

## Mobile Application Development

### Week 7: Data Persistence

#### Data Persistence

In most apps users change or add data, and we need that data to be persistent.

Our data model objects hold data and should support data persistence across app launches. There are multiple ways you can implement data persistence.

For apps that need to store some user settings, configurations, or small amounts of data:

- User Defaults
  - Good for user preferences for configuring an app such as a name, settings (i.e. was sound turned on), color scheme, or where the user last navigated to
  - Used for storing data as key-value pairs
  - Not secure (for secure data use keychain access)
- Property lists
  - Data is stored in XML
  - Simple to use
  - Supports a limited number of data types
  - Have to read and write the entire file to access and store data
- Object archives
  - Supports more data types including custom objects
  - More scalable than property lists
- Not meant to store larger sets of data for your app
- No ability to query data

#### Local Data Stores

- SQLite
  - Very popular RDBMS so many developers have experience with it
  - Relational database embedded in iOS
  - Fairly lightweight and has good performance
  - Must setup the tables, relationships, and SQL queries or use a wrapper
  - API is written in C and the C-style syntax looks dated
    - Wrapper libraries available to simplify this (we will look at this approach on Android)
- Core Data
  - Apple's framework for managing an object graph, not a database
    - Uses SQLite as its database
  - Provides complex queries and complex relationship between objects
  - Has additional features like change tracking, data versioning, automatic validation, advanced undo/redo functionality
  - Steeper learning curve
  - Very useful when you have to manage a lot of complex objects and relations between them but overly complex for simpler data persistence needs

We'll be using plists today. We'll also be looking at the Realm framework as another local option and Firebase as a cloud solution next. These are also both cross platform so they can be used in Android development as well.

## Property Lists

A property list is a simple data file in XML that stores item types and values. They use a key to retrieve the value, just as a Dictionary does. Property lists can have Boolean, Data, Date, Number, and String node types to hold individual pieces of data, as well as Arrays or Dictionaries to store collections of nodes.

All of our apps have an Info.plist file in Supporting Files.

Property lists can be created using the Property List Editor application

(*/Developer/Applications/Utilities/Property List Editor.app*) or directly in Xcode.

We are going to use a property list to store our app's data.

Only certain objects can be stored in property lists and then written to a file.

- Array or NSArray
- Dictionary or NSDictionary
- NSData
- String or NSString
- (and the mutable versions of the above)
- NSNumber
- Date or NSDate

If you can build your data model from just these objects, you can use property lists to save and load your data. This is ok for simple data models. Otherwise use another method.

## Communication

In iOS there are four common patterns for objects to communicate

1. Target-Action: a single object calls a single method when a single event occurs
  - i.e. buttons
2. Delegation: an object responds to numerous methods to modify or add behavior
  - text fields, table views, etc that have delegate methods
3. Notification: Register an object to be notified when an event occurs
  - Sets up how to handle when an event fires
4. Key-Value Observing (KVO): register to be one of many objects notified when single property of another object changes.
  - used for archiving

## Notifications

A notification is a callback mechanism that can inform multiple objects when an event occurs.

- NotificationCenter manages the notification process
- Objects register for the notifications they're interested in
- Notification senders post notifications to a notification center
- The notification center notifies any objects registered for that notification

## Grocery

File | New Project

iOS App

Product Name: Grocery

Team: None

Org identifier: ATLAS (can be anything, will be used in the bundle identifier)

Interface: Storyboard

Language: Swift

Uncheck core data and include tests.  
Uncheck create local git repo

Go into MainStoryboard. The initial scene is a view controller but we want a table view controller. Click on the scene and delete it. Then drag onto the storyboard a table view controller. In the attributes inspector check Is Initial View Controller. For the table view cell make the style Basic and give it a reuse identifier "Cell". Select the table view and go into the connections inspector and make sure the dataSource and delegate are connected to the Table View Controller.

Delete the ViewController.swift file (and move to trash).  
Add a Cocoa touch classes to control this view called GroceryTableViewController and subclass UITableViewController.  
Back in the storyboard select the table view controller and in the identity inspector change the class to GroceryTableViewController,

Since we're going to use a plist to store our list we can't save custom objects, but we can save an array.

In GroceryTableViewController define an array to store our grocery list items.

```
var groceries = [String]()
```

We also need to implement the table view datasource and delegate methods.

```
override func numberOfSections(in tableView: UITableView) -> Int {  
    // #warning Incomplete implementation, return the number of sections  
    return 1  
}  
  
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection:  
section: Int) -> Int {  
    // #warning Incomplete implementation, return the number of rows  
    return groceries.count  
}  
  
override func tableView(_ tableView: UITableView, cellForRowAt  
indexPath: IndexPath) -> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",  
for: indexPath)  
    var cellConfig = cell.defaultContentConfiguration()  
    cellConfig.text = groceries[indexPath.row]  
    cell.contentConfiguration = cellConfig  
    return cell  
}
```

#### Add items

In the storyboard embed the GroceryTableViewController in a navigation controller.  
In the navigation item it adds to the GroceryTableViewController change the title to Groceries.  
Add a bar button item to the navigation bar and change it to Add. Connect it as an action called addGroceryItem.

Go into GroceryTableViewController to implement addGroceryItem(). Instead of using another view to add one item we're going to do it in an alert by adding a textfield in the alert.

```
@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
    if newItem.text?.isEmpty == false {
        let newGroceryItem = newItem.text! //gets textfield text
        self.groceries.append(newGroceryItem) //adds textfield text
to array
        self.tableView.reloadData()
    }
    })
    addalert.addAction(addItemAction)
    present(addalert, animated: true, completion: nil)
}
```

#### Delete items

Now let's add the ability to delete items.

In GroceryTableViewController.swift in viewDidLoad uncomment the following line and update it so the edit button will be on the left since the add is on the right.

```
self.navigationItem.leftBarButtonItem = self.editButtonItem
```

I could put the buttons next to each other on the right by adding a bar button item called Edit in the storyboard but then it won't automatically call the methods needed, so I went with this method.

Uncomment the following:

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

Uncomment and implement

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        //Delete the item from the array
        groceries.remove(at: indexPath.row)
        // Delete the row from the data source
        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
    }
}

```

### Data Persistence

So this is working fine but the data isn't persistent across app launches. Let's use a plist to save our data. Create a GroceryDataHandler class.

File | New File

iOS Source Swift File

GroceryDataHandler

Make sure it's saving to your project folder and the target is checked.

In GroceryDataHandler define an array to store our data.

```
var groceryData = [String]()
```

We'll create a method that will return the url to a given file.

```

func dataFileURL(_ filename:String) -> URL? {
    //returns an array of URLs for the document directory in the user's
    home directory
    let urls = FileManager.default.urls(for:.documentDirectory, in:
    .userDomainMask)
    //append the file name to the first item in the array which is the
    document directory
    let url = urls.first?.appendingPathComponent(filename)
    //return the URL of the data file or nil if it does not exist
    return url
}

```

FileManager.SearchPathDirectory.documentDirectory can be shortened to .documentDirectory (value from the SearchPathDirectory enum)

FileManager.SearchPathDomainMask.userDomainMask can be shortened to .userDomainMask (value from the SearchPathDomainMask enum). On iOS this maps to the user's home directory.

On iOS, there is always only one Documents directory for each application, so we can safely assume that exactly one object will be returned and can be accessed as the first item of the returned array.

Now we'll create our loadData() method that will use this method to get the url of the file and load our data. We'll use a Property List Decoder as we have before.

```

func loadData(fileName: String){
    //url of data file
    let fileURL = dataFileURL(fileName)
    //if the data file exists, use it
    if FileManager.default.fileExists(atPath: (fileURL?.path)!){
        do {
            //creates a data buffer with the contents of the plist
            let data = try Data(contentsOf: fileURL!)
            //create an instance of PropertyListDecoder
            let decoder = PropertyListDecoder()

```

```

        //decode the data using the structure of the Favorite class
        groceryData = try decoder.decode([String].self, from: data)
    } catch {
        print("no file")
    }
}
else {
    print("file does not exist")
}
}

```

To save our data we'll use a Property List Encoder.

```

func saveData(fileName: String){
    //url of data file
    let fileURL = dataFileURL(fileName)
    do {
        //create an instance of PropertyListEncoder
        let encoder = PropertyListEncoder()
        //set format type to xml
        encoder.outputFormat = .xml
        let encodedData = try encoder.encode(groceryData)
        //write encoded data to the file
        try encodedData.write(to: fileURL!)
    } catch {
        print("write error")
    }
}

```

We'll also create methods to return all the grocery items as well as add and delete items.

```

func getGroceryItems() -> [String]{
    return groceryData
}

func addItem(newItem: String){
    groceryData.append(newItem)
}

func deleteItem(index: Int){
    groceryData.remove(at: index)
}

```

In GroceryTableViewController create an instance of the GroceryDataLoader class to load, save and access our data, and constant for our data file.

```

var groceryData = GroceryDataHandler()
let dataFile = "grocery.plist"

```

Our application needs to save its data before the application is terminated or sent to the background, so we'll use the UIApplication.willResignActiveNotification notification. This notification is posted whenever an app is no longer the one with which the user is interacting. This includes when the user quits the application and when the application is pushed to the background.

Update viewDidLoad()

```
        groceryData.loadData(fileName: dataFile)
        groceries = groceryData.getGroceryItems()

        //subscribe to the UIApplicationWillResignActiveNotification
notification
        NotificationCenter.default.addObserver(self, selector:
#selector(self.applicationWillResignActive(_:)), name:
UIApplication.willResignActiveNotification, object: nil)
```

Now we need to add applicationWillResignActive()

```
        //called when the UIApplicationWillResignActiveNotification notification
is posted
        //all notification methods take a single NSNotification instance as
their argument
        @objc func applicationWillResignActive(_ notification: Notification){
            groceryData.saveData(fileName: dataFile)
        }
```

The #selector and @objc syntax is because underneath it's running an Objective-C method.

Lastly, we need to make sure we're keeping the array of data in the data handler class up to date so let's make sure we add and delete items there as well.

Update addGroceryItem() to add the new item to the array in the data handler class.

```
self.groceryData.addItem(newItem: newGroceryItem) // add to data handler
```

And update when we delete an item to delete it from the data handler instance.

```
groceryData.deleteItem(index: indexPath.row) // delete the row from the data
handler
```

To test the app you can't just stop the app in the simulator, as you'll never receive the notification that the application is terminating, and your data will not be saved. You need to swipe up and return to the home screen before stopping the app in the simulator. This is just a function of running in the simulator and is not needed on a device.

Note that in testing this functionality you might need to change your plist file name often, otherwise once the file is created it will always find the file in loadData() and use it. So if you want to start fresh, use a different file name for your plist.