Instructor Code

<https://github.com/Arusey/Coredata-Bucket.git>

**Data Persistence**

There are many ways to persist data in an iOS app. The most common ways are:

* NSCoding
* NSUserDefaults
* Core Data
* Back-end API Server

We'll be focusing heavily on Core Data and using a Back-end API server, but it is important to note that each of these methods of data persistence has its pros and cons.

* NSCoding
  + A method of persisting data where you encode data and save it in a file. Gives great control over how the data is structured, but may not be the best option when dealing with relational data. Data is stored on the device.
* NSUserDefaults  
  + Very easy to use, but generally reserved for small pieces of data like settings and user preferences. Should not be used for storing any core data for your application. Data is stored on the device.
* Core Data
  + An iOS framework that uses an object-oriented style for storing and modeling data. A robust and relatively easy solution to implement for larger data sets and relational data. Data is stored on the device.
* Back-end API Server
  + Using the full force of a back-end API server allows one to have full control over their database and use any database (MySQL, MongoDB, PostgreSQL, etc.). Relies on an internet connection. Data is not stored on the device.

We will be starting off with Core Data and then we will learn how to use a Back-end API server in the later chapters.

**Core Data Basics**

For an iOS application to use a database, it relies on a Cocoa API known as  *Core Data.*For those of you familiar with SQL, you probably want to separate the concept of "database" into two kinds: There are relational databases, which is what you're probably used to, and object databases, which is what Core Data provides.

In a relational database, the key actions are  *select* (SELECT \* FROM *table\_name*) and *transaction* (INSERT, UPDATE or DELETE). In the *select,*we ask the database to form itself into a collection of columns (through JOIN statements), filter and sort according to some set of rules, and then return a linear collection of rows. In the *transaction*, we inform the system that we are performing some series of data transformations, we then commit the transaction. A great model but Core Data doesn't look much like it.

The main adjustments for developers coming from RDBMS-land (Relational Database Management System) transitioning to Core Data are:

1. The concept of Unique Identifiers (Primary keys) - They are going on behind the scenes but you never have to worry about them and should not try to define your own.
2. Setting up relationships between objects - We don't save the unique ID of a related object, instead, we use Swift semantics to relate the objects.
3. Data Normalization - You don't need to worry about data normalization in Core Data. In practice, you still need to worry about good data design, because that's always important, but you don't need to worry as much about the underlying implementation details as in an SQL database.

Core Data is hardware-agnostic. Runtime support is available on the same processors that MacOS X and iOS support. Plus, Core Data works directly with SQLite, the public-domain database engine bundled with MacOS X and iOS. This means that in any Apple ecosystem, you can use the same API commands to work your database.

We can also save data on to a  **.plists**file. Although useful, we would not consider plists for anything other than saving simple preferences and really basic data such as remembering preferred default language, audio/video preferences, default templates, etc. Performance wise, retrieving and saving data to plists can be quite slow.

Core Data will give you the flexibility to expand your object models as needed, easily fetch objects from the persistent storage based on certain parameters, create relationships between models (One-to-Many, One-to-One), memory management, etc.

* Core Data manages save and undo functionality for you. It has a persistent store, which tracks changes, and can be flushed to the disk automatically at any number of times (app close, function call, etc.).
* Core Data and related classes provide easy ways to get your entities into UITableViews, like NSFetchedResultsController.
* Core Data abstracts away a lot of the messy things you'd otherwise have to deal with yourself, such as lists of objects, one-to-many or many-to-many relationships, or constraints on object attributes, into a  \*clean object-oriented interface\*.
* Core Data comes with a nice graphical object model editor that can help you think through your object/entity design, and refine it as you go.

**Let's Start by Adding Core Data to our Bucket List project**

Most likely your Bucket List project does not contain Core Data already. **If you have a file with the .xcdatamodeld extension then you do have Core Data already in your project and you do not need to go through the steps below** (they should already be done). If you don't have the .xcdatamodeld file, then follow the directions bellow to add core data to your project.

First, open your **AppDelegate.swift** file and add the following import statement at the top:

import CoreData

Next, go to File > New > File and under the iOS section select Core Data and then add a new Data Model. Name the Data Model "BucketList". If you start a project with Core Data this file will have the same name as the project.

After creating the Data Model you should now have a file in your project named "BucketList.xcdatamodeld"

Next go back to your AppDelegate.swift file and add the following code at the bottom of the class (inside of the last bracket).

// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentContainer = {

/\*

The persistent container for the application. This implementation

creates and returns a container, having loaded the store for the

application to it. This property is optional since there are legitimate

error conditions that could cause the creation of the store to fail.

\*/

let container = NSPersistentContainer(name: "BucketList")

container.loadPersistentStores(completionHandler: { (storeDescription, error) in

if let error = error as NSError? {

// Replace this implementation with code to handle the error appropriately.

// fatalError() causes the application to generate a crash log and terminate. You should not use this function in a shipping application, although it may be useful during development.

/\*

Typical reasons for an error here include:

\* The parent directory does not exist, cannot be created, or disallows writing.

\* The persistent store is not accessible, due to permissions or data protection when the device is locked.

\* The device is out of space.

\* The store could not be migrated to the current model version.

Check the error message to determine what the actual problem was.

\*/

fatalError("Unresolved error \(error), \(error.userInfo)")

}

})

return container

}()

// MARK: - Core Data Saving support

func saveContext () {

let context = persistentContainer.viewContext

if context.hasChanges {

do {

try context.save()

} catch {

// Replace this implementation with code to handle the error appropriately.

// fatalError() causes the application to generate a crash log and terminate. You should not use this function in a shipping application, although it may be useful during development.

let nserror = error as NSError

fatalError("Unresolved error \(nserror), \(nserror.userInfo)")

}

}

}

Finally add the following line inside of the "applicationWillTerminate(application: UIApplication)" method of AppDelegate:

// You should have this method already in your AppDelegate.swift file. Only add the line inside.

func applicationWillTerminate(application: UIApplication) {

// Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.

// ------ ADD THE LINE BELOW ------

self.saveContext() // ADD THIS LINE

}

**Wow, that's a lot of code.***Do not worry about understanding this code right now.*Apple would give us this code automatically if we had checked the "use CoreData" field when creating the project. In all of your future projects where, you know, you will use Core Data you will check the box when creating the project and you won't have to do this.

On the next tab, we'll get started creating our entity and storing data!

# Add CoreData to BucketList

**Note: This example is recorded in swift 3. All examples will work but there is more up to date syntax in the following tab.**

### Setup

Follow the instructions on previous page to alter the "AppDelegate" file and create a new file `file > new > file` of type "Data Model" and name it after your app.

**OR**

Start a new project and check off the "Use Core Data" box.

Inside the Data Model `.xcdatamodeld` file, create an entity and add attributes.

### Saving an Item

First we need a reference to the persistentContainer stored in AppDelegate. This will enable you to communicate with Core Data. It's like a scratch pad for our data. Before we commit our request to Core Data, we first need to write what we want (retrieve or save) in the NSManagedObject:

let managedObjectContext = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext

Create new entity, replace "AwesomeEntity" with your entity name, both in the function call and the type cast:

let thing = NSEntityDescription.insertNewObject(forEntityName: "AwesomeEntity", into: managedObjectContext) as! AwesomeEntity

Set attributes of the item, replace "coolTextAttribute" with your attribute names:

thing.coolTextAttribute = "Some Totally Cool Text"

Commit changes using the managedObjectContext:

// This code is identical to the saveContext method in our AppDelegate

// Try and reference that method to make your code cleaner!

if managedObjectContext.hasChanges {

do {

try managedObjectContext.save()

print("Success")

} catch {

print("\(error)")

}

}

### Fetch All Items

Tell Core Data that we want to fetch items of type "AwesomeEntity", replace this with your entity name:

let itemRequest = NSFetchRequest<NSFetchRequestResult>(entityName: "AwesomeEntity")

Iterate through records:

do {

// get the results by executing the fetch request we made earlier

let results = try managedObjectContext.fetch(itemRequest)

// downcast the results as an array of AwesomeEntity objects, replace with your entity name

    // we are assuming items is an array if type AwesomeEntity defined at the top of your class

items = results as! [AwesomeEntity]

// print the details of each item

for item in items {

print("\(item.coolTextAttribute)")

}

} catch {

// print the error if it is caught (Swift automatically saves the error in "error")

print("\(error)")

}

### Reading from Core Data and displaying in our Table View

Previously we had an array of Strings that we would get the count of and loop through to display our data on the Table View and fulfill the TableViewDataSource methods. Rather than use an array of Strings, we will now use an array of objects!

Store the items in an array:

var items = [AwesomeEntity]()

Fetch all items on page load, and save into data source:

override func viewDidLoad() {

super.viewDidLoad()

fetchAllItems()

}

func fetchAllItems(){

    let itemRequest = NSFetchRequest<NSFetchRequestResult>(entityName: "AwesomeEntity")

do {

let results = try managedObjectContext.fetch(userRequest)

items = results as! [AwesomeEntity]

} catch {

print("\(error)")

}

}

Set text of cell with item text:

cell.textLabel?.text = items[indexPath.row].coolTextAttribute

You will still have errors in your other methods because now our array contains "AwesomeEntity" items rather than Strings.

**Note: This example was written in swift 3. It still works but the syntax for CoreData has been simplified in swift 4. See the following tab for syntax updates.**

# Core Data CRUD

### Managed Objects

Our managed object context manages the changes that have been made to the objects that represent, or correspond to, the underlying entries in our database.

Note: in this module we frequently refer to **context,**butkeep in mind that we are referring to a variable that has been previously set, pointing the default viewContext provided in our AppDelegate. The name of the variable doesn't matter:

let context = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext

### Saving

Whether we are creating, updating, or deleting records, in order for those changes to persist we must call **save** on the managed object context, flushing the changes to the database:

do{

    try context.save()

}catch{

    print(error)

}

**Note:** Since our context's .save() method is a throwing function, we need to place it inside of a do/try/catch block.

After calling save on the context, we can exit our app comfortably knowing that our changes will be saved to the database. If you want to make your code a little cleaner, here's a secret: the above code is identical to the saveContext method in AppDelegate, so rather than rewrite it, we can create a reference to that method like this:

let saveContext = (UIApplication.shared.delegate as! AppDelegate).saveContext

// we call this function like normal: saveContext()

// again, keep in mind that you can call it whatever you want, it doesn't have to be saveContext

### Create

Creating managedObjects has gotten a lot easier in Swift 3. Whereas in the past we had to call a creation function from the NSEntityDescription class, passing several pieces of data, the generous engineers at Apple have provided us with the same functionality wrapped into the init function for the ManagedObject class.

If you want to learn more about manually extending managedObject classes and writing custom methods in your model class definitions, check out the advanced assignments in this chapter! Once we have initialized a new managedObject, we set its values and save it to the context. Done!

let newThing = Thing(context: context)

newThing.property = "value"

saveContext()

### Fetch

In iOS, we write fetch requests to get information from our database. The NSManagedObject class for our entity has a built in method fetchRequest() that we use to generate a new request (note the type of the request). When executed against a managed object context, the fetch returns a nice array of managedObjects. Easy!

let thingRequest:NSFetchRequest<Thing> = Thing.fetchRequest()

do {

    let fetchedThings = try context.fetch(thingRequest)

    // Here we can store the fetched data in an array

} catch {

    print(error)

}

### Update

Updating is very easy with CoreData. All we have to do is change the values of a managedObject and then save the context!

someManagedObject.property = "newValue"

// remember that the code bellow is the same as the saveContext method in AppDelegate

saveContext()

Whether we were holding onto this data in a tableView, some array, or literally the results from a fetchRequest, it doesn't matter. As long as we have reference to the object, we can change its properties and save those changes to the database.

### Delete

To delete something from the database, we call upon the delete function of the managed object context. Like all changes to the context, we must save when we are done.

context.delete(someManagedObject)

saveContext()