Persistence

Persistence refers to saving user data to an iOS device's persistent storage (long-term flash storage) so that the data can survive even if the iOS device quits or crashes. When an iOS app is running, the data is stored in a volatile storage (i.e., memory). Without persistence, this data will be lost when the app is launched again.

iOS provides a number of mechanisms for persisting data on a device:

- Files
    - Low-level file operations
    - Property list
    - Archiving
- Database
    - SQLite3 (iOS's embedded relational database)
    - Core Data (a object-oriented framework on top of SQLite3)
- Key-value store: User-defaults

Saving data in files are easy. However, the contents in files can only be read/written sequentially. If at any time, a change has been made, the entire file has to be written to the storage. Therefore, files are more suitable for storing small amounts of data that do not require frequent changes, e.g., application settings (Info.plist).

A database, on the other hand, allows random access to the data. If a change has been made, only the change needs to be saved into the database. Therefore, a database is a suitable way to handle a large set of data with frequent modifications.

In this document, we will discuss three persistence strategies, object archiving, core data, and user-defaults.

Archiving

Object archiving is a technique for generic serialization. Archiving converts any objects in a structured format and then writes the structure to a file. There are two types of archives: a keyed archive and a non-keyed archive. In a keyed archive, all the objects are associated with unique names (keys). These objects are unarchived by their keys. Whereas in a non-keyed archive, the objects have to be decoded (unarchived) in the same order in which they were encoded (archived).

Keyed archives provide better support for forward and backward compatibility. The classes for doing keyed archiving are *NSKeyedArchiver* and *NSKeyedUnarchiver.*

Any objects that need to be archived must conform to the *NSCoding* protocol. This protocol defines two methods, one for archiving (encoding) and the other for unarchiving (decoding).

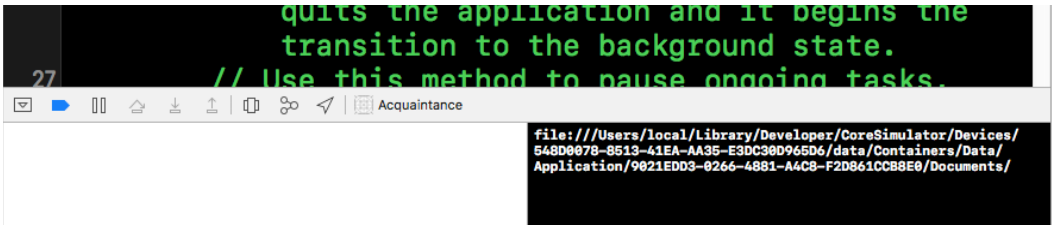
Now, let's use the keyed archiving to make the *Acquaintance* app persistent. The archive file will be named *acqList.plist* and will be stored in this application's /*Documents* directory. iOS uses URLs to refer to the files. In *Person.swift*, define two type constants representing the URLs of /Documents directory and the achieve file:

```
// MARK: Archiving Paths
static let DocumentsDirectory = FileManager().urls(for: .documentDirectory, in: .userDomainMask).first!
static let ArchiveURL = DocumentsDirectory.appendingPathComponent("acqList.plist")
```

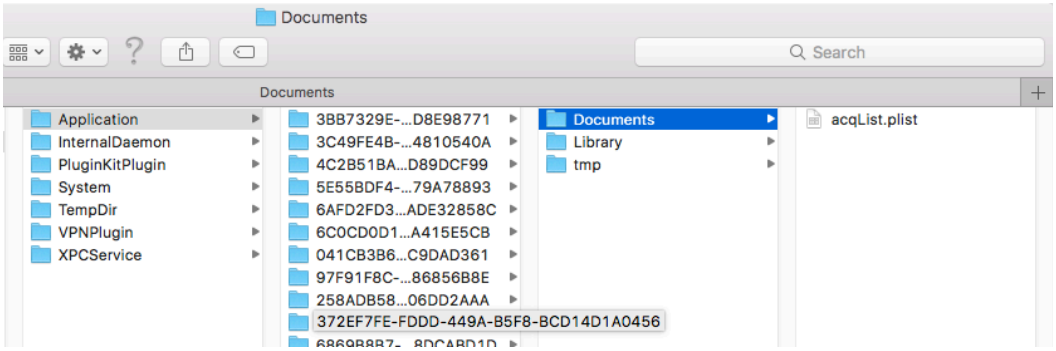
Then open *AppDelegate.swift*, add a line to print out the value of the /Documents Directory in *application(_:didFinishLaunchingWithOptions:)* method:

```
print(Person.DocumentsDirectory)
```

Build, run, and then stop the app. In the Xcode debug area, you will see the path of the app's /Document directory. The folder name is a random ID that Xcode picks when it installs the app on the Simulator or the device. Anything inside that folder is part of the app's sandbox.



Open a new Finder window. In the menu, select Go->Go to Folder…, and paste the full path in the dialog. (Don't include the file://. The path starts with /Users/yourname/…). The finder window will navigate to that folder. Currently, this folder does not contain anything. Later, the persistent file acq.plist will be created in this folder.



Now, let's make the *Person* class archivable, i.e., conform to the *NSCoding* protocol. Open the *Acquaintance* project in Xcode and open *Person.swift*, add *NSCoding* after the class name.

```
class Person: NSCoding
```

Since we are using keyed archiving, we need to define keys (strings) for the properties in the *Person* class. A common method is to define them as type constants in a structure.

```
struct PropertyKey {
    static let nameKey = "name"
    static let photoKey = "photo"
    static let phoneKey = "phone"
    static let emailKey = "email"
    static let notesKey = "notes"
}
```

We then encode the object in the e*ncode(with:)* method from the *NSCoding* protocol:

```
func encode(with aCoder: NSCoder) {
    aCoder.encode(name, forKey: PropertyKey.nameKey)
    aCoder.encode(photo, forKey: PropertyKey.photoKey)
    aCoder.encode(notes, forKey: PropertyKey.notesKey)
}
```

The decoding method from the *NSCoding* protocol is defined as a fallible convenience initialization method, when a *Person* object is created and populated with the archived data. This method is defined as follows:

```
required convenience init?(coder aDecoder: NSCoder) {
    let name = aDecoder.decodeObject(forKey: PropertyKey.nameKey) as? String
    let photo = aDecoder.decodeObject(forKey: PropertyKey.photoKey) as? UIImage
    let notes = aDecoder.decodeObject(forKey: PropertyKey.notesKey) as? String

    if name == nil {
        return nil
    }

    // Must call designated initializer.
```

```
        self.init(name: name!,
                  photo: photo,
                  notes: notes)
}
```

Next, we will define a method to archive the acquaintance list and save it to the file, and another method to load the file and restore the list from the archive. Both methods will be defined in *ListTableViewController.swift*.

```
    func saveList() {
        let isSuccessfulSave = NSKeyedArchiver.archiveRootObject(acqList, toFile: Person.ArchiveURL.path)
        if !isSuccessfulSave {
            print("Failed to save the acquaintance list ...")
        }
    }

    func loadList() -> [Person]? {
        return NSKeyedUnarchiver.unarchiveObject(withFile: Person.ArchiveURL.path) as? [Person]
    }
```

NSKeyedArchiver.archiveRootObject traverses every objects in the *acqList* array, encodes them one by one, and then writes the archive to the file *acqList.plist*. *NSKeyedUnarchiver.unarchiveObject* reads the archive from the file, decodes and reconstitutes the original array.

In this app, all the modifications to *acqList* happen in *ListTableViewController*'s *unwindToList(segue)* method. So we just need to add the following one line of code at the end of this method to call the *saveList()* method save the changes to the persistent storage.

```
@IBAction func unwindToList(segue:UIStoryboardSegue) {
        if segue.identifier == "SaveToList",
           let detailViewController = segue.source as? DetailTableViewController,
           let person = detailViewController.person {
             ……
        }

        saveList()
  }
```

We should load the acquaintance list from the persistence storage when the List table view is created and initialized (i.e. in the *init* method). When the app launches for the first time, there is no saved acquaintance list. In this case, we can instead populate the acquaintance list with the sample list. To implement this loading function, we first need to make the following changes to the *acqList* property.

```
var acqList = [Person]()
var sampleList = [Person("Ameir Al-Zoubi"),  …Person("Sean McMains")]
```

A table view controller, like many objects, has more than one init method:

- init?(coder) for view controllers that are automatically loaded from a storyboard
- init(nibName, bundle) for view controllers that you manually want to load from a nib (a nib is like a storyboard but only contains a single view controller)
- init(style) for table view controllers that you want to create without using a storyboard or nib

This view controller comes from a storyboard, so we'll put the loading code into the *init?(coder)* method. The *init?(coder)* should look like the following:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    // Load any saved acquaitance, otherwise load sample data.
    if let savedList = loadList() {
        acqList += savedList
    } else {
        // Load the sample data.
        for person in sampleList {
            person!.photo = UIImage(named: person!.name)
            acqList.append(person!)
        }
    }
}
```

The last thing you need to do is to comment out the loading code in *ViewDidLoad*():

```
    /*
    for person in acqList {
        person!.photo = UIImage(named: person!.name)
    }*/
```

Build and run the app. You can make some changes to the existing entries and/or add new entries. Stop the app. Double-click the Home button to remove the app from background running. Re-start the app. The changes you made should persist. In Finder, navigate to this app's /Documents folder. You should see acqList.plist residing in that folder. You can double-click to open this file in Xcode or other text editors to examine its content.

Core Data

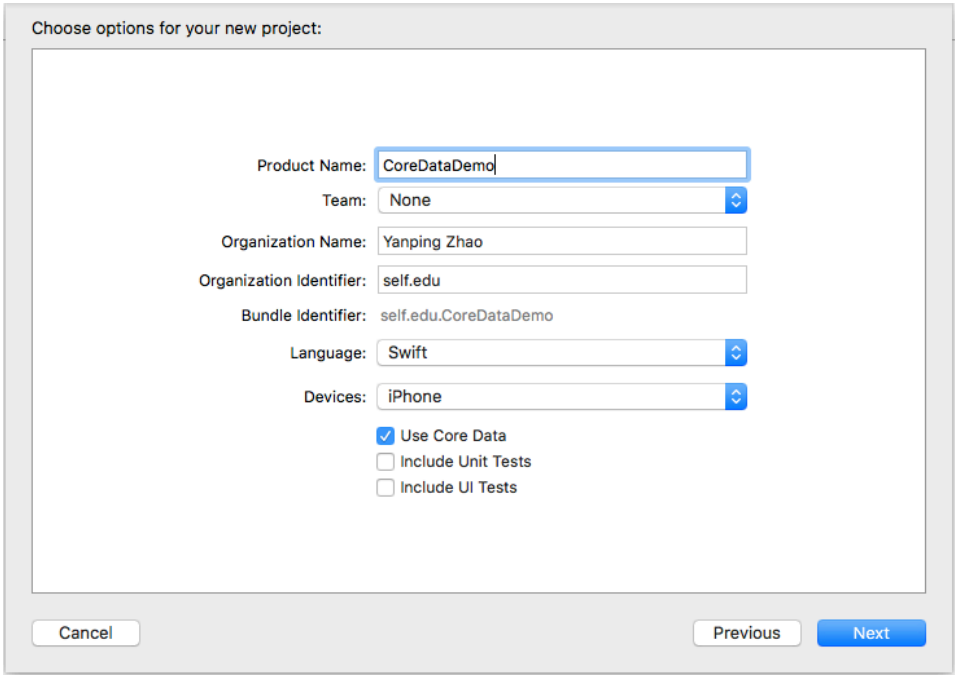
Core Data is not exactly a relational database - it is actually a framework that lets developers interact with database (or other persistent storage) in an object-oriented way. SQLite database is the default persistent store for Core Data on iOS.

If you directly use SQLite3 to save data, you are responsible for writing the code to connect to the database and retrieve or update the data using SQL. This would be a burden for developers, especially for those who do not know SQL.

Core Data provides a simpler way to save data to a persistent store of your choice. You can map the objects in your app to the tables in the database. It allows you to manage records (select/insert/update/delete) in the database without even knowing any SQL.

Using Core Data Template

The simplest way to use Core Data is to enable the Core Data option during project creation. Xcode will generate the required code in *AppDelegate.swift* and create the data model file for Core Data.



If you create a CoreDataDemo project with Core Data option enabled, you will see the following variables and method generated in the *AppDelegate* class:

```
// MARK: - Core Data stack
    lazy var persistentContainer: NSPersistentContainer = {
……
}
// MARK: - Core Data Saving support
    func saveContext () {
……
```

```
}
```

The generated code provides a variable and a method:

- The variable *persistentContainer* is an instance of *NSPersistentContainer*, and has been initialized with a persistent store named "CoreDataDemo". The variable will be used to interact with the Core Data stack.
- The *saveContext*() method provides data saving. When you need to insert/update/delete data in the persistent store, you will call up this method to save the changes to the persistent store.

The simplest way to use Core Data in the *Acquaintance* app is to copy/past the code in the *AppDelegate.swift* file of the *Acquaintance* project. Change the name of the persistent store from *CoreDataMemo* to *Acquaintance.*

```
let container = NSPersistentContainer(name: "Acquaintance")
```

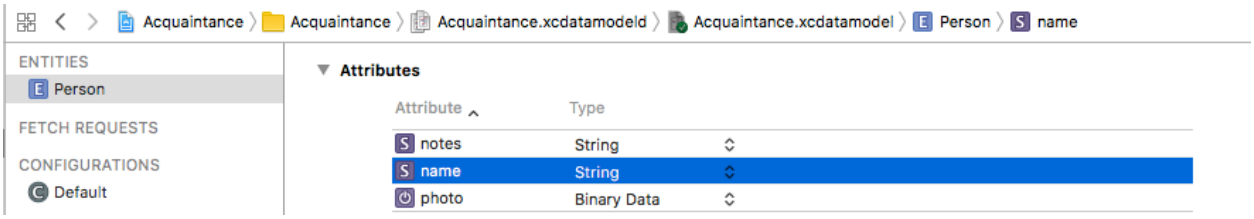
Finally, add the following *import* statement at the beginning of the *AppDelegate* class to import the Core Data framework.

```
import CoreData
```

Creating the Data Model

To create the data model, from the menu, select File->New…. Choose Core Data and select *Data Model*. Name the model *Acquaintance* and click Create. Once created, you should find a file named *Acquaintance.xcdatamodeld* in the project navigator.

Select it to open the data model editor. We are going to create a *Person* entity that matches the *Person* class. Click the *Add Entity* button at the bottom of the editor pane and name the entity *Person*. Then for every property of the *Person* class, we are going to add an attribute of the matching type in the entity. Click the + button under the attributes section to create a new attribute.



Select a particular attribute, you can further configure its properties in the Data Model inspector. Since the *name* property is a required property, you should uncheck the Optional checkbox to make it mandatory. The other two attributes can be kept *Optional*.

For the *photo* property, we will save the actual data of the image into database, so we set its type to *Binary Data.* In the Data Model Inspector, check the *Store in External Record File* checkbox so that the image data are saved outside of the database to make the database smaller.

Creating the Managed Objects

Now that you've created the managed object model, the next thing is to create managed object classes for the entities in the model. In this app, we are going to create one managed object class for the *Person* entity. Although Xcode 8 can automatically generate the managed object class, we will manually do that so that we have a better understanding of how a managed object class works.

First delete *Person.swift* from the project. Select the *Person* entity, in the Data Model Inspector, set the *Name* to *PersonMO* and also change *Codegen* to *Manual/None*. From the menu bar, choose *Editor-> Create NSManagedObject Subclass*. Following the instructions and you will see two files created in the project. One is named *PersonMO+CoreDataClass.swift*, and the other is *PersonMO+CoreDataProperties.swift*.

In *PersonMO+CoreDataProperties.swift*, Xcode has created properties for the attributes that you specified in the Data Model editor. If you change the Core Data model at some later time and you want to automatically update the code to match those changes, you can choose *Create NSManagedObject Subclass* again and Xcode will only overwrite what is in *PersonMO+CoreDataProperties.swift*.

Since we are going to save the person object into the database, we now have to replace the original Person class with the new PersonMO class.

Start with the *ListTableViewController.swift* file to fix compiling errors. We no longer need to initialize the acquaintance list with default values as we will pull the data from database. Therefore, we redefine *acqList* as:

```
var acqList = [PersonMO]()
```

Then comment out the initialization code in *ViewDidLoad*:

```
        /*
        for person in acqList {
            person!.photo = UIImage(named: person!.name)
        }*/
```

Next, modify *tableView(_:cellForRowAt:)* as follows:

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

        let person = acqList[indexPath.row]

        // Configure the cell...
        if let photoData = person.photo {
            cell.photoImageView.image = UIImage(data: photoData)
        } else {
            cell.photoImageView.image = UIImage(named:"photoalbum")
        }
        cell.nameLabel.text = person.name
        cell.notesLabel.text = person.notes

        return cell
    }
```

Switch to *DetailTableViewController.swift*, first change the declaration of the instance variable *person*:

```
var person: PersonMO?
```

Again, you will see some errors after changing the code. In the *viewDidLoad*() method, initialize the photo using data instead of name:

```
if let photoData = person?.photo {
        photoImageView.image = UIImage(data: photoData)
    } else {
        photoImageView.image = UIImage(named:"photoalbum")
    }
```

In *prepare(segue:)*, comment out the code block inside the *if segue.identifier == "SaveToList"* statement, since we are going to create an object in the database.

Build and run the app. As there is no data, the app should now display a blank table when launched. Next, we'll modify the *DetailTableViewController* class to insert and save a new acquaintance object to database.

Creating a New Person to the Database

The following method *addToContext* in *AppDelegate.swift* first creates a new *PersonMO* object with the view context of the persistent container, sets the object's properties, and then call the *context.save* method to save this object permanently to the database. This method also returns the newly created object.

```
func addToContext(name: String,
                  photo: UIImage?,
                  notes: String?) -> PersonMO {
      let person = PersonMO(context: persistentContainer.viewContext)

      person.name = name
      if let photo = photo {
         person.photo = UIImagePNGRepresentation(photo)
      }
      person.notes = notes

      print("Saving new person to context ...")
      saveContext()

      return person
   }
```

The place where a new object is created is in the *prepare(for:)* method of the *DetailTableViewController* class. When the user clicks the *Save* button, this method is called to prepare the *Person* object with what the user enters in the interface before returning to the List table view page. Add the following code to this method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
      // Get the new view controller using segue.destinationViewController.
      // Pass the selected object to the new view controller.
      if segue.identifier == "SaveToList", let appDelegate = (UIApplication.shared.delegate as?
AppDelegate) {
         let name = nameTextField.text!
         let photo = photoImageView.image
         let notes = notesTextView.text

         if person == nil { // add a new entry
            self.person = appDelegate.addToContext(name: name, photo: photo, notes: notes)
         } else { // updating the existing entry

         }
      }
   }
```

In the above method, *UIApplication.shared.delegate as? AppDelegate* returns a reference to the *AppDelegate* object.

If you run the app now and add some person, the app should be able to save the records into database without any errors. However, your app is not ready to display the list just added when it is re-launched. We need to write some code to fetch the records from the database when the app starts.

Fetching Data From Database

The simplest way to fetch data from the database is to create a fetch request and then invoke the fetch method provided by the view context. In *AppDelegate.swift,* add a new method for fetching the acquaintance list:

```
func fetchContext() -> [PersonMO]? {

    var fetchedList:[PersonMO]? = nil

    do {
        fetchedList = try persistentContainer.viewContext.fetch(PersonMO.fetchRequest()) as?[PersonMO]
    } catch {
        print("Failed to fetch employees: \(error)")
    }

    return fetchedList
}
```

The generated *PersonMO* class has a built-in *fetchRequest*() method. When being called, it returns an *NSFetchRequest* object that specifies the search criteria and which entity to search (here, it is the Person entity).

With the fetch request, we can then call the fetch method of managed object context to retrieve data from a persistent store (here, it's the database). The *fetch* method returns an array of *PersonMO* objects or nil.

Note that Swift comes with an exception-like model using try-throw-catch keywords. You use do-catch statement to catch errors and handle them accordingly. As you may notice, we put a *try* keyword in front of the method call. In *ListTableViewController.swift*, call *fetchContext* in the *ViewDidLoad()* method so that the list table view will display all the person records in the database.

```
override func viewDidLoad() {
    super.viewDidLoad()

    if let appDelegate = (UIApplication.shared.delegate as? AppDelegate), let fetchedList =
appDelegate.fetchContext() {
        acqList += fetchedList
    }
}
```

Build and run the app. Add some entries to the *List* table. Quit the app. Double-click the Home button and remove the app from background running. Relaunch the app. The list table view should show the newly added entries.

Deleting Data Using Core Data

To delete a record from the persistent data store, you just need to call the method *delete* of the managed object context with the object to delete. Then, you call the saveContext method to apply the changes. In *AppDelegate.swift*, define a new method

```
func deleteFromContext(person: PersonMO) {
        persistentContainer.viewContext.delete(person)

        print("deleting the person from context ...")
        saveContext()
    }
```

Deleting happens when a user swipes on the row of the List table view. Therefore, add the following code in the *tableView(:commit)* method of *ListTableViewController.swift*.

```
if editingStyle == .delete {
            // Delete the row from the data source
            let person = acqList[indexPath.row]
            if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {

                appDelegate.deleteFromContext(person: person)
                acqList.remove(at: indexPath.row)

                tableView.deleteRows(at: [indexPath], with: .fade)
            }
        }
```

Build and run the app to test if you can permanently delete a record in the app.

Updating a Managed Object

Similar to creating a new acquaintance, you can update an existing acquaintance record in the persistent store by updating the corresponding managed object and then call saveContext() to apply the changes.

In *AppDelegate.swift*, define a new method *updateToContext* as follows:

```
func updateToContext(person: PersonMO,
                     name: String,
                     photo: UIImage?,
                     notes: String?) {
      person.name = name
      if let photo = photo {
         person.photo = UIImagePNGRepresentation(photo)
      }
      person.notes = notes

      print("updating the person to context ...")
      saveContext()
   }
```

This method is called in the *prepare(segue)* method of the *DetailViewController* class when the user taps the *Save* button. Modify that method as follows:

```
if person == nil { // add a new entry
   person = appDelegate.addToContext(name: name, photo: photo, notes: notes)
} else { // update an existing entry
    appDelegate.updateToContext(person: person!, name: name, photo: photo, notes: notes)
}
```

Build and run the app. Test if the app can save the changes you make after relaunch.

Preloading the database

We have learned how to save data into the database and retrieve them back. In this section, we will discuss how to preload existing data into the database. The data file can be either bundled in the app or hosted on a server. One way to implement preloading is to check if the database file exists. In iOS 10, the database file is located under /Library/ Application Support folder. If SQLite3 is used, the name of the database file is xxx.sqlite (here it is *Acquaintance.sqlite*).

We can first define the database file URL in *PersonMO+CoreDataProperties* as:

```
// MARK: Core Data Paths
    static let ApplicationSupportDirectory = FileManager().urls(for: .applicationSupportDirectory,
in: .userDomainMask).first!
    static let StoreURL = ApplicationSupportDirectory.appendingPathComponent("Acquaintance.sqlite")
```

As an example, the data to be preloaded is an array of names, define in *ListTableViewController.* Add the preloading code in the *ViewDidLoad* method of the *ListTableViewController* class as follows. If the database file does not exist, we populate the database with an array of pre-defined names. Otherwise, we fetch data directly from the database.

```
override func viewDidLoad() {
        super.viewDidLoad()
       if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
           if !FileManager().fileExists(atPath:PersonMO.StoreURL.path) {
                  for name in acqNames {
                     let person = appDelegate.addToContext(name: name, photo: UIImage(named: name),
notes: nil)
                     acqList.append(person)
                  }
           } else {
              if let fetchedList = appDelegate.fetchContext() {
                 acqList += fetchedList
              }
           }
       }
    }
```