that you create a unique instance of the NSFileManager class and use your delegate with that instance.

I. Useful File Operations

a) Moving and Copying Items

- copyItemAtURL:toURL:error:
- copyItemAtPath:toPath:error:
- moveItemAtURL:toURL:error:
- moveItemAtPath:toPath:error:

b) Creating and Deleting Items

-CreateDirectoryAtURL:withIntermediateDirectories:attributes:error:
-createDirectoryAtPath:withIntermediateDirectories:attributes:error:
-createFileAtPath:contents:attributes:-
-removeItemAtURL:error:
-removeItemAtPath:error:

c) Discovering Directory Contents

- contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:
- contentsOfDirectoryAtPath:error:

2. Using NSSearchPathForDirectoriesInDomains

eg .let documentsPath = **NSSearchPathForDirectoriesInDomains** (.DocumentDirectory, .UserDomainMask, true)[0]

Note: you can user Bundle.main.url(forResource or path(forResource for application resource files if you can't simply access the resource by its filename

eg.

```
//Reading String file
if let fileStr =
     Bundle.main.url(forResource: "myfile",
     withExtension: "txt") {
   do {
      let data =  try String.init(contentsOf: fileStr)
        print(data)
      } catch {
        print("Can't read data from file ")
      }
}
```

```
// Reading Binary file

 if let fileStr =
     Bundle.main.url(forResource: "forest",
                withExtension: "jpg") {
   do {
        let data =  try Data.init(contentsOf: fileStr)
        print(data)
        let image = UIImage(data: data)
        // You may use UIImage(contentsOf: fileStr)
        let imageView = UIImageView(image: image)
        imageView.frame =  view.bounds
        imageView.contentMode = .scaleAspectFill
        view.addSubview(imageView)
      } catch {
            print("Can't read data from file ")
      }
}
```

2. Using SQLite for Persistence

To use SQLite for Database Operation, following framework are necessary to import to project file:

-libsqlite3.0.tbd, libsqlite3.tbd, Foundation and SystemConfiguration framework

and, it is necessary using third party frame work, create an objective-C header file and set the header name in the target's Objective-C Bridging Header build setting and declare objective-c header file name in it.

a) **Useful SQL Commands:** Refer to SQL Note
b) **Useful simple and lightweight Third Party SQLite wrapper for Swift:**

https://github.com/FahimF/SQLiteDB
https://github.com/ccgus/fmdb

e.g Using SQLiteDB Wrapper to display the table contents from sqlite dictionary database file, dict.sqlite

```
//set Database name in SQLiteDB.swift with let DB_NAME = "dict.sqlite"
let db = SQLiteDB.sharedInstance
let data = db.query("SELECT * FROM dictionary ")
for row in data
{
        print((row["en"] as! String)+(row["mm"] as! String))
 }
```

eg.

3. Using Realm for Persistence

Realm is a cross-platform mobile database solution designed specifically for mobile applications.

It's fast, lightweight, and extremely simple to integrate in your project. Most common functions such as querying the database consist of a single line of code!

Note: The most common location to store writable Realm files is the "Documents" directory on iOS and the "Application Support" directory on macOS.

Ref: https://realm.io/docs/swift/latest/

https://itunes.apple.com/app/realm-browser/id1007457278

1. Concepts and Major Classes

In order to better understand what Realm does, here's an overview of the Realm classes and concepts.

Realm: Realm instances are the heart of the framework; it's the access point to the underlying database, similar to a Core Data managed object context. It will create instances using the `Realm()` initializer.
Object: This is your Realm model. The act of creating a model defines the schema of the database; to create a model you simply subclass `Object` and define the **fields** you want to persist as properties.
Relationships: You create one-to-many relationships between objects by simply declaring a property of the type of the `Object` you want to refer to. You can create many-to-one and many-to-many relationships via a property of type `List`, which leads you to…

- **Write Transactions:** Any operations in the database such as creating, editing, or deleting objects must be performed within **writes** which are done by calling `write(_:)` on `Realm` instances.
- **Queries:** To retrieve objects from the database you'll need to use queries. The simplest form of a query is calling `objects()` on a `Realm` instance, passing in the class of the `Object` you are looking for. If your data retrieval needs are more complex you can make use of predicates, chain your queries, and order your results as well.

Results: Results is an auto updating container type that you get back from object queries. They have a lot of similarities with regular `Arrays`, including the subscript syntax for grabbing an item at an index.

2. Realm Installation & Usage (Swift version)

- **Install Realm Swift version with cocoa pod**

  pod 'RealmSwift'

- **Create or Open Default realm object as in eg.**

```
// Get the default Realm - file name is default.realm in iOS
do {
    let realm = try Realm()
} catch let error as NSError {
    // handle error
}
```

```
// Persist your data easily
try! realm.write {
    realm.add(yourObject)
}
```

- **Create Realm class, then object instant and save**

```
class Dog: Object {
    @objc dynamic var name = ""
    @objc dynamic var age = 0
}
```

Realm data models are defined as regular Swift classes with regular properties. To create one, simply subclass `Object` or an existing Realm model class. Realm model objects mostly function like any other Swift objects. You can define your own methods on them, conform them to protocols, and use them like you would any other object. The main restriction is that you can only use an object on the thread which it was created.

// Use them like regular Swift objects & Save

```
let myDog = Dog()
myDog.name = "Rex"
myDog.age = 1
print("name of dog: \(myDog.name)")
try! realm.write {
    realm.add(myDog)
}
```

```
//Save again

try! realm.write {
    myDog.age = 5
}
```

. **Getting Data back from DB, by querying**

```
// Query Realm for all dogs less than 2 years old
let puppies = realm.objects(Dog.self).
filter("age < 2")
puppies.count
```

3. **Configuration Realm and setting custom DB**

```
var realmFileName = "user1"

var config = Realm.Configuration()

config.fileURL =
config.fileURL!.deletingLastPathComponent()
.appendingPathComponent("\(realmFileName).realm")

Realm.Configuration.defaultConfiguration = config
//OR
let realm = try! Realm(configuration: config)
```

4. **In-Memory Realm**

By setting the `inMemoryIdentifier` rather than the `fileURL` on your `Realm.Configuration`, you can create a Realm that runs entirely in memory without being persisted to disk.

eg.

```
let realm = try! Realm(configuration:
Realm.Configuration(inMemoryIdentifier:
    "MyInMemoryRealm")
```

Note:

In-memory Realms create several files in a temporary directory for coordinating things like cross-process notifications. No data is actually written to the files unless the operating system needs to swap to disk due to memory pressure.

5. Bundling a Realm file

It's common to seed an app with initial data, making it available to your users immediately on first launch. Steps are as follows:

1. First, populate the Realm. You should use the same data model as your final, shipping app to create a Realm and populate it with the data you wish to bundle with your app. Since Realm files are cross-platform, you can use a macOS app (see our JSONImport example) or your iOS app running in the simulator.
2. In the code where you're generating this Realm file, you should finish by making a compacted copy of the file (see `Realm().writeCopyToPath(_:encryptionKey:)`). This will reduce the Realm's file size, making your final app lighter to download for your users.
3. Drag the new compacted copy of your Realm file to your final app's Xcode Project Navigator.
4. Go to your app target's build phases tab in Xcode and add the Realm file to the "Copy Bundle Resources" build phase.
5. At this point, your bundled Realm file will be accessible to your app. You can find its path by using `NSBundle.main.pathForResource(_:ofType:)`.
   eg.

```
let config = Realm.Configuration(
    fileURL: Bundle.main.url(forResource: "MyBundledData",
                             withExtension: "realm"),
                        readOnly: true)

let realm = try! Realm(configuration: config)
```

6. If the bundled Realm contains fixed data that you don't need to modify, you can open it directly from the bundle path by setting `readOnly = true` on the `Realm.Configuration` object. Otherwise, if it's initial data that you'll be modifying, you can copy the bundled file to your application's Documents directory using `NSFileManager.default.copyItemAtPath(_:toPath:)`.

6. Auxiliary Realm files

Alongside the standard `.realm` files, Realm also generates and maintains additional files and directories for its own internal operations.

- `.realm.lock` - A lock file for resource locks.
- `.realm.management` - Directory of interprocess lock files.
- `.realm.note` - A named pipe for notifications.

These files don't have any effect on `.realm` database files, and won't cause any erroneous behavior if their parent database file is deleted or replaced.

7. Compacting Realm files

```
let config = Realm.Configuration(shouldCompactOnLaunch:
{ totalBytes, usedBytes in
        // totalBytes => the size of the file on disk total in bytes
        // usedBytes refers to the number of bytes used by data in the

  // Compact if the file is over 100MB in size and less than 50% 'used'
  let oneHundredMB = 100 * 1024 * 1024
      return (totalBytes > oneHundredMB) && (Double(usedBytes) /
      Double(totalBytes)) < 0.5
      })
do {
  // Realm is compacted on the first open if the configuration block
      conditions were met.
  let realm = try Realm(configuration: config)
} catch {
  // handle error compacting or opening Realm
}
```

Note:

The compaction operation works by reading the entire contents of the Realm file, rewriting it to a new file at a different location, then replacing the original file. Depending on the amount of data in a file, this may be an expensive operation and we need to balance between performing the compaction too often and letting Realm files grow too large.
Finally, if another process is accessing the Realm, compaction will be skipped even if the configuration block's conditions were met. That's because compaction cannot be safely performed while a Realm is being accessed.

8. Deleting Realm files

In some cases, such as clearing caches, or resetting your entire dataset, it may be appropriate to completely delete a Realm file from disk.

Because Realm avoids copying data into memory except when absolutely required, all objects managed by a Realm contain references to the file on disk, and must be deallocated before the file can be safely deleted.

In practice, deleting a Realm file should be done either on application startup before you have opened the Realm, or after only opening the Realm within an explicit autorelease pool, which ensures that all of the Realm objects will have been deallocated.

Although not strictly necessary, you should delete auxiliary Realm files as well as the main Realm file to fully clean up all related files.

Eg.

autoreleasepool {

```
    // all Realm usage here
}
let realmURL = Realm.Configuration.defaultConfiguration.fileURL!
let realmURLs = [
```

```
    realmURL,
    realmURL.appendingPathExtension("lock"),
    realmURL.appendingPathExtension("note"),
    realmURL.appendingPathExtension("management")
]
for URL in realmURLs {
    do {
        try FileManager.default.removeItem(at: URL)
    } catch {
        // handle error
    }
}
```

9. Filtering

If you're familiar with NSPredicate, then you already know how to query in Realm. Objects, Realm, List, and Results all provide methods that allow you to query for specific Object instances by simply passing in an NSPredicate instance, predicate string, or predicate format string just as you would when querying an NSArray.

See Apple's Predicates Programming Guide for more information about building predicates and use NSPredicate Cheatsheet. Realm supports many common predicates:

- The comparison operands can be property names or constants. At least one of the operands must be a property name.
- The comparison operators **==**, **<=**, **<**, **>=**, **>**, **!=**, and **BETWEEN** are supported for Int, Int8, Int16, Int32, Int64, Float, Double and Date property types, e.g. age == 45
- Identity comparisons **==**, **!=**, e.g. Results<Employee>().filter("company == %@", company).
- The comparison operators **==** and **!=** are supported for boolean properties.
- For String and Data properties, the **==**, **!=**, **BEGINSWITH**, **CONTAINS**, and **ENDSWITH** operators are supported, e.g. name CONTAINS 'Ja'
- For String properties, the **LIKE** operator may be used to compare the left hand property with the right hand expression: ? and * are allowed as wildcard characters, where ? matches 1 character and * matches 0 or more

characters. Example: `value LIKE '?bc*'` matching strings like “abcde” and “cbc”.

- Case-insensitive comparisons for strings, such as `name CONTAINS[c] 'Ja'`. Note that only characters “A-Z” and “a-z” will be ignored for case. The `[c]` modifier can be combined with the `[d]` ` modifier.
- Diacritic-insensitive comparisons for strings, such as `name BEGINSWITH[d] 'e'` matching **étoile**. This modifier can be combined with the `[c]` modifier. (This modifier can only be applied to a subset of strings Realm supports: see limitations for details.)
- Realm supports the following compound operators: **“AND”**, **“OR”**, and **“NOT”**, e.g. `name BEGINSWITH 'J' AND age >= 32`.
- The containment operand **IN**, e.g. `name IN {'Lisa', 'Spike', 'Hachi'}`
- Nil comparisons **==**, **!=**, e.g. `Results<Company>().filter("ceo == nil")`. Note that Realm treats `nil`as a special value rather than the absence of a value; unlike with SQL, `nil` equals itself.
- **ANY** comparisons, e.g. `ANY student.age < 21`.
- The aggregate expressions **@count**, **@min**, **@max**, **@sum** and **@avg** are supported on `List` and `Results` properties, e.g. `realm.objects(Company.self).filter("employees.@count > 5")` to find all companies with more than five employees.
- Subqueries are supported with the following limitations:
  - **@count** is the only operator that may be applied to the **SUBQUERY** expression.
  - The `SUBQUERY(…).@count` expression must be compared with a constant.
  - Correlated subqueries are not yet supported.

10.Sorting

`Results` allows you to specify a sort criteria and order based on a key path, a property, or on one or more sort descriptors.

```
// Sort tan dogs with names starting with "B" by name
let sortedDogs = realm.objects(Dog.self).filter("color = 'tan' AND name BEGINSWITH 'B'").sorted(byKeyPath: "name")
```

1. Exercise

1. store the following value in app using UserDefault
   Name, age , NRIC and Address by capturing from user input using textfield and load the value from the disk and store if the value is already stored

2. Create a projects using SQLite as database to store and retrieve data of the app

3. Create a project for personal finance using Realm