FREE eBook

LEARNING core-data

#core-data

Table of Contents

About 1

Chapter 1: Getting started with core-data 2

Remarks 2

Examples 2

Creating Your First Model 2

Creating the project 6

Chapter 2: Core Data Stack 8

Remarks 8

Objective-C 8

Swift 2 8

Swift 3 8

NSManagedObjectModel 8

NSPersistentStoreCoordinator 8

NSManagedObjectContext 9

Examples 9

Objective-C Example 9

Swift 2 Example 10

iOS 10 Example in Swift 11

Chapter 3: Creating an Core Data Model 13

Remarks 13

Examples 13

Adding an Entity to Core Data Model 13

Adding Attributes to Entity 14

Adding Relationships to Core Data Model 16

Chapter 4: NSFetchedResultsController 18

Introduction 18

Examples 18

NSFetchedResultsController for UITableView 18

Chapter 5: Sort Descriptors 20

Examples 20

Ordering Data Returned By Fetch Requests 20
Multiple Sort Descriptors 20
Chapter 6: Using Predicates 21
Examples 21
Matching an exact string 21
Substitutions 21
Credits 22

About

Chapter 1: Getting started with core-data

Remarks

Core Data is a framework in Apple's various OS SDK including, but not limited to iOS and OS X. It has two major roles a model layer and a persistence layer. The model layer is used in the management of model objects and persist data. Simply you can store and manage data in an object-oriented interface. Primary features include filtering, querying, sorting, persisting data and creating relationships between data. Other subjects of interest to Core Data projects are NSPredicate, threading, and among others.

An example application of Core Data could a Catalog app for your local library. In the Catalog app a librarian could add or remove books. They could also filter books by genre, sort books by publication date, or search for a specific authors work. An entity "Book" would have various attributes such as title, author, publication date, isbn, call number, etc. Core Data including the above example can also store data gathered from a server.

Major components of the framework include:

- Data Models (entities, attributes, and relationships)
- Core Data Stack (NSPersistentStoreCoordinator,NSManagedObjectModel, NSManagedObjectContext)
- NSFetchRequest
- NSFetchedResultsController

Sources:

Framework Documentation

Programming Guide

Core Data Release Notes 2016

CoreData & Concurrency

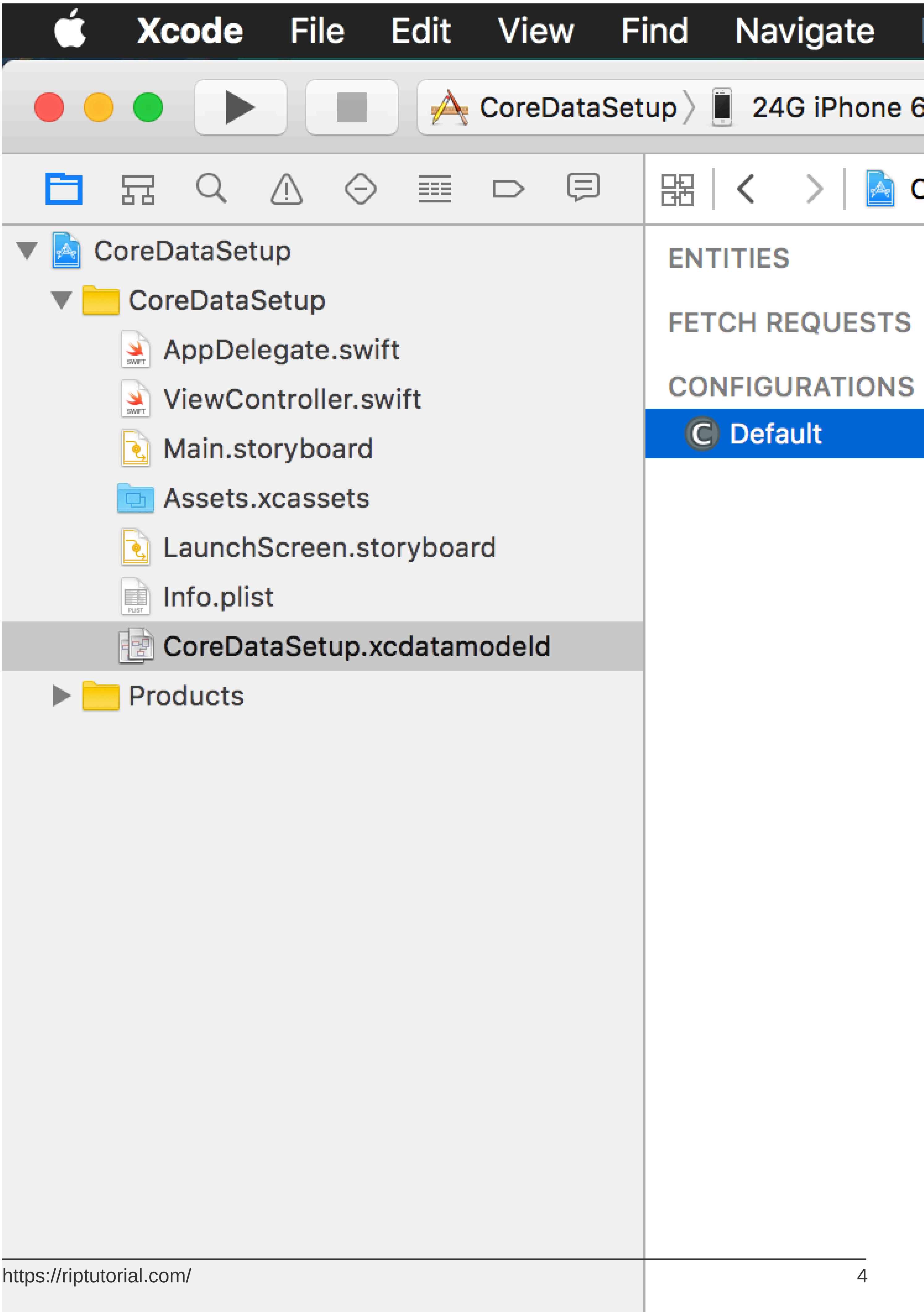
It's important to remember that CoreData is **NOT** thread-safe, which means that if it's necessary to use for example a background-thread to work on ManagedObjects, there are new things to consider, like *PrivateQueue- / MainQueue*-ManagedObjectContexts.

From Apples documentary: *Core Data expects to be run on a single thread. You should never share managed object contexts between threads. This is a hard rule you should not break.*

Examples

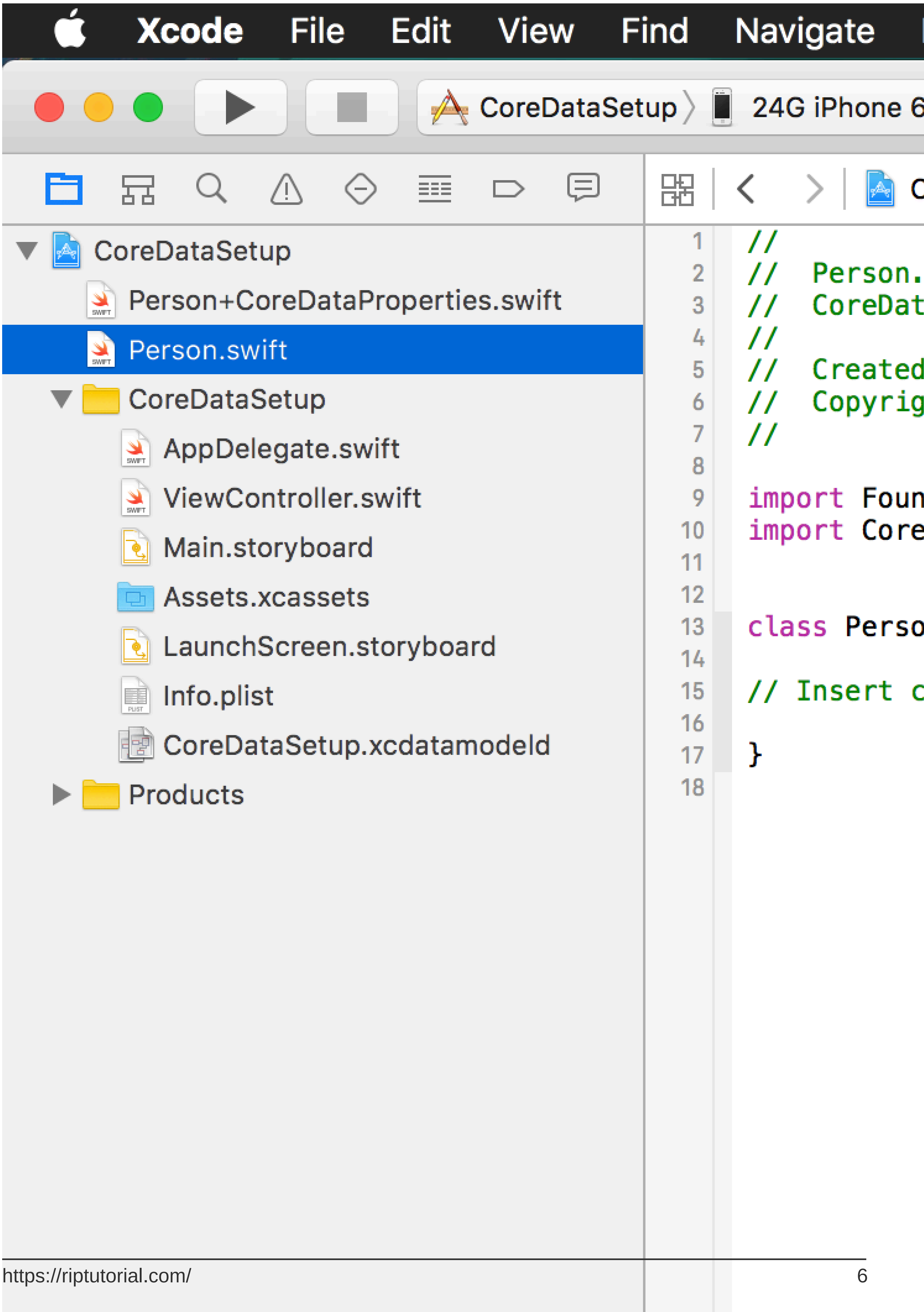
Creating Your First Model

- Select the `.xcdatamodeld` file. You will notice you have no entities. You will have to create one yourself. At the bottom of Xcode you will notice a button that says "Add Entity" click it and you will have a new entity for you to work with on the project.


Xcode
File
Edit
View
Find
Navigate
CoreDataSetup
24G iPhone 6
CoreDataSetup
CoreDataSetup
AppDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
CoreDataSetup.xcdatamodeld
Products
ENTITIES
FETCH REQUESTS
CONFIGURATIONS
Default

- All additions should be done in `Person.swift`, since if you ever change your model and re-run the class generator, it will overwrite everything in `Person+CoreDataProperties.swift`.

  , since if you ever change your model and re-run the class generator, it will overwrite everything in `Person+CoreDataProperties.swift`.



```
//
//  Person.
//  CoreDat
//
//  Created
//  Copyrig
//

import Foun
import Core


class Perso

// Insert c

}

```

https://riptutorial.com/core-data/topic/1718/getting-started-with-core-data

Chapter 2: Core Data Stack

Remarks

This is an implementation of the Core Data Stack which is initially placed in the `AppDelegate` file if the project is created with Core Data when project is created. These functions can also implemented in separate class for `CoreDataStack.swift`. One of the major functions is to get the NSManagedObjectContext.

Objective-C

```
- (NSManagedObjectContext *)managedObjectContext {...}
```

Swift 2

```
lazy var managedObjectContext: NSManagedObjectContext = {...}
```

Swift 3

```
lazy var persistentContainer: NSPersistentContainer = {...)
let managedObjectContext = persistentContainer.viewContext
```

The Core Data stack that communicates between the objects in your application and external data stores. The Core Data stack handles all of the interactions with the external data stores so that your application can focus on its business logic. The stack consists of three primary objects: the managed object context (`NSManagedObjectContext`), the persistent store coordinator (`NSPersistentStoreCoordinator`), and the managed object model (`NSManagedObjectModel`).

`NSManagedObjectModel`

The `NSManagedObjectModel` instance describes the data that is going to be accessed by the Core Data stack. `NSManagedObjectModel` (often referred to as the “mom”) is loaded into memory as the first step in the creation of the stack. An example of the `NSManagedObjectModel` is DataModel.momd. The `NSManagedObjectModel` defines the structure of the data

`NSPersistentStoreCoordinator`

The `NSPersistentStoreCoordinator` realizes objects from the data in the persistent store and passes those objects off to the requesting `NSManagedObjectContext`. It creates new instances of the entities in the model, and it retrieves existing instances from a persistent store (`NSPersistentStore`). The `NSPersistentStoreCoordinator` also verifies that the data is in a consistent state that matches the definitions in the `NSManagedObjectModel`.

NSManagedObjectContext

When you fetch objects from a persistent store, you bring temporary copies onto the scratch pad where they form an object graph (or a collection of object graphs). You can then modify those objects, unless you actually save those changes, however, the persistent store remains unaltered.

All managed objects must be registered with a managed object context. You use the context to add objects to the object graph and remove objects from the object graph. The context tracks the changes you make, both to individual objects' attributes and to the relationships between objects. By tracking changes, the context is able to provide undo and redo support for you. It also ensures that if you change relationships between objects, the integrity of the object graph is maintained.

When you save changes the context ensures that your objects are in a valid state. The changes are written to the persistent store (or stores), new records are added for objects you created, and records are removed for objects you deleted.

Source: Apple Core Data Programming: Initializing the Core Data Stack

Examples

Objective-C Example

This is a simple but robust core-data set-up for iOS 10+. There are exactly two way to access core-data:

1. **viewContext**. The `viewContext` can only be used from the main thread, and only for reading.
2. **strong enqueueCoreDataBlock**. All writing should be done using `enqueueCoreDataBlock`. There is no need to save at the end it will automatically save. All writes are enqueued in an operationQueue so there can never be be write conflicts.

Make sure to NEVER use any managedObjects from context in another context. Also discard all objects that are created or fetched in `enqueueCoreDataBlock` as the context that backs them will be destroyed after the block is executed.

// CoreDataManager.h

```
@interface CoreDataManager : NSObject
@property (nonatomic, readonly) NSManagedObjectContext * viewContext;
- (void)enqueueCoreDataBlock:(void (^)(NSManagedObjectContext* context))block;
@end
```

// CoreDataManager.m

```
@implementation NSManagedObjectContext(SaveIfNeeded)
-(BOOL) saveIfNeeded{
    BOOL toReturn = YES;
    if ([self hasChanges]) {
        NSError *error;
        toReturn = [self save:&error];
        if (toReturn == NO || error)
        {
            //Here you should log to your analytics service
            NSLog(@"--- Failed to commit data\n error: %@", error);
        }
    }
    return toReturn;
}
@end
@interface CoreDataManager ()
@property (nonatomic, strong) NSPersistentContainer* persistentContainer;
@property (nonatomic, strong) NSOperationQueue* persistentContainerQueue;
@end
@implementation CoreDataManager

- (id)init
{
    self = [super init]
    if (self)
    {
        self.persistentContainer = [[NSPersistentContainer alloc]
initWithName:@"PROJECT_NAME_ALSO_NAME_OF_MODEL" managedObjectModel:managedObjectModel];
        [self.persistentContainer
loadPersistentStoresWithCompletionHandler:^(NSPersistentStoreDescription * description,
NSError * error) {
        }];
        self.persistentContainer.viewContext.automaticallyMergesChangesFromParent = YES;
        _persistentContainerQueue = [[NSOperationQueue alloc] init];
        _persistentContainerQueue.maxConcurrentOperationCount = 1;
        _persistentContainerQueue.name = @"persistentContainerQueue";
        dispatch_queue_t queue =
dispatch_queue_create("persistentContainerQueue.dispatchQueue", DISPATCH_QUEUE_SERIAL);
        _persistentContainerQueue.underlyingQueue = queue;
    }
}

- (void)enqueueCoreDataBlock:(void (^)(NSManagedObjectContext* context))block{
    void (^blockCopy)(NSManagedObjectContext*) = [block copy];

    [self.persistentContainerQueue addOperation:[NSBlockOperation blockOperationWithBlock:^{
        NSManagedObjectContext* context =  self.persistentContainer.newBackgroundContext;
        [context performBlockAndWait:^{
            blockCopy(context);
            [context saveIfNeeded];
        }];
    }]];
}

-(NSManagedObjectContext*) viewContext{
    if (![NSThread mainThread]) {
        //here you should log to you analytics service. If you are in developer mode you
should crash to force you to fix this
        NSLog(@"access context on wrong thread!!");
    }
    return self.persistentContainer.viewContext;
}
```

Swift 2 Example

```
// Core Data stack
```

```
lazy var applicationDocumentsDirectory: NSURL = {
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains:
.UserDomainMask)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    let modelURL = NSBundle.mainBundle().URLForResource("ProjectName", withExtension: "momd")!
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {

    let coordinator = NSPersistentStoreCoordinator(managedObjectModel:
self.managedObjectModel)
    let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading the application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL:
url, options: nil)
    } catch {
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        print("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }

    return coordinator
}()

lazy var managedObjectContext: NSManagedObjectContext = {
    let coordinator = self.persistentStoreCoordinator
    var managedObjectContext = NSManagedObjectContext(concurrencyType:
.MainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()

// Core Data Saving support

func saveContext () {
    if managedObjectContext.hasChanges {
        do {
            try managedObjectContext.save()
        } catch {
            let nserror = error as NSError
            print("Unresolved error \(nserror), \(nserror.userInfo)")
            abort()
        }
    }
}
```

iOS 10 Example in Swift

```
lazy var persistentContainer: NSPersistentContainer = {

    let container = NSPersistentContainer(name: "ProjectName")
    container.loadPersistentStores(completionHandler: { (storeDescription, error) in

        if let error = error {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

func saveContext () {
    let context = persistentContainer.viewContext

    do {
        try context.save()
    } catch {
        let nserror = error as NSError
        fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
    }

    if context.hasChanges {
        print("changes occured")
    }else {
        print("no changes occured")
    }

}
```

Read Core Data Stack online: https://riptutorial.com/core-data/topic/2596/core-data-stack

Chapter 3: Creating an Core Data Model

Remarks

Attribute types include: Undefined, Integer 16, Integer 32, Integer 64, Decimal, Double, Float, String, Boolean, Date, Binary, Data, or Transformable

When defining an `Entity` as abstract you won't be creating any instances of that entity. For example a Person would be abstract and a Employee or Customer would be a concrete subentities.

`Transient` attributes are properties that you define as part of the model, but which are not saved to the persistent store as part of an entity instance’s data. Core Data does track changes you make to transient properties, so they are recorded for undo operations. You use transient properties for a variety of purposes, including keeping calculated values and derived values.

The `Destination` field defines what object (or objects) are returned when the relationship is accessed in code.

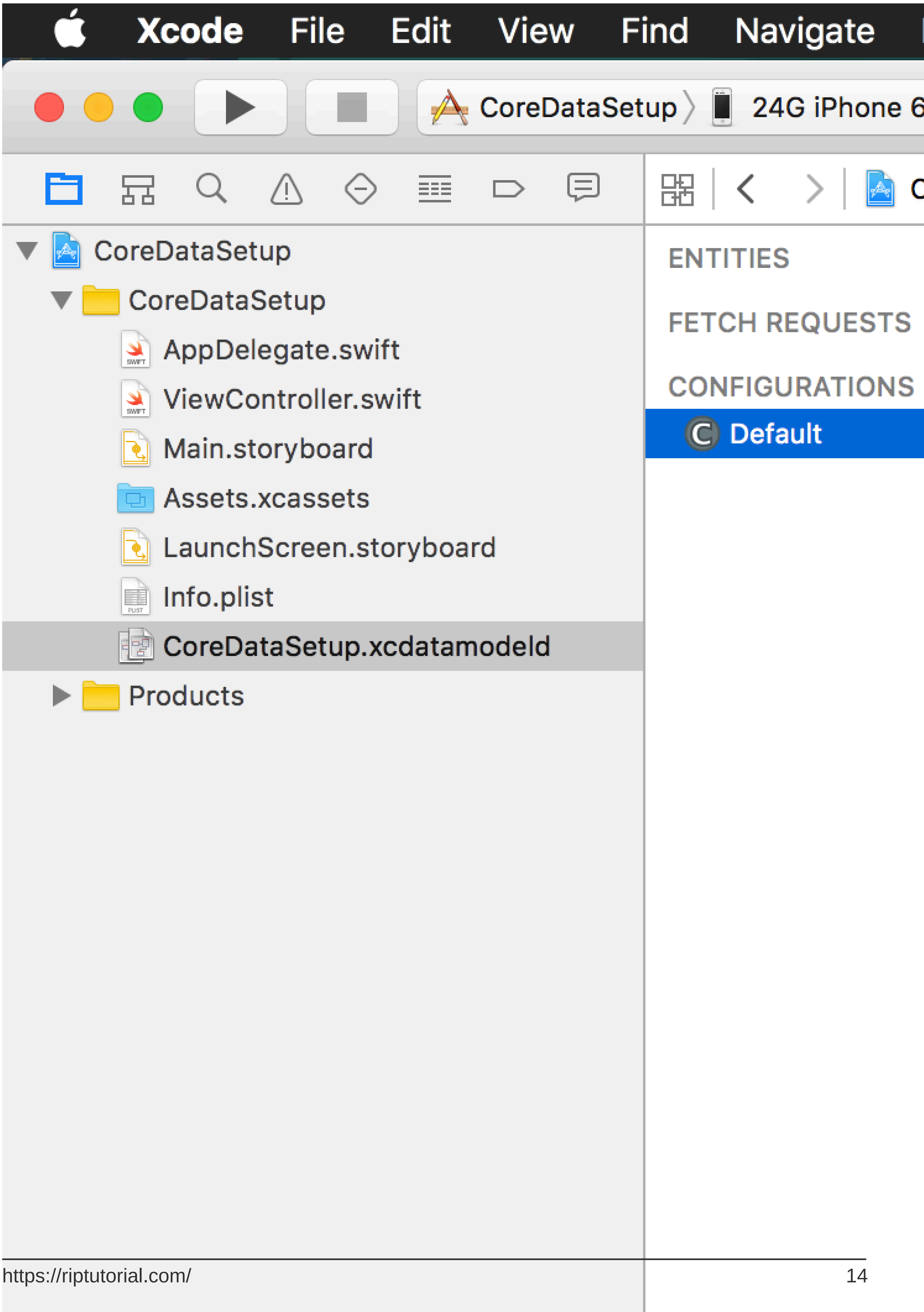
The `Inverse` field defines the other half of a relationship. Because each relationship is defined from one direction, this field joins two relationships together to create a fully bidirectional relationship.

Source: Core Data Programming Guide

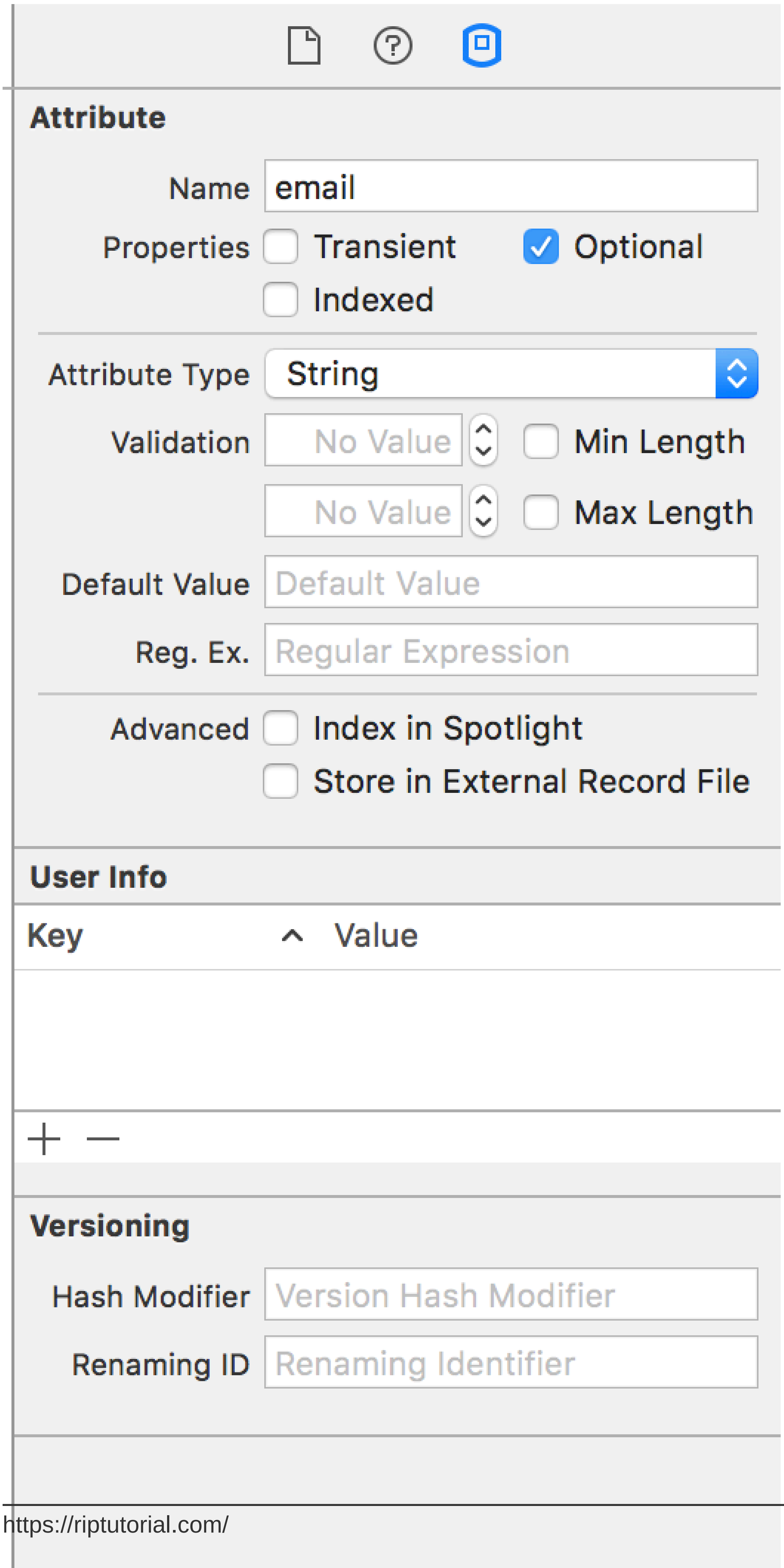
Examples

Adding an Entity to Core Data Model

1. First it is important to understand that the Core Data Model is the `*.xcdatamodeld` file. You will notice you have not entities. You will have to create one yourself. At the bottom of Xcode you will notice a button that says "Add Entity" click it and you will have a new entity in the navigator area for you to work with on the project.


Xcode
File
Edit
View
Find
Navigate
CoreDataSetup
24G iPhone 6
CoreDataSetup
CoreDataSetup
AppDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
CoreDataSetup.xcdatamodeld
Products
ENTITIES
FETCH REQUESTS
CONFIGURATIONS
Default

for example if you were adding an email you could provide a regex string `".+@([A-Za-z0-9-]+\\.)+[A-Za-z]{2}[A-Za-z]*"` to prevent postal addresses from being added to your email Attribute. Validation could allow for a min and max character for a phone number.



Attribute

Name: email

Properties: [ ] Transient [✓ Optional] [ ] Indexed

Attribute Type: String

Validation: No Value [ ] Min Length

No Value [ ] Max Length

Default Value: Default Value

Reg. Ex.: Regular Expression

Advanced: [ ] Index in Spotlight [ ] Store in External Record File

User Info

Key	Value

+ —

Versioning

Hash Modifier: Version Hash Modifier

Renaming ID: Renaming Identifier

https://riptutorial.com/core-data/topic/2853/creating-an-core-data-model

Chapter 4: NSFetchedResultsController

Introduction

NSFetchedResultsController is a connection between core-data table (entity in core-data, table in sqlite) and UITableView. UITableView can be attached to any core-data entity using NSFetchedResultsController and UITableView will be updated as and when core-data updates that entity/table.

Examples

NSFetchedResultsController for UITableView

```
class ConversationsTableViewController: UITableViewController,
NSFetchedResultsControllerDelegate {

private var fetchedResultsController: NSFetchedResultsController<Conversation>!

override func viewDidLoad() {
    super.viewDidLoad()
    initializeFetchedResultsController()
}
private func initializeFetchedResultsController() {
    let request = NSFetchRequest<Conversation>(entityName: "Conversation")
    let timeSort = NSSortDescriptor(key: "lastMessageTime", ascending: false)
    request.sortDescriptors = [timeSort]

    let MOC = AppManagedObjectContext()//It should be main thread MOC
    fetchedResultsController = NSFetchedResultsController(fetchRequest: request,
managedObjectContext: MOC, sectionNameKeyPath: nil, cacheName: nil)
    fetchedResultsController.delegate = self

    do {
        try fetchedResultsController.performFetch()
    } catch {
        print("Failed to initialize FetchedResultsController: \(error)")
    }
}

//table view methods
override func numberOfSections(in tableView: UITableView) -> Int {
    if let n = fetchedResultsController?.sections!.count {
        return n
    }
    return 0
}
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    let sectionInfo = fetchedResultsController.sections![section]
    let n =  sectionInfo.numberOfObjects
    return n
}
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "identifier", for: indexPath)
    //configure cell
    return cell
}

//NSFetchedResultsController Delegates
func controllerWillChangeContent(_ controller:
NSFetchedResultsController<NSFetchRequestResult>) {
    tableView.beginUpdates()
}

func controller(_ controller: NSFetchedResultsController<NSFetchRequestResult>, didChange
sectionInfo: NSFetchedResultsSectionInfo, atSectionIndex sectionIndex: Int, for type:
NSFetchedResultsChangeType) {
    switch type {
    case .insert:
        tableView.insertSections(IndexSet(integer: sectionIndex), with: .fade)
    case .delete:
        tableView.deleteSections(IndexSet(integer: sectionIndex), with: .fade)
    case .move:
        break
    case .update:
        break
    }
}

func controller(_ controller: NSFetchedResultsController<NSFetchRequestResult>, didChange
anObject: Any, at indexPath: IndexPath?, for type: NSFetchedResultsChangeType, newIndexPath:
IndexPath?) {
    switch type {
    case .insert:
        tableView.insertRows(at: [newIndexPath!], with: .fade)
    case .delete:
        tableView.deleteRows(at: [indexPath!], with: .fade)
    case .update:
        tableView.reloadRows(at: [indexPath!], with: .none)
    case .move:
        tableView.deleteRows(at: [indexPath!], with: .fade)
        tableView.insertRows(at: [newIndexPath!], with: .fade)
    }
}

func controllerDidChangeContent(_ controller:
NSFetchedResultsController<NSFetchRequestResult>) {
    tableView.endUpdates()
}
}
```

Read NSFetchedResultsController online: https://riptutorial.com/core-data/topic/9985/nsfetchedresultscontroller

Chapter 5: Sort Descriptors

Examples

Ordering Data Returned By Fetch Requests

Set the NSFetchRequest property sortDescriptors to determine how data is returned.

```
let fetchRequest = NSFetchRequest(entityName: "NAME_OF_ENTITY")
let sortDescriptor = NSSortDescriptor(key: "NAME_OF_ATTRIBUTE", ascending: true)
fetchRequest.sortDescriptors = [sortDescriptor]
```

Multiple Sort Descriptors

You can also set multiple sort descriptors, to sort by one attribute within another. For example, return all entries ordered by date, and by name within each date:

```
let fetchRequest = NSFetchRequest(entityName: "NAME_OF_ENTITY")
let sortDescriptor1 = NSSortDescriptor(key: "name", ascending: true)
let sortDescriptor2 = NSSortDescriptor(key: "date", ascending: true)
fetchRequest.sortDescriptors = [sortDescriptor1, sortDescriptor2]
```

Read Sort Descriptors online: https://riptutorial.com/core-data/topic/5971/sort-descriptors

Chapter 6: Using Predicates

Examples

Matching an exact string

```
let fetchRequest = NSFetchRequest(entityName: "Foo")
var thePredicate: NSPredicate?
thePredicate = NSPredicate(format: "message == 'example'")
```

> The entity `Foo` has a `message` string attribute

Substitutions

Rather than passing a static string as a predicate's criteria. It is possible to substitute values by using format specifiers. There are five format specifiers:

- `%K` is a var arg substitution for a key path.
- `%@` is a var arg substitution for an object value-often a string, number, date, or an array.
- `%ld` is a var arg substitution for an int value.
- `%la` is a var arg substitution for a double.
- `%a` is a var arg substitution for a float.

In the following example, the `%K` format specifier serves as the left-hand argument which passes in the *"message"* property dynamically. The `%@` format specifier serves as the right-hand argument to dynamically pass in a string containing the word *"example"*.

```
let predicate = NSPredicate(format:"%K == %@", "message", "example")
```

Read Using Predicates online: https://riptutorial.com/core-data/topic/4283/using-predicates

Credits

S. No	Chapters	Contributors
1	Getting started with core-data	Asdrubal, Community, Josh Caswell, rgeorge, Santa Claus, TMob
2	Core Data Stack	Asdrubal, Jon Rose, Santa Claus
3	Creating an Core Data Model	Asdrubal
4	NSFetchedResultsController	D4ttatraya
5	Sort Descriptors	Jonas
6	Using Predicates	Dan Beaulieu, David