Core Data

iOS App Development
Fall 2010 — Lecture 12

Questions?

Announcements

- Assignment #3 out earlier today
  - Due Monday, October 18th by 11:59pm
- Select order of presenters for next week

Today’s Topics

- SQLite
- What is Core Data?
- Core Data stack
- Xcode modeling tool
- Implementing CRUD operations...
  - Create
  - Read
  - Update
  - Delete
- Odds & Ends

Notes

- I'm showing the relevant portions of the view controller interfaces and implementations in these notes
- Remember to release relevant memory in the -dealloc methods — they are not shown here
- You will also need to wire up outlets and actions in IB
- Where delegates or data sources are used, they too require wiring in IB

SQLite

What is SQLite?

- “SQLite is a in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine”
- See the SQLite website for additional details...
  - http://www.sqlite.org/

SQLite on the iPhone

- Prior to the iPhone 3.0 SDK, SQLite was the route that many developers took for more complex, transactional storage of data
- However, the 3.0 SDK brought Core Data support to iOS,
  - You may want to consider Core Data — especially if you have complex relationships
- But, if you'd rather go the SQL route over Core Data it is still a viable option
  - Just be aware that the APIs are fairly low-level C

What is Core Data?

What is Core Data?

- Core Data is a schema-driven object graph management and persistence framework
- Core Data helps you to save model objects (in the sense of the MVC design pattern) to a file and get them back again
- Typically store data into a database SQLite, but you can also use other backends or even provide your own

Core Data Features

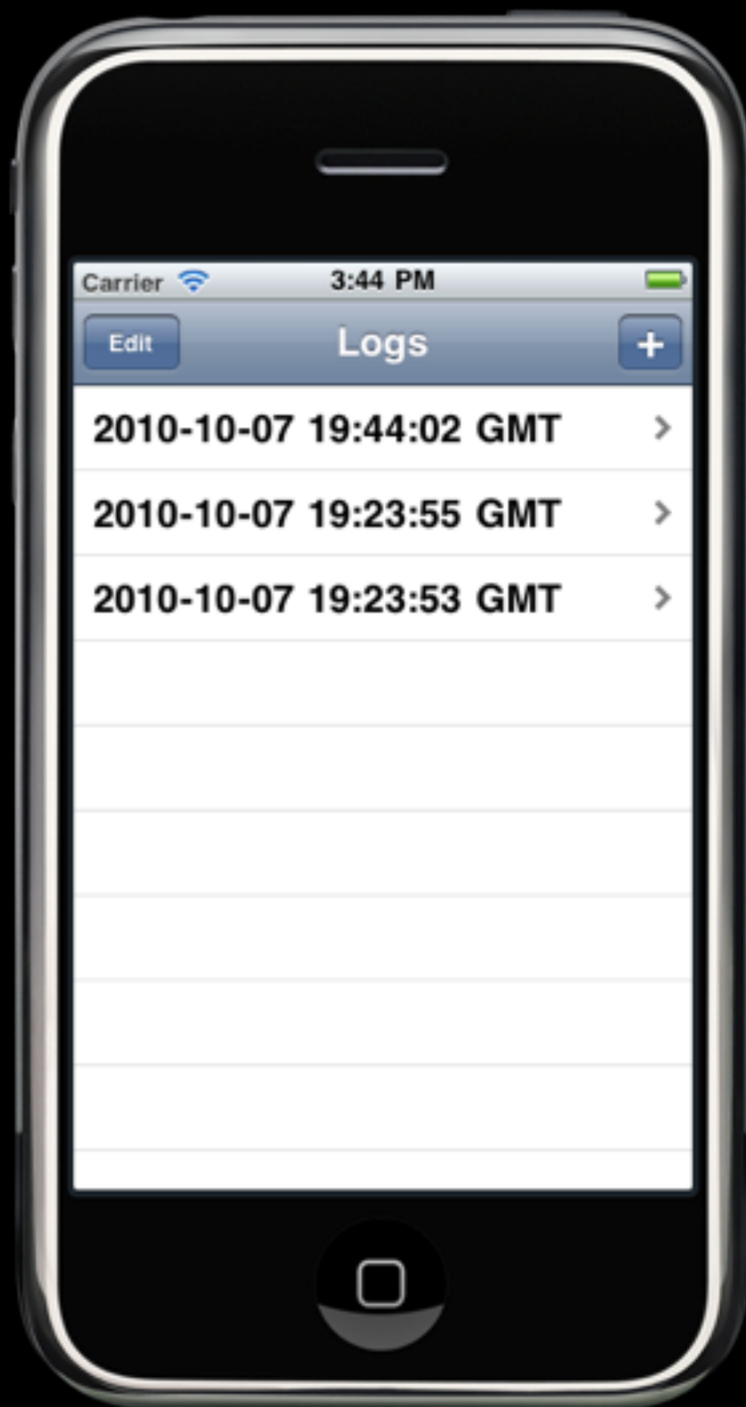
- Infrastructure for managing changes to your model objects
- Support for undo and redo
- Support for maintaining relationships between objects
- Allows you to keep a subset of your objects in memory
  - Important on iOS where conserving memory is critical
- Schema to describe model, defined using a GUI-based editor
- Infrastructure for data store versioning and migration
  - Allows easy upgrades from an old to the current version

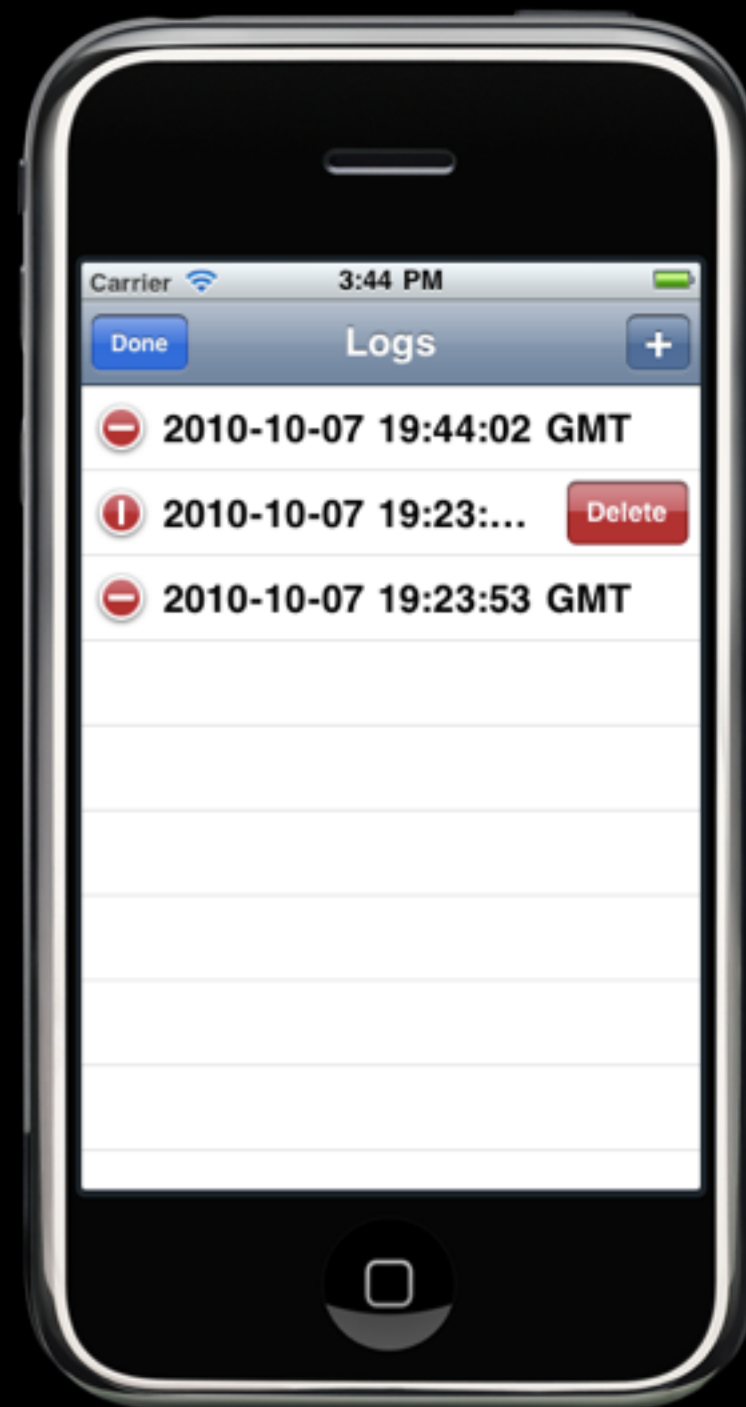
Example

What We're Going to Build

- A simple app that allows us to create a running list of logs
- The user will be able to...
  - Add a log (automatically recording the timestamp) to the logs
  - Edit a log (be able to add or change a note) within the log
  - Delete a specific log from a collection of logs
  - Be able to view the list of existing logs

What We're Going to Build



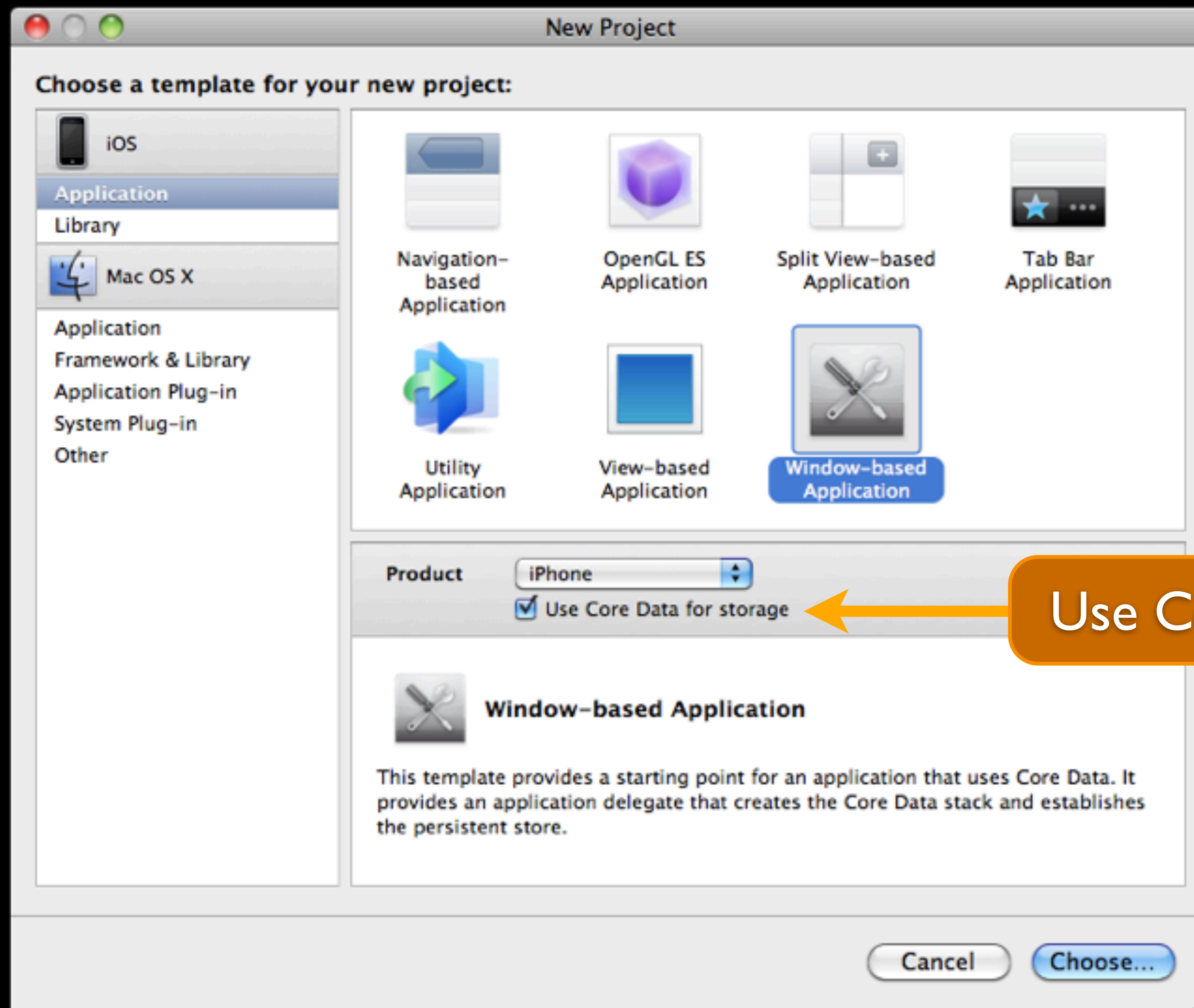


Getting Started

Window-Based Application

- For this example, we're going to use a Window-based application that utilized Core Data
- Why a window-based app?
  - Gives us a chance to look at another project template
  - It's one of the templates that you can choose to use Core Data with by simply checking a box on the new project template screen

New Window-based App using Core Data



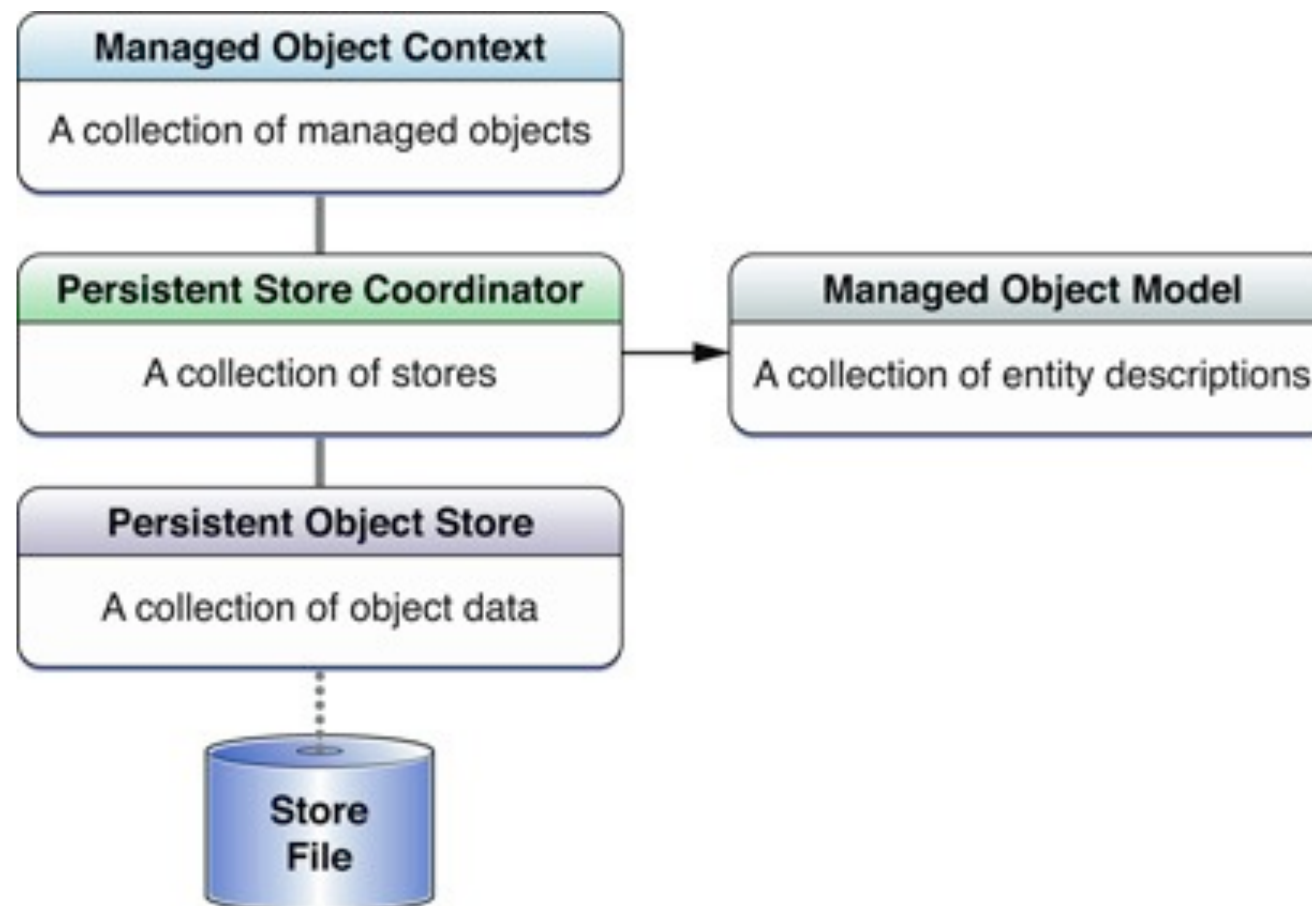
Use Core Data

LogsAppDelegate.h

```
#import <UIKit/UIKit.h>
#import <CoreData/CoreData.h>

@interface LogsAppDelegate : NSObject <UIApplicationDelegate> {

    UIWindow *window;

@private
    NSManagedObjectContext *managedObjectContext_;
    NSManagedObjectModel *managedObjectModel_;
    NSPersistentStoreCoordinator *persistentStoreCoordinator_;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

- (NSString *)applicationDocumentsDirectory;
- (void)saveContext;

@end
```

Core Data Stack

Core Data Stack



Managed Objects

- A managed object is an instance of NSManagedObject
- It's typically an object representation of a record in a table of a database
- Managed objects represent the data you operate on in your application
- A managed object is always associated with a managed object context

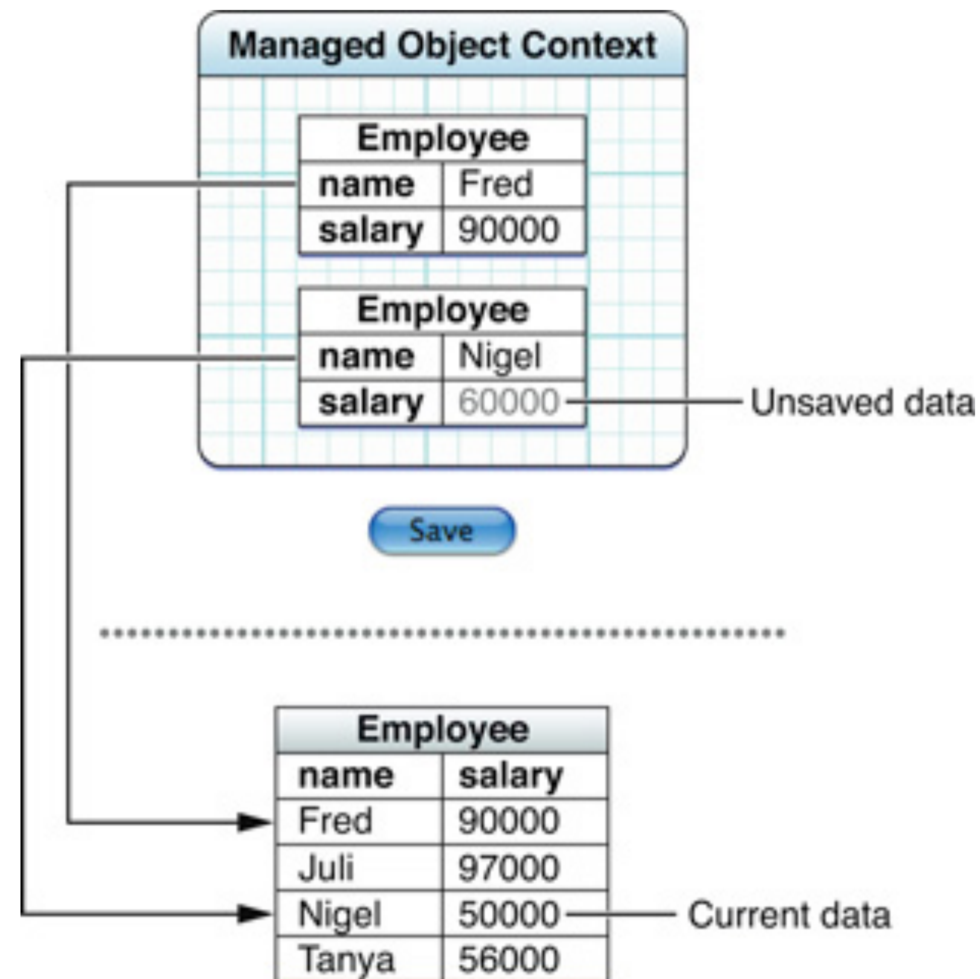
NSManagedObjectContext

- The managed object context is an instance of NSManagedObjectContext
- Think of this context as a scratch pad for an application
- Its primary responsibility is to manage a collection of managed model objects
- These objects form a group of related model objects that represent an internally consistent view of one or more persistent stores
- The context is a powerful object with a central role in your application, with responsibilities from life-cycle management to validation, relationship maintenance, and undo/redo

Using Contexts

- Newly created managed objects are inserted into a context
- Records are fetched from a database into a context as managed objects
- All changes you make (inserts, deletes, updates) are kept in memory
- You must save the context to commit the managed objects to the underlying datastore

Context to Database Mapping



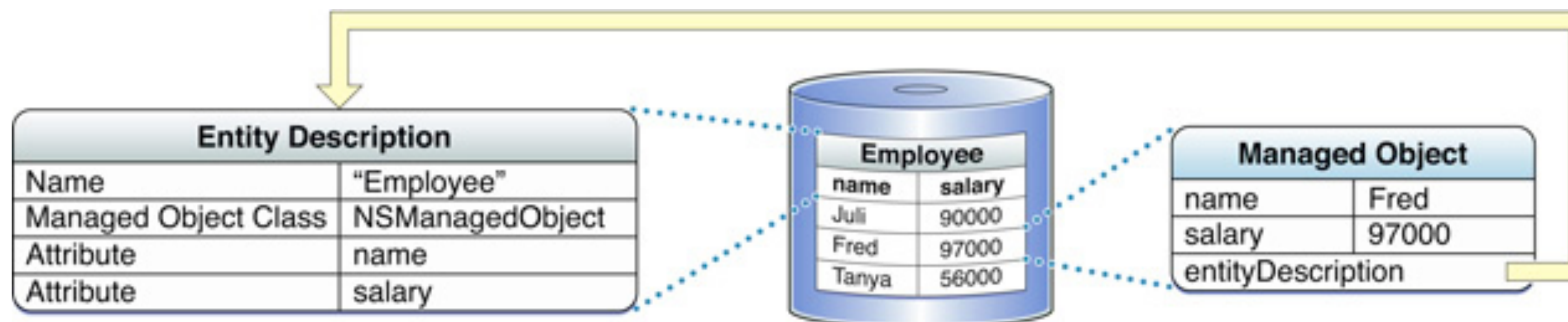
NSManagedObjectModel

- A managed object model is an instance of NSManagedObjectModel
- Object representation of a schema that describes your database, and the managed objects you use in your app
- A model is a collection of entity description objects (instances of NSEntityDescription)

NSEntityDescription

- An entity description describes an entity consisting of the following...
  - The name of a table in a database where instances of your object will be persisted
  - The name of the class used to represent the entity in your application
  - The properties (attributes and relationships) an object has

Entity Description Mapping



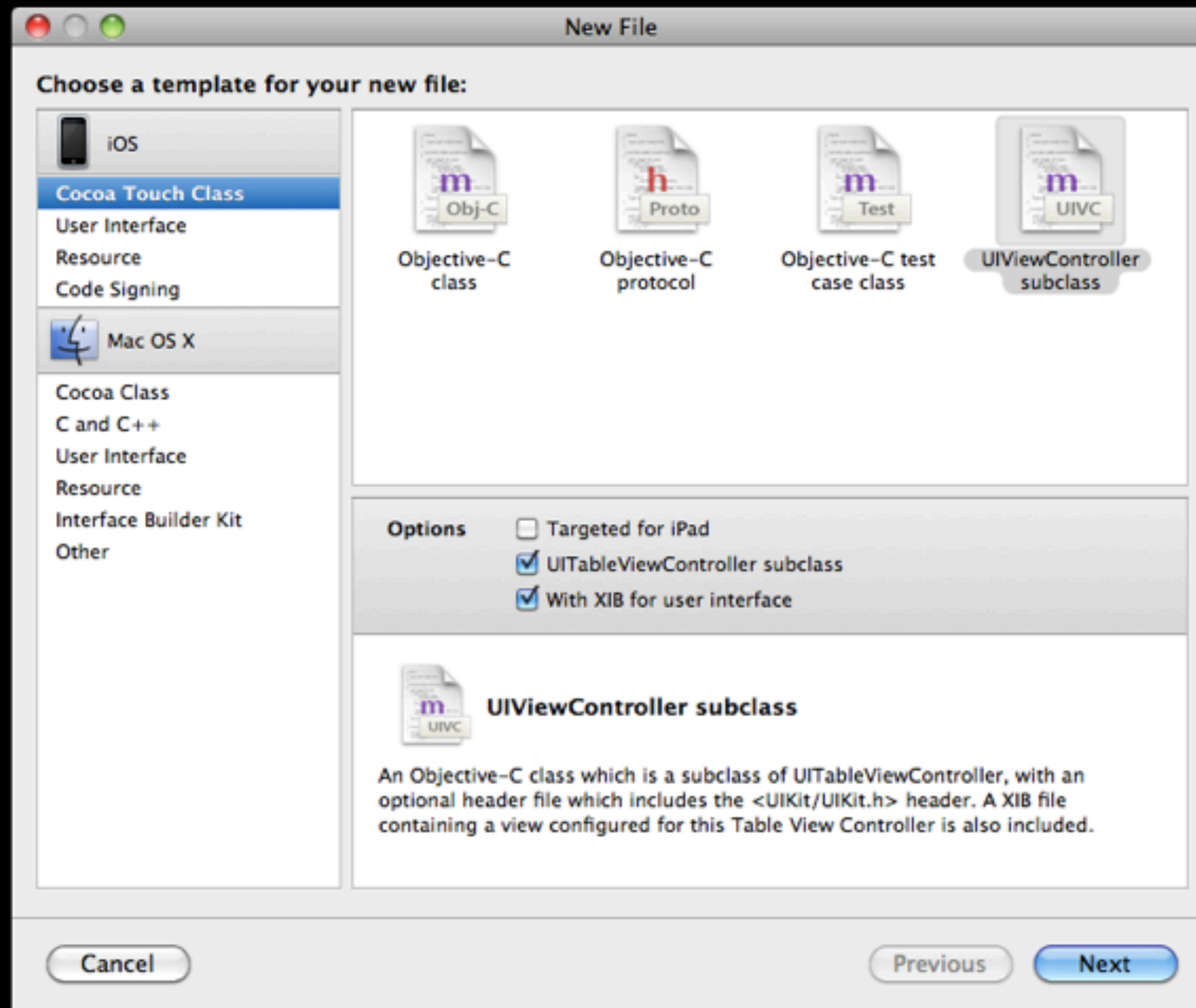
NSPersistentStoreCoordinator

- The persistent store coordinator plays a central role in how Core Data manages data
  - However, you don't often interact with it directly
- It manages a collection of persistent object stores
  - Though, on iOS we typically use a single store
- A persistent object store represents an external store (file) of persisted data
- It's the object that actually maps between objects in your application and records in the database

Creating the Root Controller

Creating the Root View Controller

- Right click Classes → Add → New File...



Designing our Root View Controller

- We'll create properties to store the following in this class...
  - An array of logs — used to back the table view
  - A managedObjectContext — used for persistence

RootViewController.h

```
#import <UIKit/UIKit.h>

@interface RootViewController : UITableViewController {

}

@property (nonatomic, retain) NSMutableArray *logs;
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;

@end
```

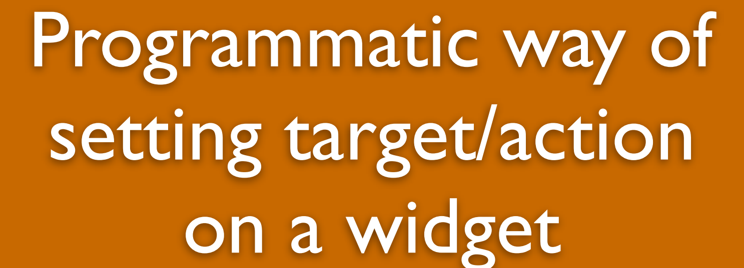
RootViewController.m

```
#import "RootViewController.h"

@implementation RootViewController

@synthesize logs;
@synthesize managedObjectContext;

/* ... */

@end
```

Configuring the Root View Controller

- We need to add code to...
  - Set the title on the view
  - Add an edit button
    - We can leverage the built in edit behaviors
  - Add an add button
    - We'll need to eventually provide our own add action

RootViewController.m




```
/* ... */

- (void)viewDidLoad {
  [super viewDidLoad];
  self.title = @"Logs";
  self.navigationItem.leftBarButtonItem = self.editButtonItem;

  UIBarButtonItem *addButton = [[[UIBarButtonItem alloc]
                                 initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                                target:self
                                action:@selector(addLog)] autorelease];
    self.navigationItem.rightBarButtonItem = addButton;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
                                    numberOfRowsInSection:(NSInteger)section {
    return 0;
}

/* ... */
```

Telling the App Delegate to Load the Root VC

- Next, we need to configure the app's delegate to load and configure the root controller as the initial view for the app
- We'll need to do the following in the app delegate...
  - Create a property to store a navigation view controller (which will house our Root VC)
  - Create an actual instance for the navigation VC
  - Instantiate an instance of the Root VC
  - Add the Root VC to the navigation VC
  - Set the Root VC's instance of NSManagedObjectContext

LogsAppDelegate.h

```
#import <UIKit/UIKit.h>
#import <CoreData/CoreData.h>

@interface LogsAppDelegate : NSObject <UIApplicationDelegate> {

    UIWindow *window;

@private
    NSManagedObjectContext *managedObjectContext_;
    NSManagedObjectModel *managedObjectModel_;
    NSPersistentStoreCoordinator *persistentStoreCoordinator_;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

@property (nonatomic, retain) UINavigationController *navigationController;

- (NSString *)applicationDocumentsDirectory;
- (void)saveContext;

@end
```

LogsAppDelegate.m

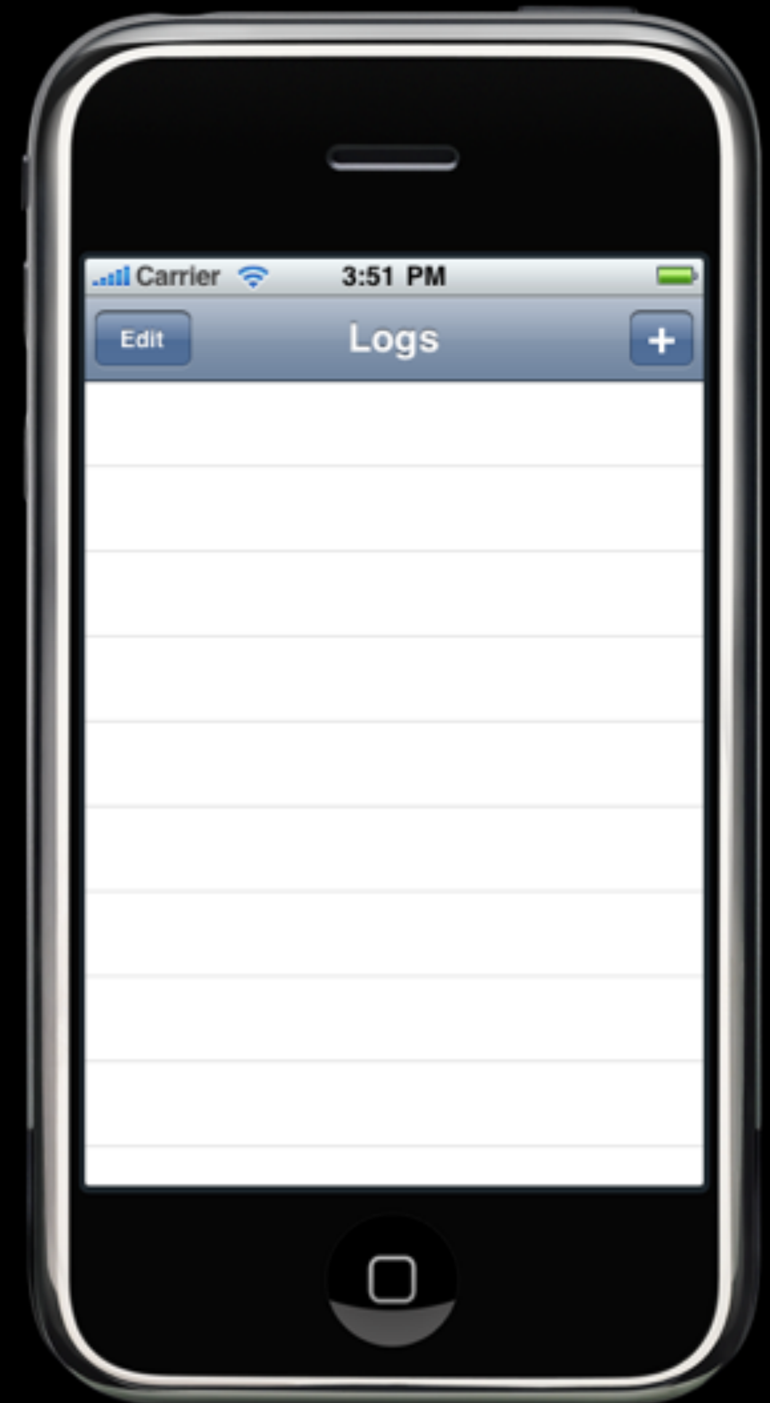
```
#import "LogsAppDelegate.h"
#import "RootViewController.h"

@implementation LogsAppDelegate

@synthesize window;
@synthesize navigationController;

/* ... */
```

LogsAppDelegate.m

```
/* ... */
- (BOOL)application:(UIApplication *)application
                    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

  RootViewController *rootViewController = [[RootViewController alloc]
                                 initWithNibName:@"RootViewController" bundle:nil];

  NSManagedObjectContext *context = [self managedObjectContext];
  if (!context) { NSLog(@"Major Error"); }

  rootViewController.managedObjectContext = context;
  UINavigationController *aNavigationController = [[UINavigationController alloc]
                                   initWithRootViewController:rootViewController];

  self.navigationController = aNavigationController;

  [window addSubview:[navigationController view]];
  [window makeKeyAndVisible];

  [rootViewController release];
  [aNavigationController release];

  return YES;
}
/* ... */
```

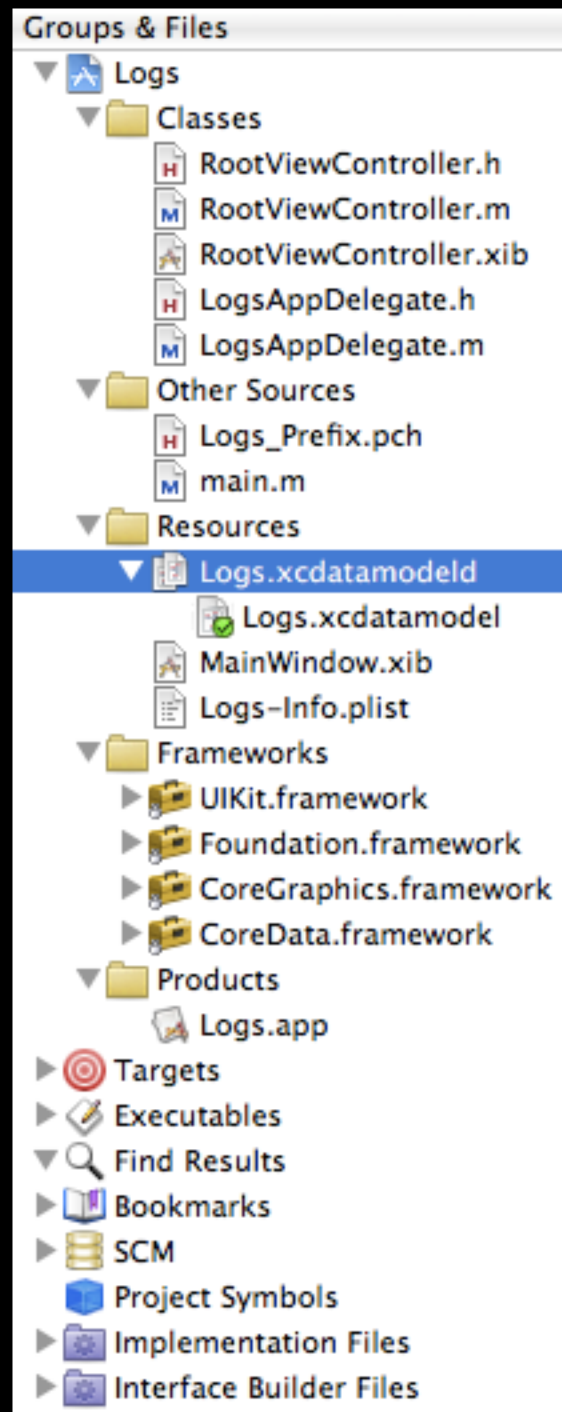
Our App Thus Far

- At this point we have a basic navigation based app that we created ourselves
- We have a root VC that has a managed object context that will be used for persistence
- We have an add button that will currently crash the app, as there's no -addLog method ...yet



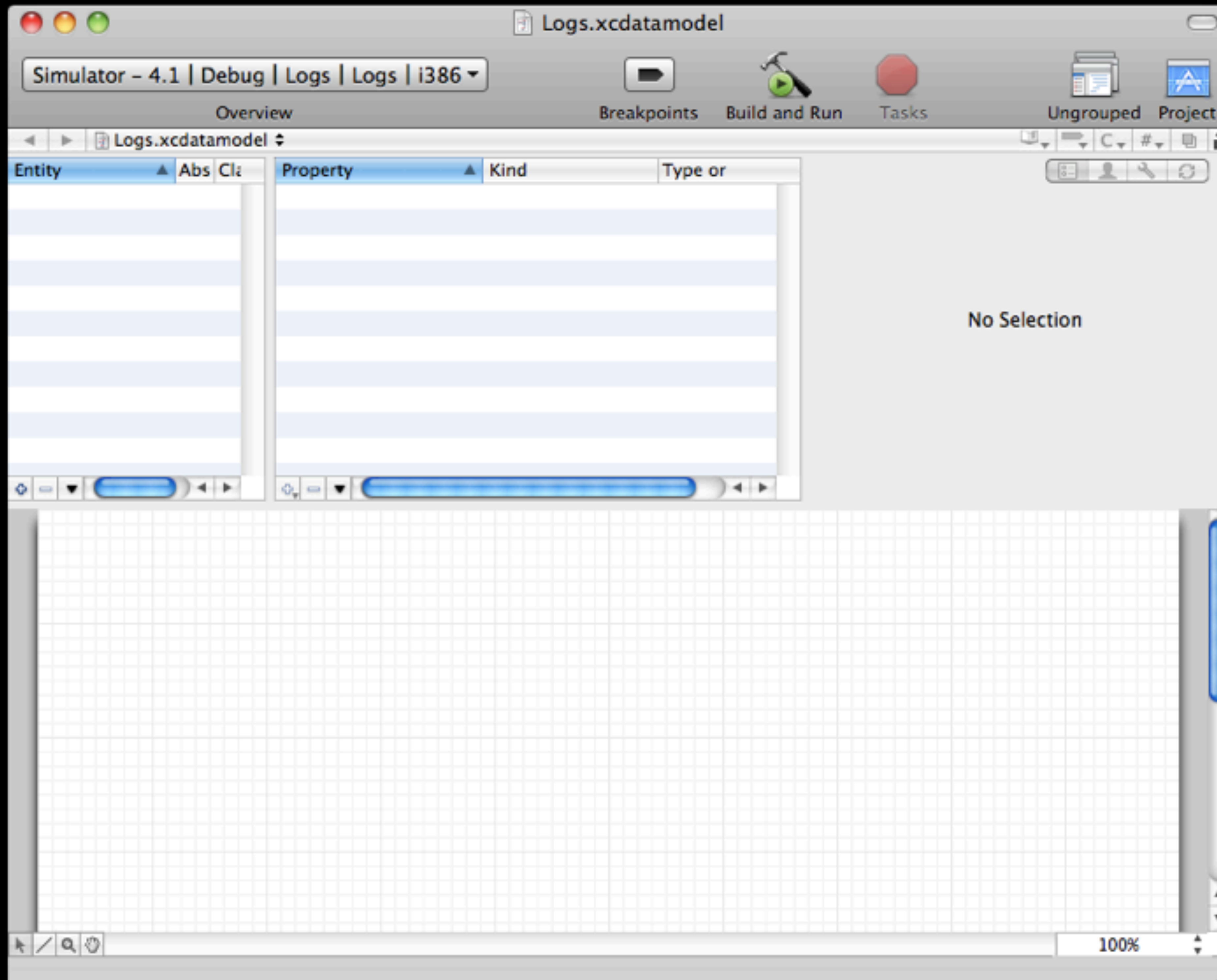
Modeling Our Data

Modeling Our Data



- The next step in the process is to use Xcode's modeling tool to design our managed objects
- If you look under Groups & Files, you'll notice some additional things...
  - Core Data has been added to the list of Frameworks for this project
  - There's also a Logs.xcdatamodel file under resources — opening this file brings up Xcode's data modeling tool

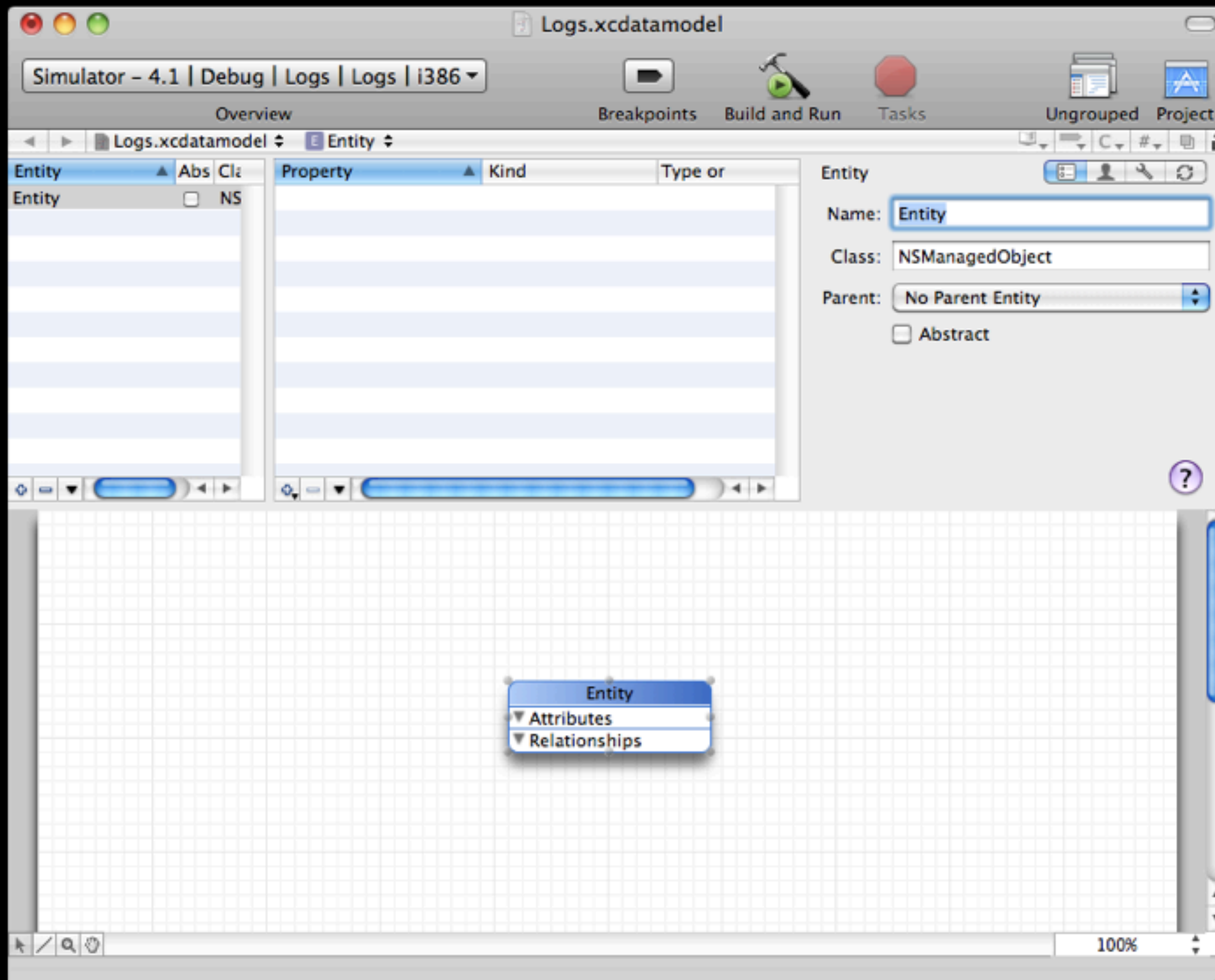
The Xcode Data Modeling Tool



Adding an Entity

- Entities represent the managed objects of our model and typically correspond to...
  - Classes in ObjC
  - Tables in a database
- To add a new Entity, right click on the bottom section and select Add Entity

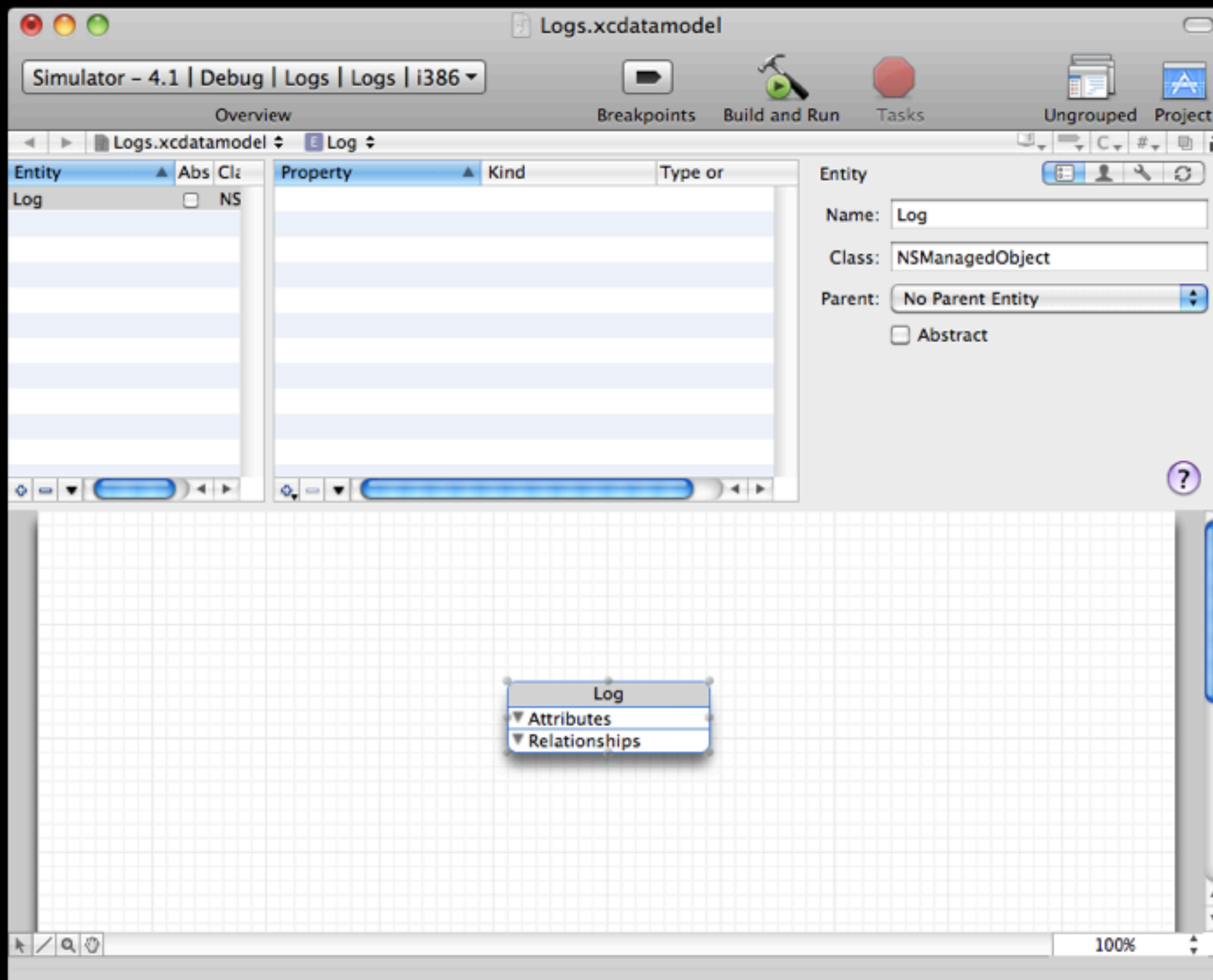
The Added Entity



Renaming the Entity

- Once added, change the name of the entity to Log
- Note that the class backing this object is an instance of NSManagedObject

The Renamed Entity



Adding Attributes

- Attributes represent the fields of the managed objects of our model and typically correspond to...
  - Properties of classes in ObjC
  - Columns of a table in a database
- To add a new Attribute, select the entity object (in the bottom section), then right click on it and select Add Attribute

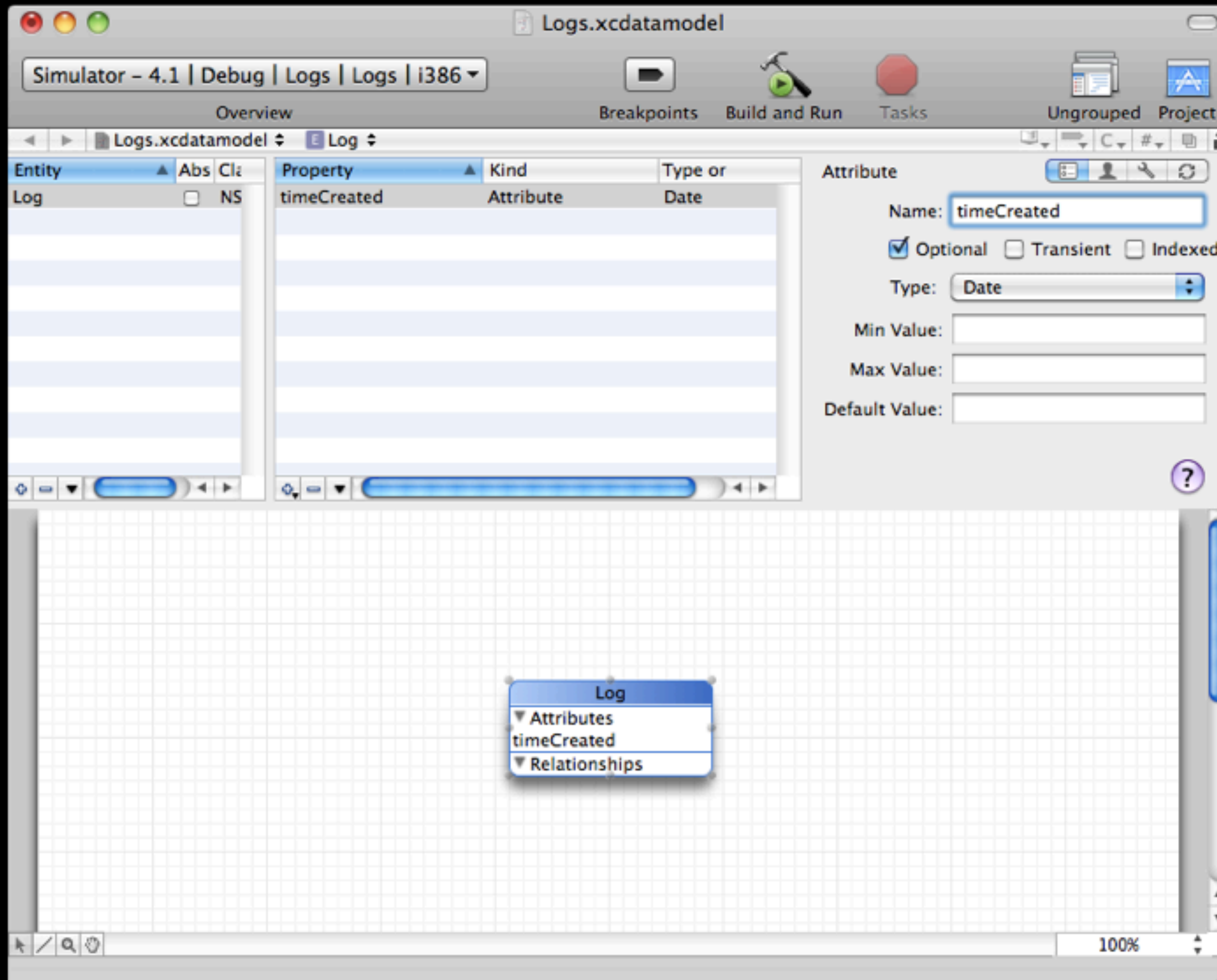
Attributes

- When we add an attribute there are values that we'll always set...
  - Name — the name of the attribute (field) in the model
  - Type — the type of the attribute in the abstract
- Once you set the type there are additional constraints that can be set based on the type, such as...
  - A required matching regex for strings
  - Minimum and maximum values for most numerical types

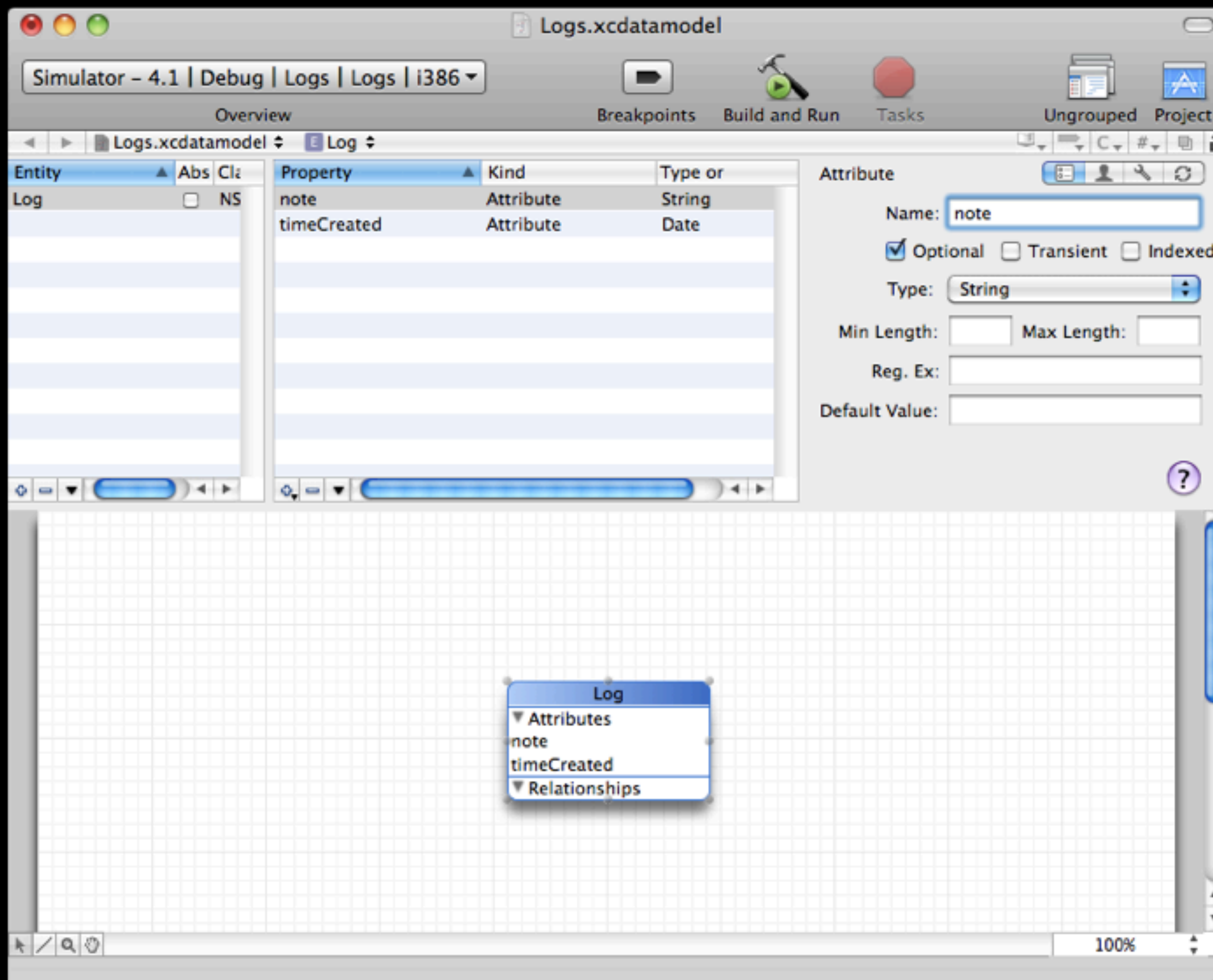
Adding Our Attributes

- For our application, we'll add the following attributes...
  - timeCreated of type Date
  - note of type String

The Added timeCreate Attribute



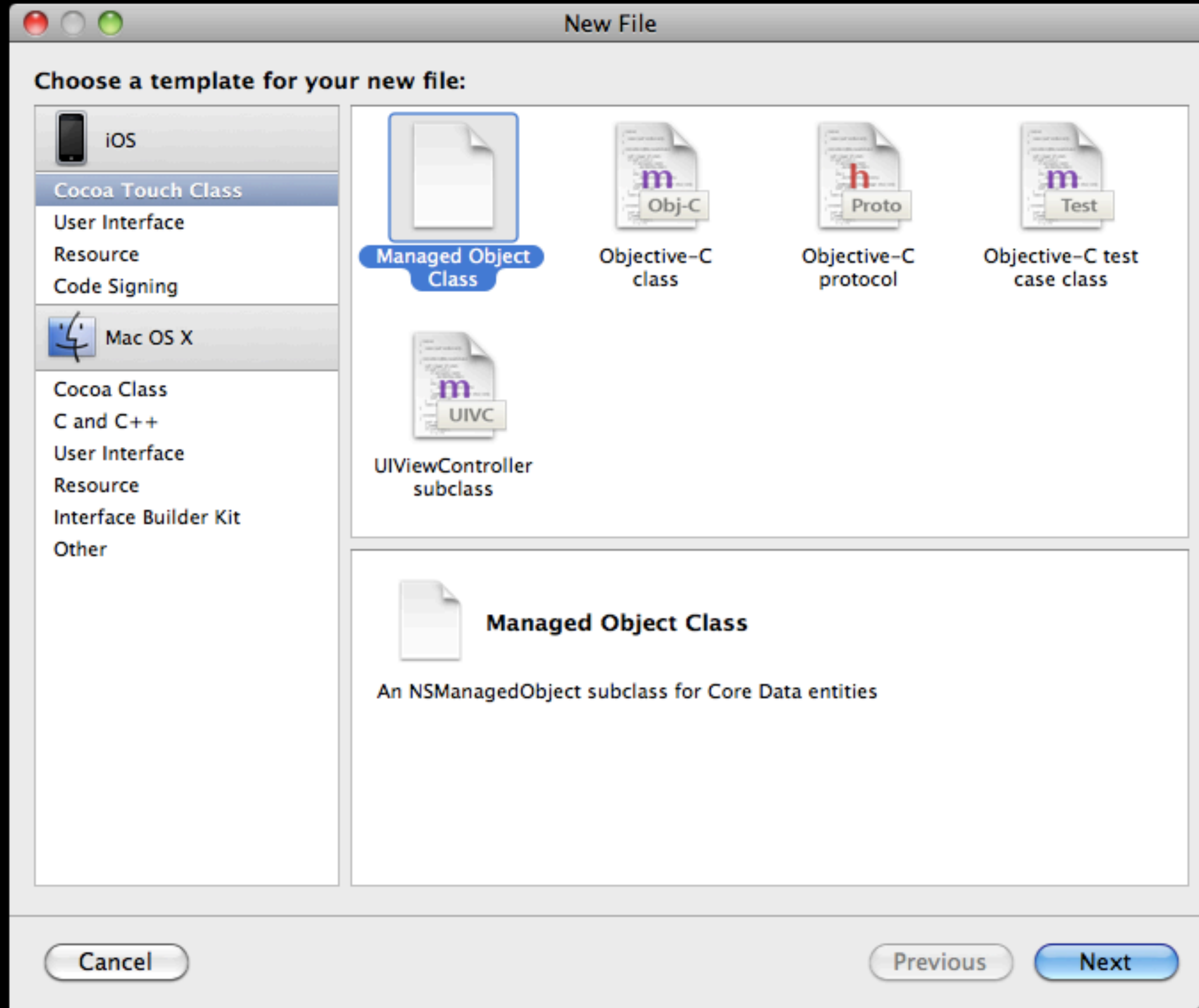
The Added note Attribute



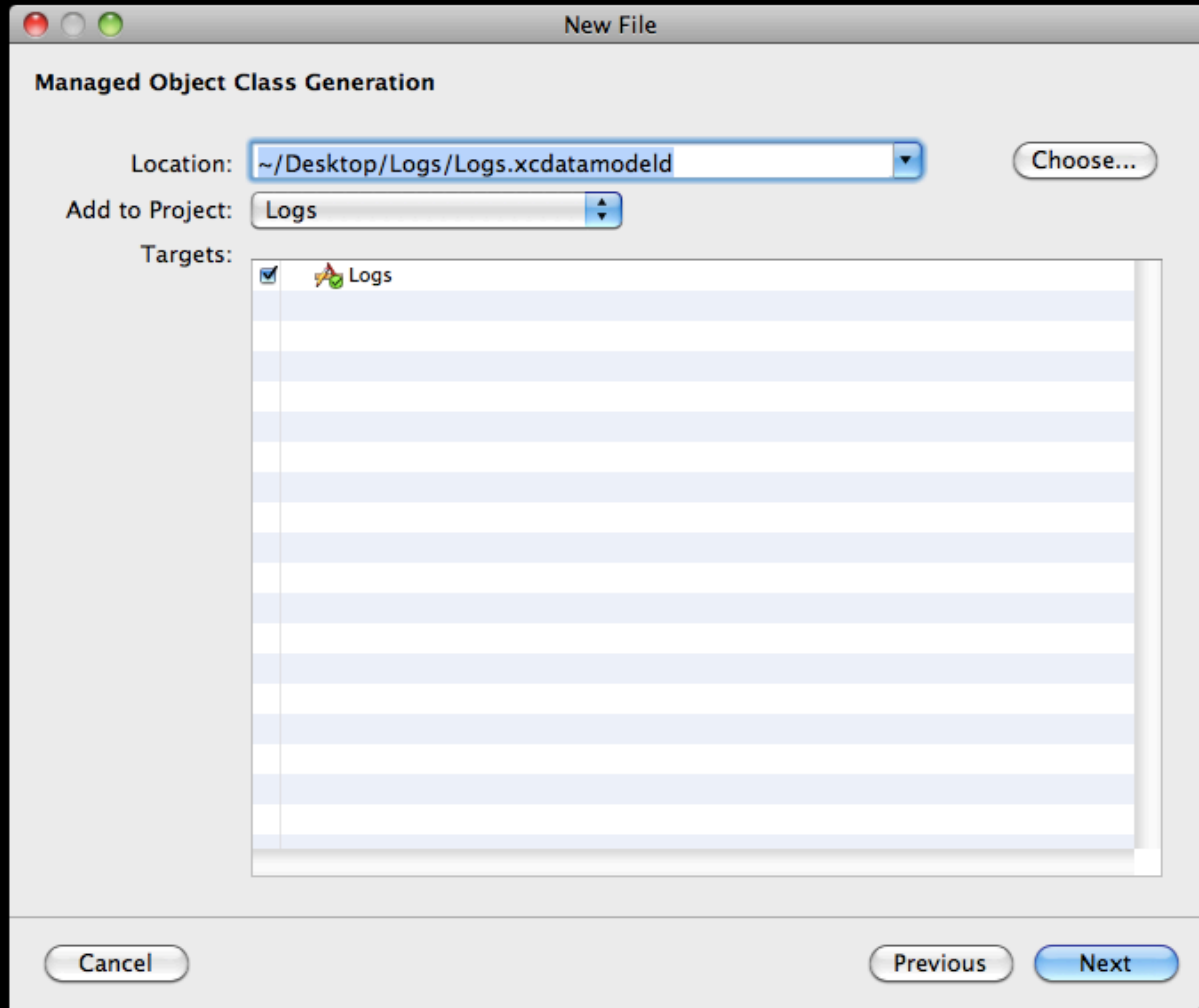
Generating the Model Objects

- The next step in our process is going to be to create the actual classes we'll use in ObjC to represent these objects
- Xcode can actually create these managed classes for us
- From the File menu, select New File...
  - Then select new Managed Object Class

Creating a New Managed Object Class



Choose a Location



Selecting Entities

- Next, we tell Xcode what entities we want classes for
- In our simple app, we only have a single entity, so we'll make sure that it is checked
- Also, Xcode gives us the option to generate accessors and properties for the object — verify they are checked

Select Entities and Options

New File

Managed Object Class Generation

Select	Entity	Class
☑	Log	Log

☑ Generate accessors
☑ Generate Obj-C 2.0 Properties
☐ Generate validation methods

Cancel | Previous | Finish

Log.h

```
#import <CoreData/CoreData.h>


@interface Log :  NSManagedObject
{
}

@property (nonatomic, retain) NSString * note;
@property (nonatomic, retain) NSDate * timeCreated;

@end
```

Log.m

```
#import "Log.h"


@implementation Log

@dynamic note;
@dynamic timeCreated;

@end
```

Observations

- The Log class is an instance of NSManagedObject and not NSObject
  - This means that Core Data is going to manage instances of this class
- Our attributes make up the properties of the class
- Properties are specified as @dynamic
  - Core Data generates the accessors at runtime, not compile time
- No -dealloc to release properties
  - Not an oversight, CoreData manages all aspects of this object's lifecycle

Adding Logs

Adding Logs

- There's a couple of things we need to add to the RootViewController to add logs...
  - First, we need to import Log.h
  - Implement the -addLog method
    - Create a new Log through Core Data
    - Configure the new Log
    - Save the new Log through Core Data
    - Insert log into logs array
    - Update the table

Adding Logs

- To create a record we tell the managed object context to create a new Log instance using the following NSEntityDescription method...

```
+ (id)insertNewObjectForEntityForName:(NSString *)entityName
               inManagedObjectContext:(NSManagedObjectContext *)context;
```

- Then to actually persist the record, you commit the action using the following NSManagedObjectContext method...

```
- (BOOL)save:(NSError **)error;
```

RootViewController.m

```
#import "RootViewController.h"
#import "Log.h"

/* ... */

- (void)addLog {
  Log *log = [NSEntityDescription insertNewObjectForEntityForName:@"Log"
                                  inManagedObjectContext:self.managedObjectContext];

  log.timeCreated = [NSDate date];

  NSError *error;
  if (![self.managedObjectContext save:&error]) { NSLog(@"Major Error"); }

  [self.logs insertObject:log atIndex:0];

  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];
  [self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                        withRowAnimation:UITableViewRowAnimationFade];
  [self.tableView scrollToRowAtIndexPath:[NSIndexPath indexPathForRow:0 inSection:0]
                        atScrollPosition:UITableViewScrollPositionTop animated:YES];
}

/* ... */
```

Implementing the Table Data Source Methods

- We need to change -tableView:numberOfRowsInSection: to return the number of items in the array
- We need to change -tableView:cellForRowAtIndexPath: to return an appropriately created cell

RootViewController.m

```
/* ... */
- (NSInteger)tableView:(UITableView *)tableView
                                        numberOfRowsInSection:(NSInteger)section {
    return [self.logs count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
                                  cellForRowAtIndexPath:(NSIndexPath *)indexPath {

  static NSString *CellIdentifier = @"Cell";
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                  reuseIdentifier:CellIdentifier] autorelease];
  }

  // Configure the cell...
  Log *log = [self.logs objectAtIndex:indexPath.row];
  cell.textLabel.text = [NSString stringWithFormat:@"%@", log.timeCreated];
  cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

  return cell;
}
/* ... */
```

One Last Thing...

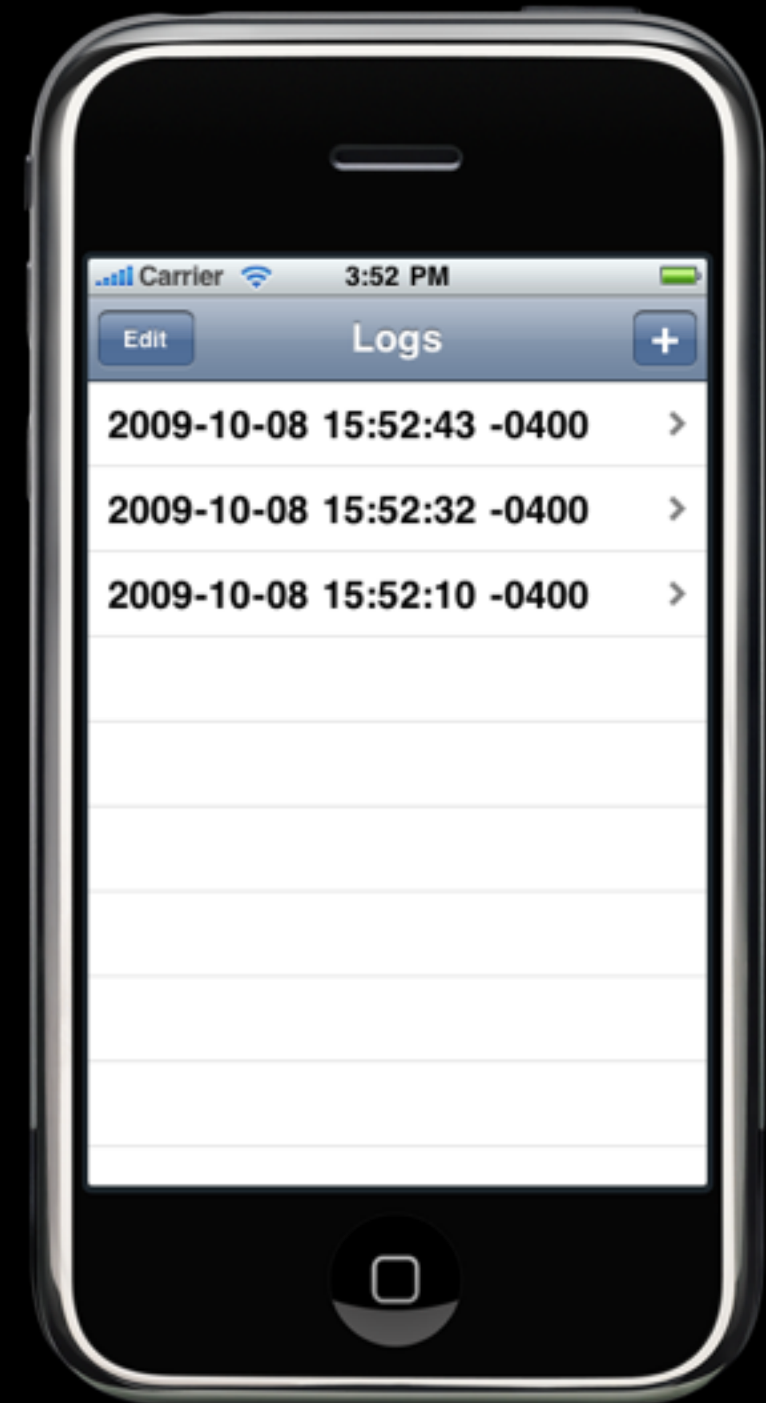
- We haven't yet initialized the logs instance variable on the Root VC
- For the time being, we'll simply add the following line to the end of -viewDidLoad in RootViewController.m

```
self.logs = [NSMutableArray array];
```

- Eventually, we'll load this from our model, but this will let us test what we have thus far

Our App Thus Far

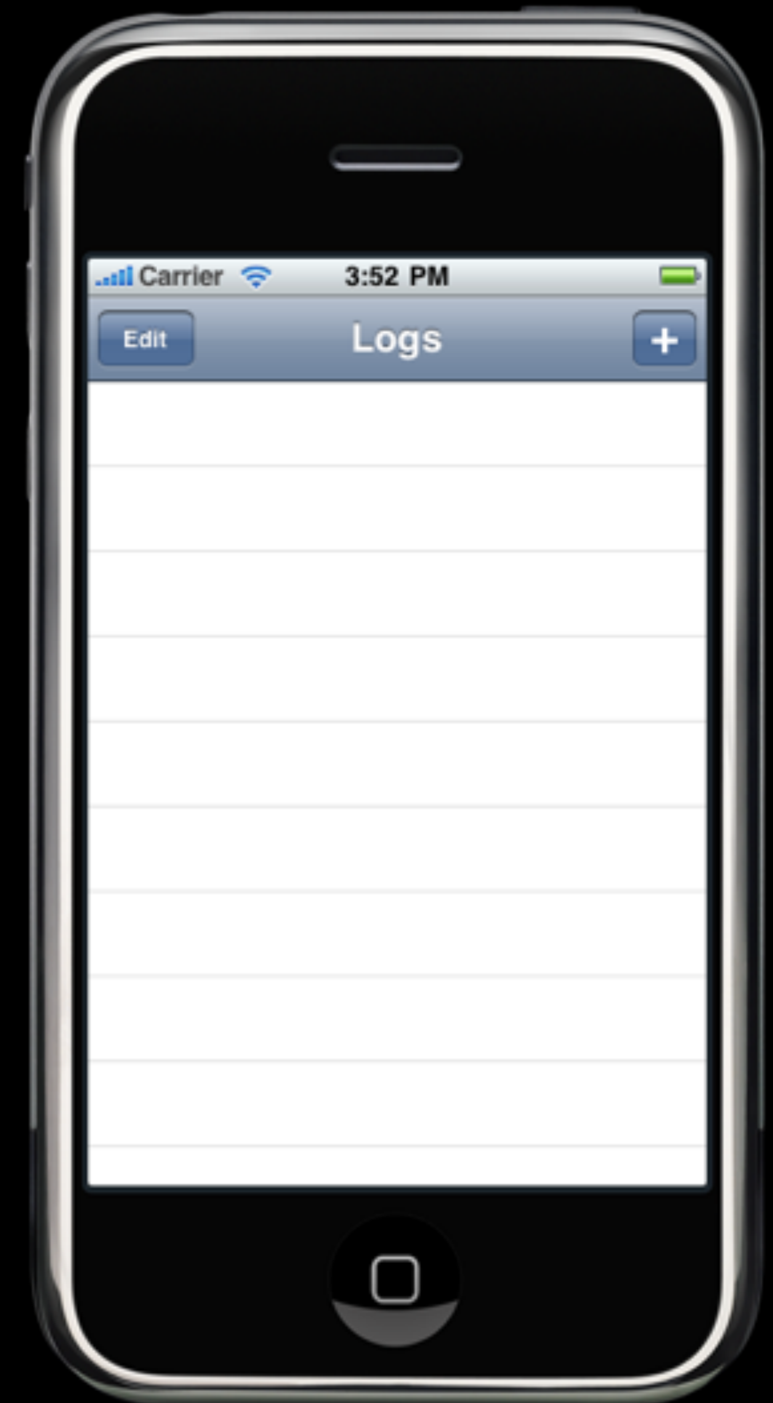
- At this point we are able to create Log instances through Core Data using our add button
- We are successfully populating the table view with the created instances



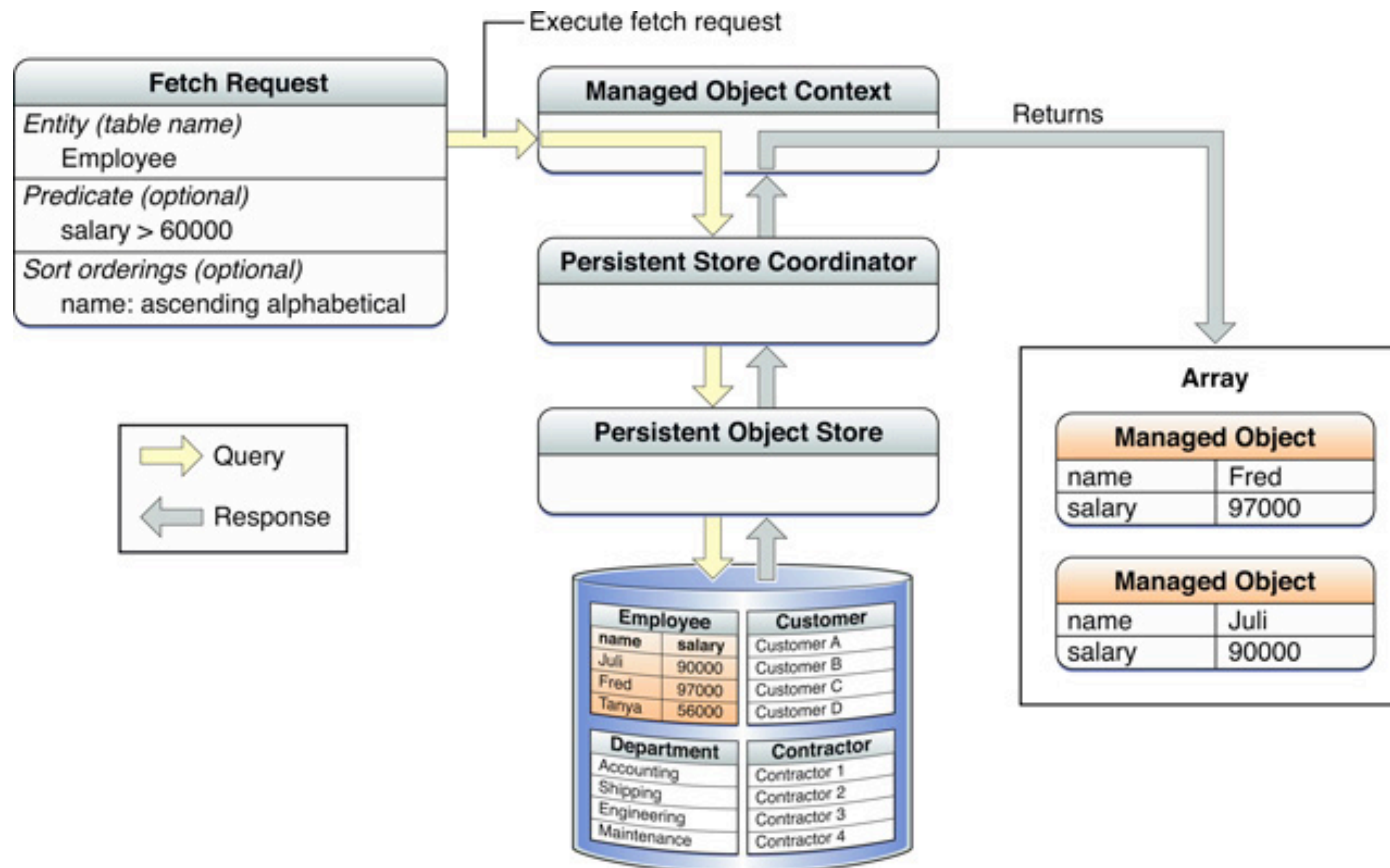
Fetching Logs

Fetching Logs

- The last thing we did to be able to make our app run was to create an empty array when the list was first displayed in -viewDidLoad
- Thus, when we first display that view, the table is always empty
- What we need to do instead, is to ask our model for all of the managed instances of Log...



Fetching Managed Objects



NSFetchRequest

- To fetch objects from a persistent store, you need a managed object context and a fetch request
- At a minimum it specifies the entity you're interested in
- It may also specify...
  - Constraints on the values that the objects should have
  - The order you want them back
- The constraints are represented by a predicate — an instance of NSPredicate
- The sort order is represented by an array of NSSortOrdering objects

Setting the Entity

- We create an NSEntityDescription using the following method...

```
+ (NSEntityDescription *)entityForName:(NSString *)entityName
                inManagedObjectContext:(NSManagedObjectContext *)context;
```

- Then we set the entity using the following NSFetchRequest method...

```
- (void)setEntity:(NSEntityDescription *)entity;
```

Setting a Sort Descriptor

- We create a sort descriptor specifying the field and ordering of the field we're interested in using the following NSSortDescriptor method..

```
- (id)initWithKey:(NSString *)key ascending:(BOOL)ascending;
```

- This descriptor then gets wrapped in an array which allows us to specify multiple fall-back descriptors in case of a tie on the primary field
- Then we set the descriptor using the following method of NSFetchRequest...

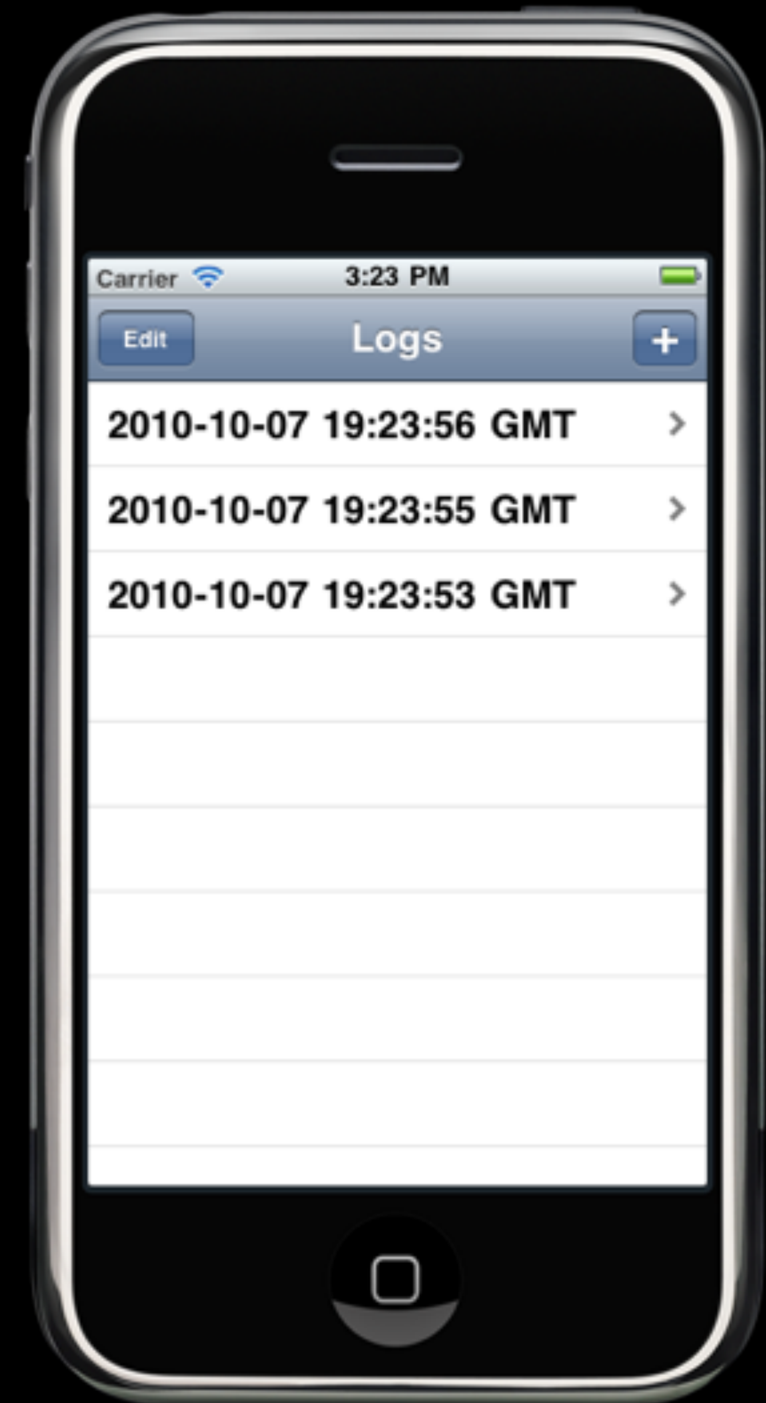
```
- (void)setSortDescriptors:(NSArray *)sortDescriptors;
```

RootViewController.m

```
- (void)viewDidLoad {
  /* ... */
  // Replace self.logs = [NSMutableArray array]; with the following...
  // The rest of -viewDidLoad should remain the same...
  NSFetchRequest *request = [[NSFetchRequest alloc] init];
  NSEntityDescription *entity = [NSEntityDescription entityForName:@"Log"
                                 inManagedObjectContext:self.managedObjectContext];
  [request setEntity:entity];

  NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
                                      initWithKey:@"timeCreated" ascending:NO];
  NSArray *sortDescriptors = [NSArray arrayWithObjects:sortDescriptor, nil];
  [request setSortDescriptors:sortDescriptors];
  [sortDescriptor release];

  NSError *error;
  NSMutableArray *mutableFetchResults = [[self.managedObjectContext
                             executeFetchRequest:request error:&error] mutableCopy];
  if (mutableFetchResults == nil) { NSLog(@"Major Error"); }

  self.logs = mutableFetchResults;
  [mutableFetchResults release];
  [request release];

  /* ... */
}
```

Our App Thus Far

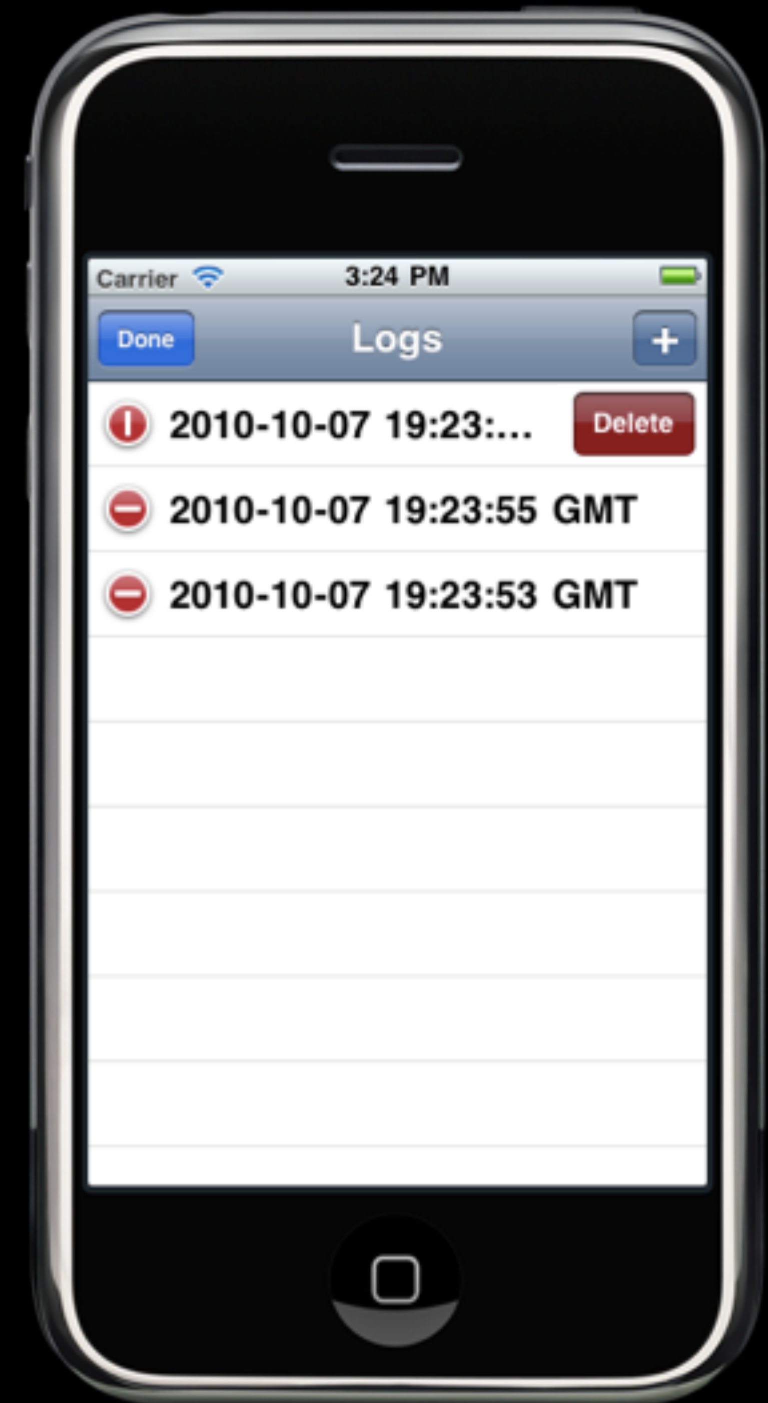
- At this point we are able now load all of the Log instances using Core Data when the app loads



Deleting Logs

Deleting Logs

- Right now our table view doesn't actually delete any Logs
- When when click Edit, then try to delete a row it refuses to go away
- Let's look at how to delete items through Core Data...



Deleting Logs

- We need to implement the following method on the RootViewController to remove logs...

```
 - (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
 forRowAtIndexPath:(NSIndexPath *)indexPath;
```

- In that method we'll need to
    - Tell the managed object context to delete the Log in question
    - Persist the change
    - Remove the Log from the array of logs
    - Update the table view

Deleting Logs

- To delete a record we tell the managed object context to delete the given object using the following method...

```
- (void)deleteObject:(NSManagedObject *)object;
```

- Then to actually persist the delete, you need to commit the action using the following NSManagedObjectContext method...

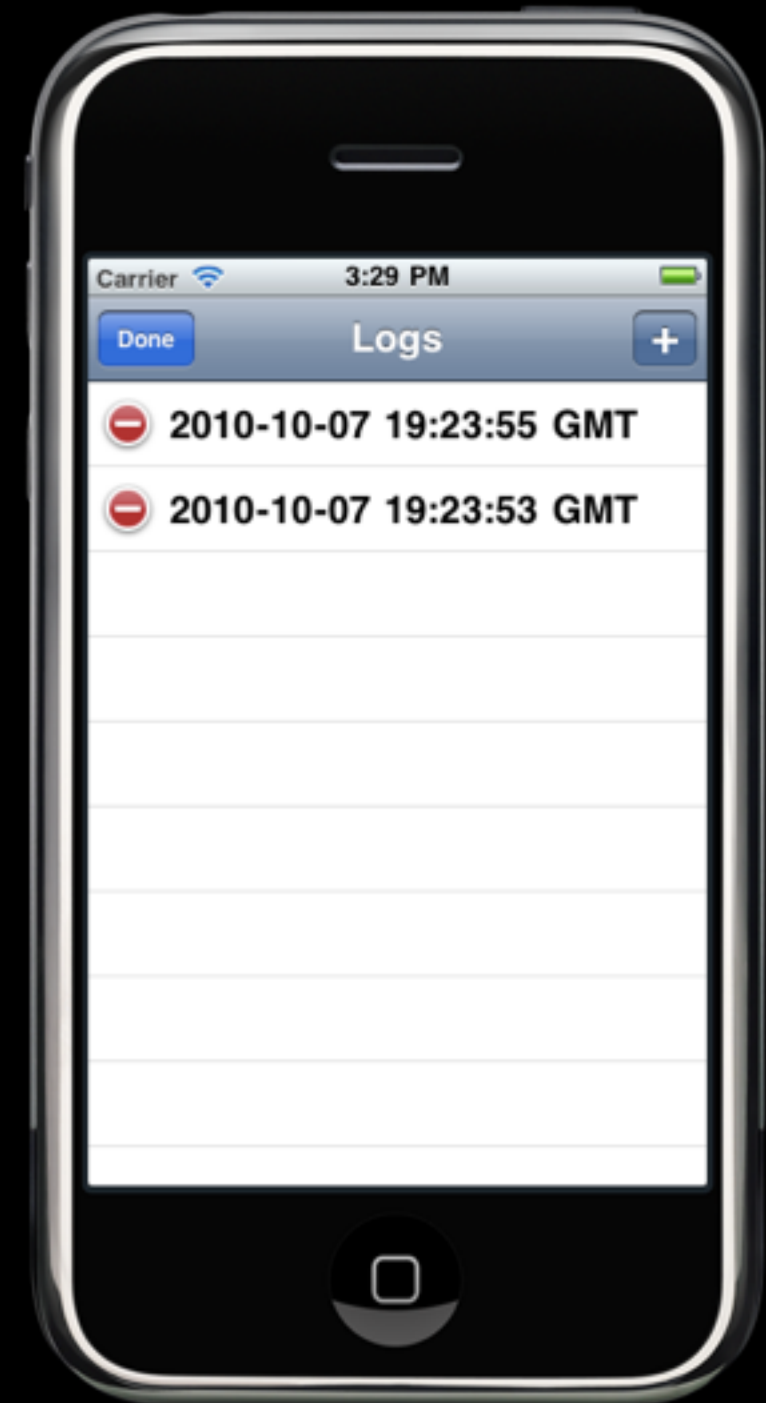
```
- (BOOL)save:(NSError **)error;
```

RootViewController.m

```
/* ... */
- (BOOL)tableView:(UITableView *)tableView
                                    canEditRowAtIndexPath:(NSIndexPath *)indexPath {
    return YES;
}

- (void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {

        NSManagedObject *logToDelete = [self.logs objectAtIndex:indexPath.row];
        [self.managedObjectContext deleteObject:logToDelete];

        NSError *error;
        if (![self.managedObjectContext save:&error]) {
            NSLog(@"Major Error");
        }

        [self.logs removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                    withRowAnimation:YES];
    }
}

/* ... */
```

Our App Thus Far

- Now we are able to successfully remove logs from our list
- We're also able to persist this change across launches of the app



Editing Logs

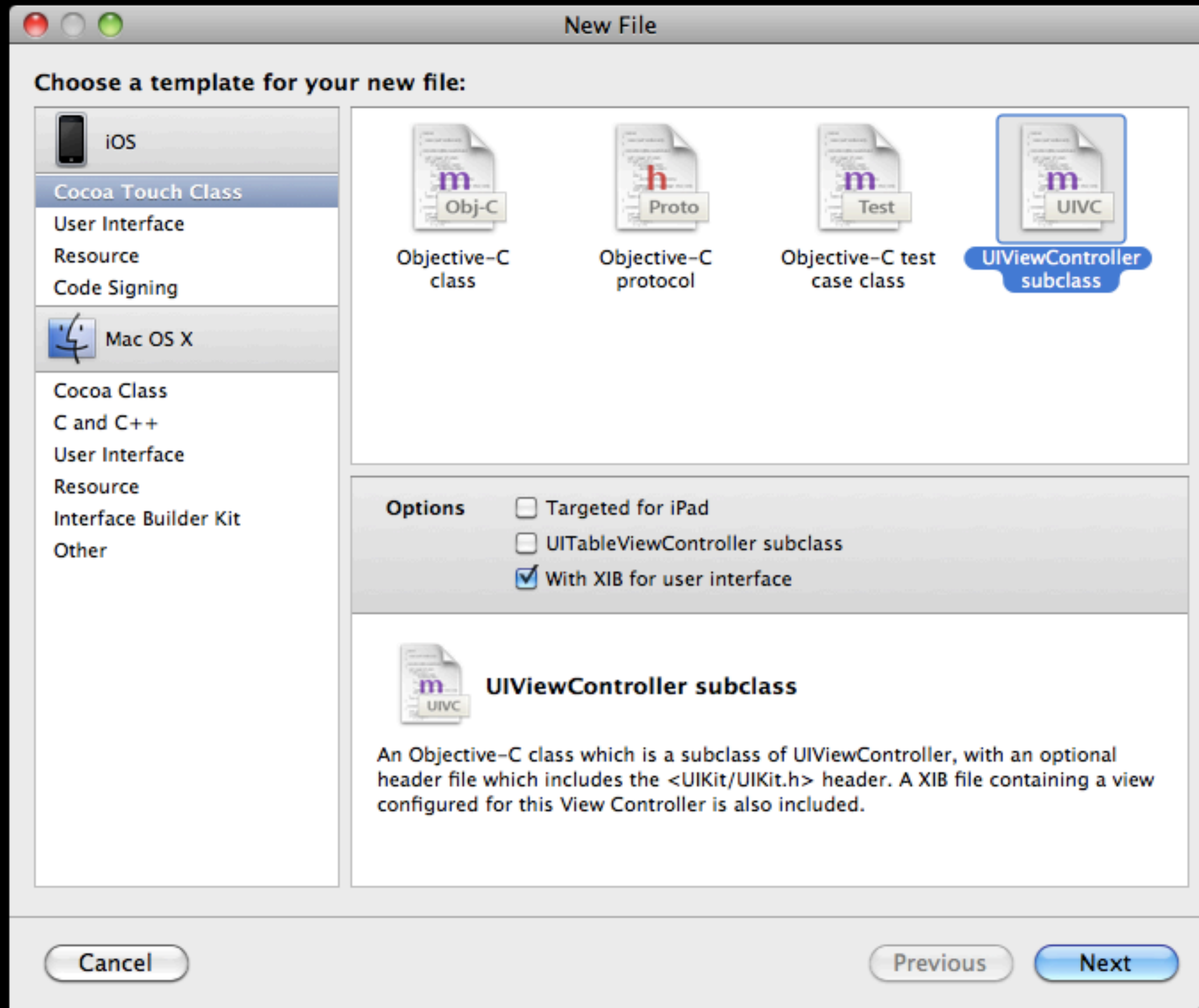
Editing Logs

- At this point our app is pretty lame — we can only record timestamps for our Logs, but can't yet add a note for that entry
- Let's fix that...

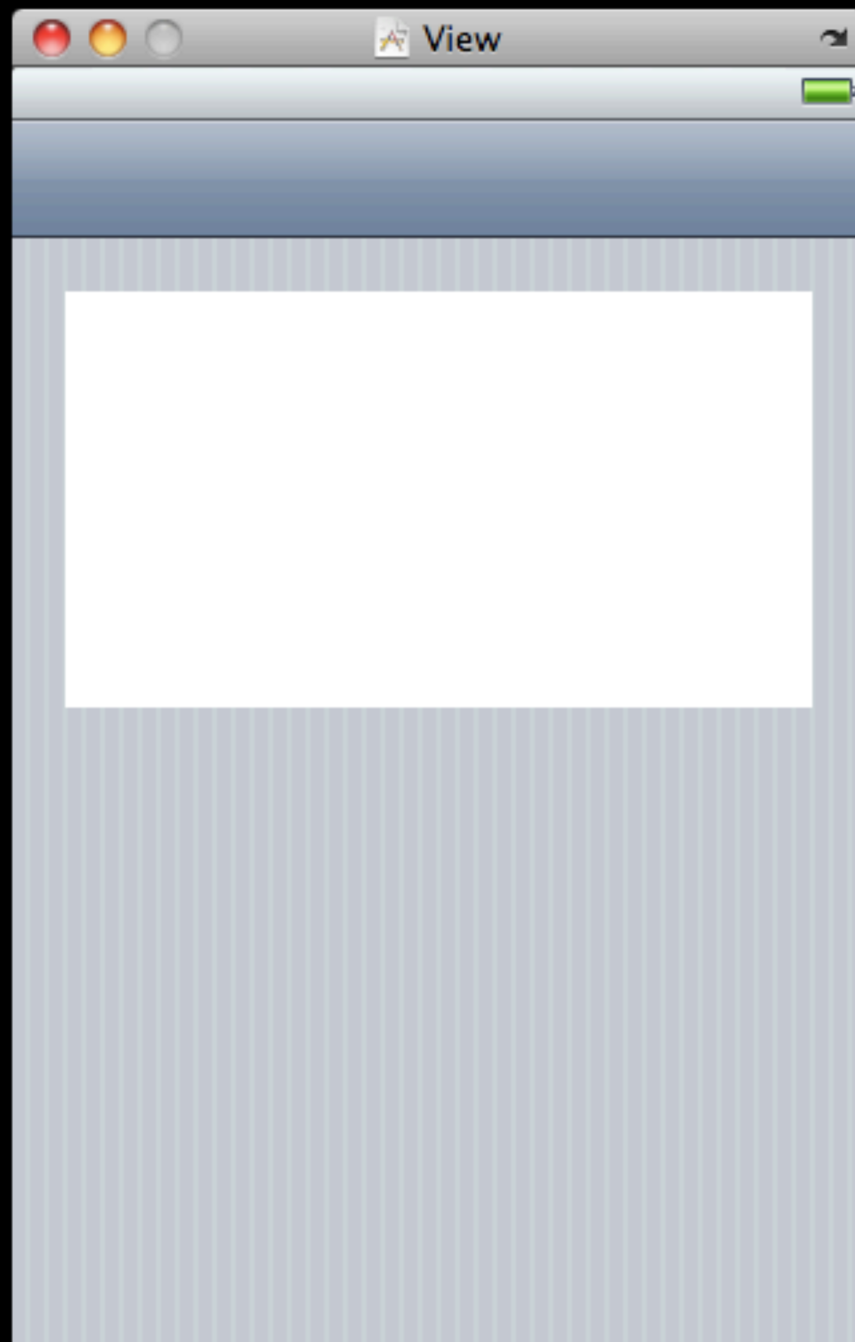
LogViewController

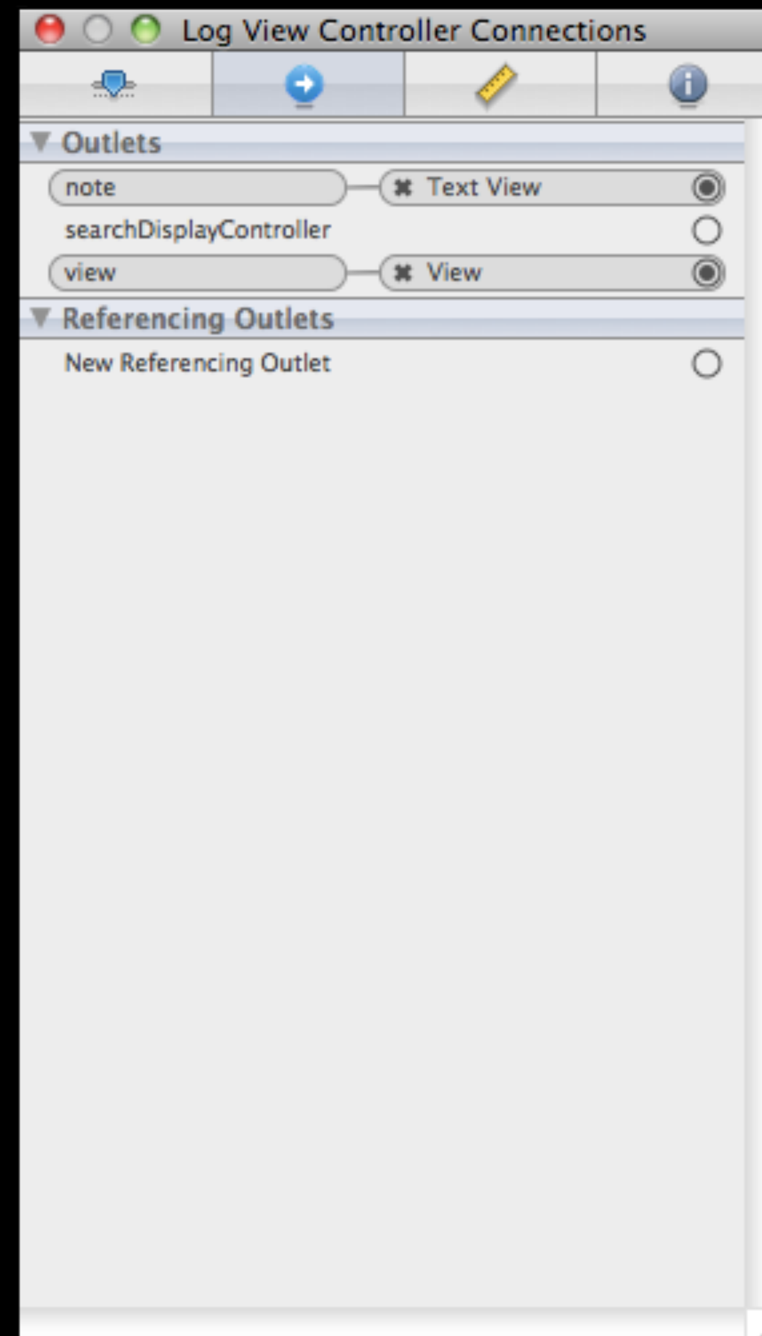
- Let’s create a LogViewController that can be used for editing a given Log
- We’ll need to...
  - Create the view controller & NIB
  - Add a Log property for the Log being viewed
  - Design the UI in IB
  - Wire up the outlet
  - Load the UI with the value of the Log
- We’ll also need to add the relevant Core Data code

Creating the LogViewController



LogViewController NIB





LogViewController.h

```
#import <UIKit/UIKit.h>

@class Log;
@interface LogViewController : UIViewController {


}

@property (nonatomic, retain) Log *log;
@property (nonatomic, retain) IBOutlet UITextView *note;

@end
```

LogViewController.m

```
#import "LogViewController.h"
#import "Log.h"

@implementation LogViewController

@synthesize log, note;

/* ... */

@end
```

Editing Logs

- To edit a record we'll simply make the changes on our managed object
- Then in order to save the changes, we'll need to get a handle back to the managed context — we can do this by asking the NSManagedObject for its context...

```
- (NSManagedObjectContext *)managedObjectContext;
```

- Then to actually persist the changes, we'll again commit the action using the following NSManagedObjectContext method...

```
- (BOOL)save:(NSError **)error;
```

LogViewController.m

```
/* ... */

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.note.text = self.log.note;
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    self.log.note = self.note.text;
    NSManagedObjectContext *context = self.log.managedObjectContext;
    NSError *error;
    if (![context save:&error]) {
       NSLog(@"Major Error");
    }
}

/* ... */
```

Changes to RootViewController

- We need to tell the Root VC about our Log VC
  - Property (with outlet)
  - Synthesize
  - Instantiate via IB
- Implement row selection code such that it...
  - Sets the selected row as the Log VC's Log
  - Push the Log VC onto the view stack

RootViewController.h

```
#import <UIKit/UIKit.h>

@class LogViewController;

@interface RootViewController : UITableViewController {

}

@property (nonatomic, retain) NSMutableArray *logs;
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;

@property (nonatomic, retain) IBOutlet LogViewController *logView;

@end
```

RootViewController.m

```
#import "RootViewController.h"
#import "LogViewController.h"
#import "Log.h"

@implementation RootViewController

@synthesize logs;
@synthesize managedObjectContext;

@synthesize logView;

/* ... */

@end
```

Modifying the RootViewController NIB

Root View Controller Connec
Outlets
logView — Log View Cont
searchDisplayController
view — Table View
Referencing Outlets
dataSource — Table View
delegate — Table View
New Referencing Outlet

Log View Controller Identi
Class Identity
Class: LogViewController
Interface Builder Identity
Name
Object ID: 8
Lock: Nothing (Inherited)
Label
Notes: Show With Selection

Log View Controller Attributes
Simulated User Interface Elements
Orientation: Portrait
Status Bar: Gray
Top Bar: Unspecified
Bottom Bar: Unspecified
View Controller
Title
Layout: Wants Full Screen
NIB Name: LogViewController
Resize View From NIB

RootViewController.xib
View Mode
Inspector
Search Field

Name	Type
File's Owner	RootViewController
First Responder	UIResponder
Table View	UITableView
Log View Controller	LogViewController

Logs.xcodeproj

RootViewController.m

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

  self.logView.log = [self.logs objectAtIndex:indexPath.row];
  [self.navigationController pushViewController:self.logView animated:YES];

}
```

Our Finished App

- We're now can perform all CRUD operations on Logs...
  - Create
  - Read
  - Update
  - Delete
- Persisted across runs of the app

Odds & Ends

A Note on Handling Errors

- In our simple app, if the data can't be saved it's likely to be indicative of some sort of catastrophic failure from which recovery might be difficult or impossible
  - Might present an alert telling the user to restart the app
- In a more complex app, the user might have changed managed objects in such they are in inconsistent state
  - Meaning they failed to meet the model's constraints
- If we had multiple managed object contexts, it's possible that the persistent store was updated and objects got out of sync between stores
- In general, you can interrogate the error object to find out what went wrong and decide what to do from there

Accessing the SQLite Database

- Sometimes it may be useful to interact with the underlying SQLite database directory when developing your app
  - Populate initial data
  - Manipulate data that's not (yet) accessible via UI
  - Debug what's going on
- The app's SQLite database resides under Documents/

```
bash-3.2$ cd ~/Library/Application\ Support/iPhone\ Simulator/4.1/Applications/
bash-3.2$ cd `\ls -1 -t | \head -n 1`/Documents
bash-3.2$ ls
Logs.sqlite
bash-3.2$
```

Connecting to the SQLite Database

- To connect to a SQLite database you simply issue the sqlite3 command, followed by the name of the database file
- I tend to set the .headers and .mode options as shown below — this tends to make the output more readable

```
bash-3.2$ sqlite3 Logs.sqlite
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .headers ON
sqlite> .mode column
sqlite>
```

Querying the SQLite Database

- You can use the .tables command to get a listing of all tables in the database
- Below I'm querying the ZLOG table which contains our managed Log objects

```
sqlite> .tables
ZLOG          Z_METADATA    Z_PRIMARYKEY
sqlite> select * from ZLOG;
Z_PK        Z_ENT       Z_OPT       ZTIMECREATED      ZNOTE
----------  ----------  ----------  ----------------  -----------
1           1           5           308172233.159379  Got Done Go
2           1           4           308172235.064852  Blah blah b
4           1           2           308173442.860356  Finish TPS
sqlite>
```

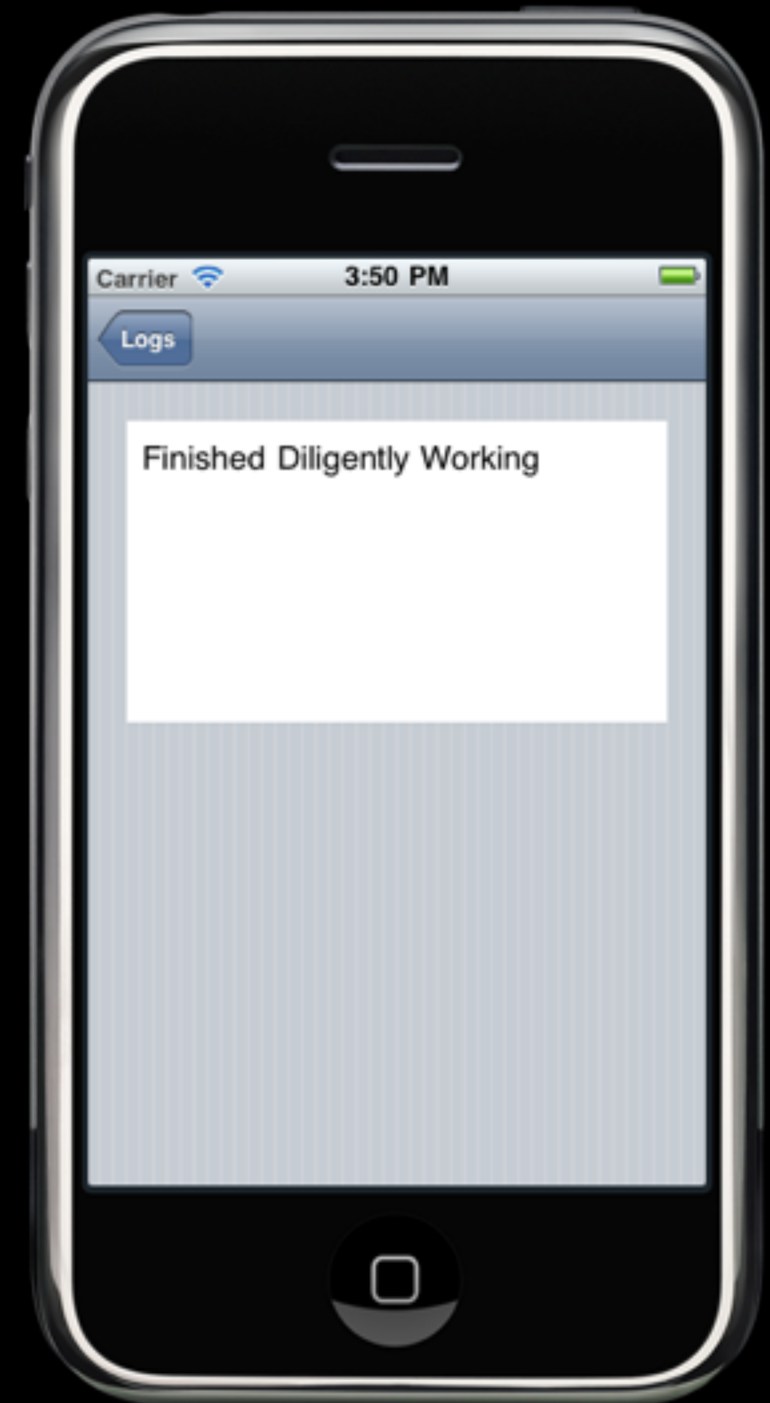
Modifying the SQLite Database

- You can also update data here using normal SQL statements
- It's probably best to exit out of your app while making changes unless you understand how the persistence stores will be affected

```
sqlite> update ZLOG set ZNOTE = 'Finished Diligently Working' where Z_PK = 2;
sqlite> select * from ZLOG;
Z_PK        Z_ENT       Z_OPT       ZTIMECREATED      ZNOTE
----------  ----------  ----------  ----------------  -----------
1           1           5           308172233.159379  Foo bar Baz
2           1           4           308172235.064852  Finished Di
4           1           2           308173442.860356  Finish TPS
sqlite>
```

Changes to Our App’s Data

- After re-launching the app, we see the changes we made manually in SQLite reflected in the UI



Complex Object Graphs

- We only looked at persisting a single object into a one table
- Core Data is very powerful and can manage persisting an entire object graph where there are several types of objects mapping to many tables
- Core Data can also manage the relationships between these objects for you and work the connections in both directions
- For example, if we had student and course objects we can model the relationship (many-to-many in this case) and ask questions such as...
    - What all courses are this student in?
    - What students are in this course?

Additional Resources

- Introduction to Core Data Programming Guide
  - http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreData/
- Getting Started with Core Data
  - http://developer.apple.com/library/ios/#referencelibrary/GettingStarted/GettingStartedWithCoreData/
- Core Data Tutorial for iOS
  - http://developer.apple.com/library/ios/#documentation/DataManagement/Conceptual/iPhoneCoreData01/

For Next Class

- Prep for your Final Project Initial Presentation
  - Slides due Monday, October 11th by 11:59pm