ATLS 4320/5320: Advanced Mobile Application Development
Week 8: iOS and Realm

Realm

https://realm.io/
Realm is a Danish startup founded in 2011. Realm is a cross-platform object-oriented database that has been open-sourced as of 2016, and is available free of charge to developers. Realm is available for iOS, Android, and JavaScript(React Native and Node.js). It's fast, easy and lightweight and doesn't have the steep learning curve of SQLite or Core Data on iOS. Their mobile platform now includes a server that handles data synchronization and an API to existing databases(Oracle, MongoDB, and others). We're going to use their database for local persistence.

Realm: Realm instances are the heart of the framework; it's your access point to the underlying database. You will create instances using the Realm() initializer.
https://realm.io/docs/swift/latest#realms

Object: Object is the Realm class used to define Realm model objects. 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. https://realm.io/docs/swift/latest#models

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. https://realm.io/docs/swift/latest#model-inheritance and https://realm.io/docs/swift/latest#relationships

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.
https://realm.io/docs/swift/latest#writes

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. https://realm.io/docs/swift/latest#queries

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.

Realm and iOS

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

Create a new single view universal app called groceryList.

In the terminal go into your project's directory and initialize cocoapods.
pod init

Open up the Podfile created in your project directory and add the RealmSwift pod
```
  # Pods for groceryList
```

pod 'RealmSwift'

now install the pod
pod install

When installation is finished open up your xcworkspace file.

Delete the ViewController.swift file.
Go into the Storyboard and delete the view controller.
Add a table view controller and embed it in a navigation controller.
Make the navigation controller the Initial View Controller.
Give the navigation item the title “Groceries”.
For the table view cell make the style Basic and give it a reuse identifier “cell”.
Go into the connections inspector and make sure the dataSource and delegate are connected to the View Controller(Groceries).

Add a Cocoa touch classes to control this view called GroceryTableViewController and subclass TableViewController.
Back in the storyboard change the views to use this classes.

In the AppDelegate import RealmSwift and if it gives you an error, build your project.

```
import RealmSwift
```

Create a data model class to represent your grocery items. Your class must inherit from Object, the Realm class for defining Realm model objects. Realm doesn’t support Swift structs as models.
https://realm.io/docs/swift/latest#faq-swift-structs
Specific datatypes in Realm, such as strings, must be initialized with a value, so we use an empty string.

The act of creating a model defines the schema of the database, so your class properties represent the data being stored in the database. If you create multiple classes you will have multiple Realm models, and you can create relationships between the models by using a class type for a property in another class.

```
import RealmSwift
class Grocery: Object {
    @objc dynamic var name = ""
    @objc dynamic var bought = false
}
```

All property types(except for List and RealmOptional) must be declared as @objc dynamic var.
By applying the dynamic declaration modifier to a member of a class, you tell the compiler that dynamic dispatch should be used to access that member. Swift uses static and virtual dispatch whenever possible over dynamic dispatch, whereas Objective-C only uses dynamic dispatch. The dynamic declaration forces the use of dynamic dispatch, which is required by Realm. As dynamic dispatch is Objective-C you need to mark it with the objc attribute to make the declaration available in Objective-C.
Dynamic dispatch means that the Objective-C runtime decides at runtime which implementation of a particular method or function it needs to invoke. For example, if a subclass overrides a method of its superclass, dynamic dispatch figures out which implementation of the method needs to be invoked, that of the subclass or that of the parent class.

The rest of the code will be in GroceryTableViewController

```
import RealmSwift

    var realm : Realm! //Realm database instance

    var groceryList: Results<Grocery> {
        get {
            return realm.objects(Grocery.self) //returns all Grocery objects
from Realm
        }
    }
```

Results is an auto-updating container type in Realm returned from object queries. Since it's a generic type so you need to specify the type in < >

Update viewDidLoad()

```
        //initialize the realm variable
        do {
            realm = try Realm()
        } catch let error {
            print(error.localizedDescription)
        }
```

Update the tableview delegate and data source methods.

```
    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        return groceryList.count
    }

    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "cell",
for: indexPath)

        let item = groceryList[indexPath.row]
        cell.textLabel!.text = item.name
         cell.accessoryType = item.bought ? .checkmark : .none //set
checkmark if bought
        return cell
    }
```

Updating Items

When the user taps a cell we want to mark the item bought, update Realm, and make the change visually.

```
    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        let item = groceryList[indexPath.row]
        try! self.realm.write { //write to realm database
            item.bought = !item.bought
        }
        tableView.reloadRows(at: [indexPath], with: .automatic)
    }
```

To allow row edits, update this method to return true.

```
    override func tableView(_ tableView: UITableView, canEditRowAt
indexPath: IndexPath) -> Bool {
        // Return false if you do not want the specified item to be editable.
        return true
    }
```

Implement deleting an item by deleting it from Realm and removing it from the table.

```
    override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        if editingStyle == .delete {
            let item = groceryList[indexPath.row]
            try! self.realm.write {
                self.realm.delete(item) //delete from realm database
            }
            tableView.deleteRows(at: [indexPath], with: .fade)
        } else if editingStyle == .insert {
            // Create a new instance of the appropriate class, insert it
into the array, and add a new row to the table view.
        }
    }
```

Add Items

In the storyboard add a bar button item to the navigation bar of the GroceryTableViewController and change it to Add. Connect it as an action called addGroceryItem.

```
    @IBAction func addGroceryItem(_ sender: UIBarButtonItem) {
        let addalert = UIAlertController(title: "New Item", message: "Add a
new item to your grocery list", preferredStyle: .alert)
        //add textfield to the alert
addalert.addTextField(configurationHandler: {(UITextField) in
        })
        let cancelAction = UIAlertAction(title: "Cancel", style: .cancel,
handler: nil)
        addalert.addAction(cancelAction)
        let addItemAction = UIAlertAction(title: "Add", style: .default,
```

```
handler: {(UIAlertAction)in
            // adds new item
             let newitem = addalert.textFields![0] //gets textfield
             let newGroceryItem = Grocery()//create new Grocery instance
             newGroceryItem.name = newitem.text! //set name with textfield
text
             newGroceryItem.bought = false

             do {
                 try self.realm.write({
                     self.realm.add(newGroceryItem) //add to realm database
                     self.tableView.insertRows(at: [IndexPath.init(row:
self.groceryList.count-1, section:0)], with: .automatic) //inserts new row
at the end of the table
                 })
             } catch let error{
                 print(error.localizedDescription)
             }
})
         addalert.addAction(addItemAction)
         present(addalert, animated: true, completion: nil)
    }
```

To test, run the app, exit the app, and start it again, the data should still be there.

Realm Browser
Realm Browser allows you read and edit Realm databases from your computer (available on the App store for Mac only). It's really useful while developing as the Realm database format is proprietary and not easily human-readable.
Download the Realm Browser from the Mac app store https://itunes.apple.com/app/realm-browser/id1007457278

To help you find where your Realm database is you can print out the path in viewDidLoad()
`print(Realm.Configuration.defaultConfiguration.fileURL!)`
Navigate there and double click on default.realm and it will open in Realm Browser.
The easiest way to go to the database location is to open Finder, press Cmd-Shift-G and paste in the path. Leave off the file:/// and the file name (Users/aileen/Library/Developer/CoreSimulator/Devices/45F751D5-389F-4EED-AE12-73A28081DEBD/data/Containers/Data/Application/2D9CEC13-8089-4F77-B474-E83578F98179/Documents)

You can view your data but also edit, delete, and add. This is a great tool for debugging.