goto; conference

# MagicalRecord ❤️ Swift

*Saul Mora*
*@casademora*

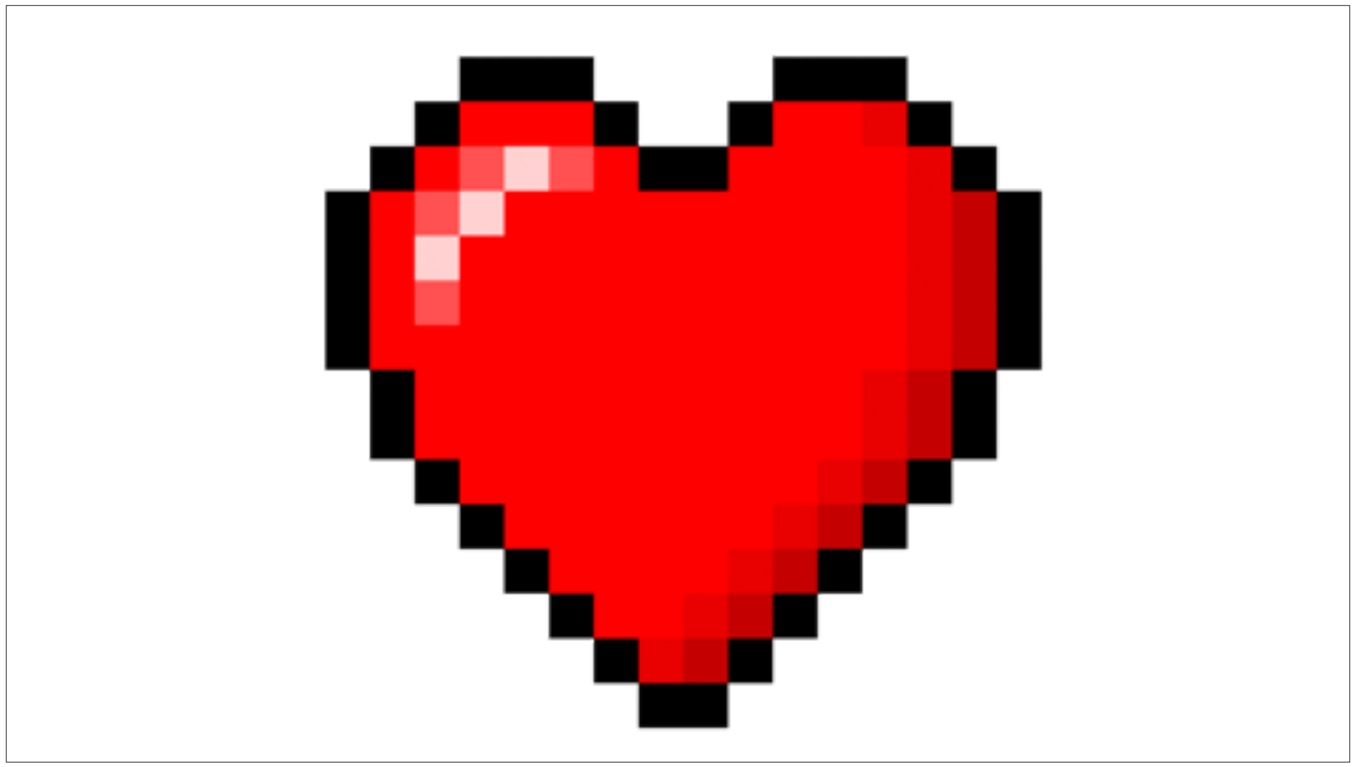
Join the conversation #gotocph

Click 'engage' to rate sessions and ask questions

NSBrief

loungebuddy

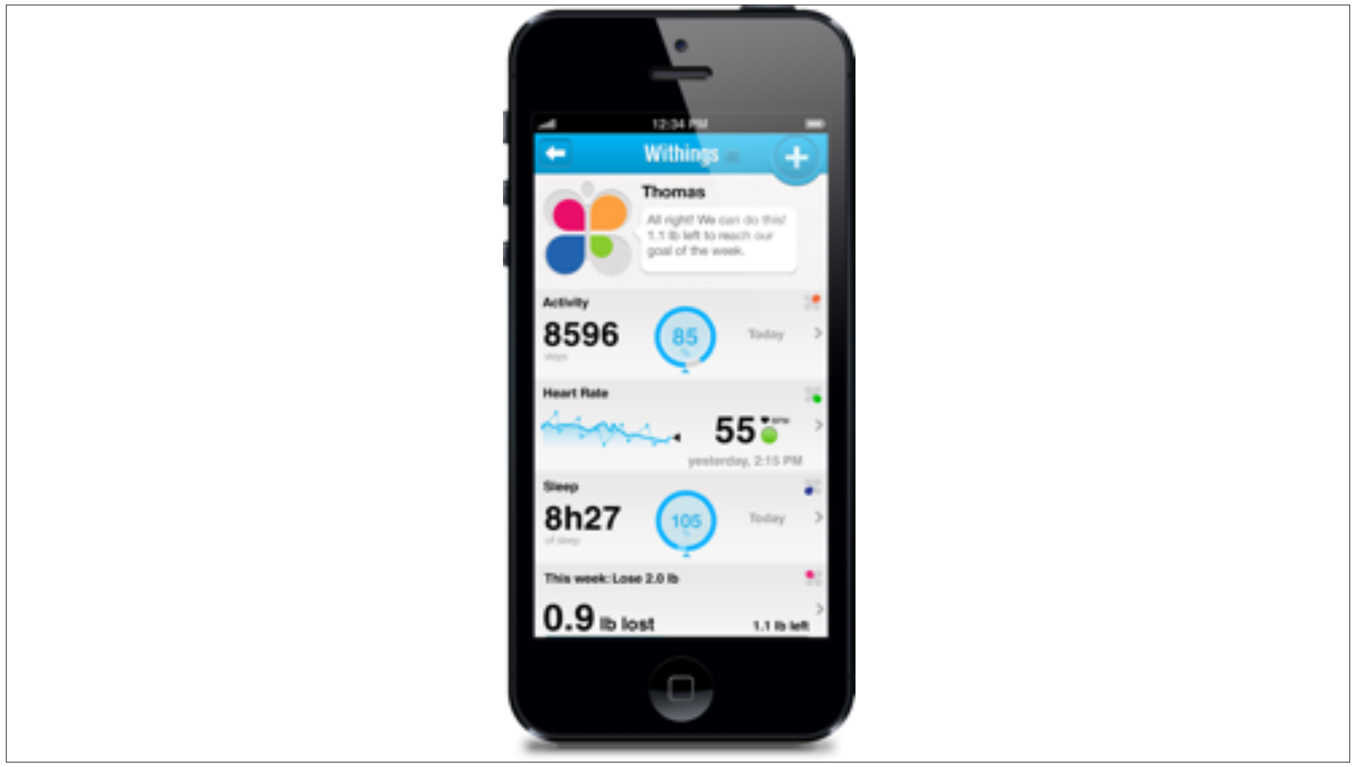
Data is the heart of your application. Or in apple terms, is the Core…



Data is the workhorse..the place where actual ideas and business logic happens in your app!

But users they want all the new shiny apps and are well designed and polished and lickable…but they don’t realize that

Data is what makes their likable app something they want to use.



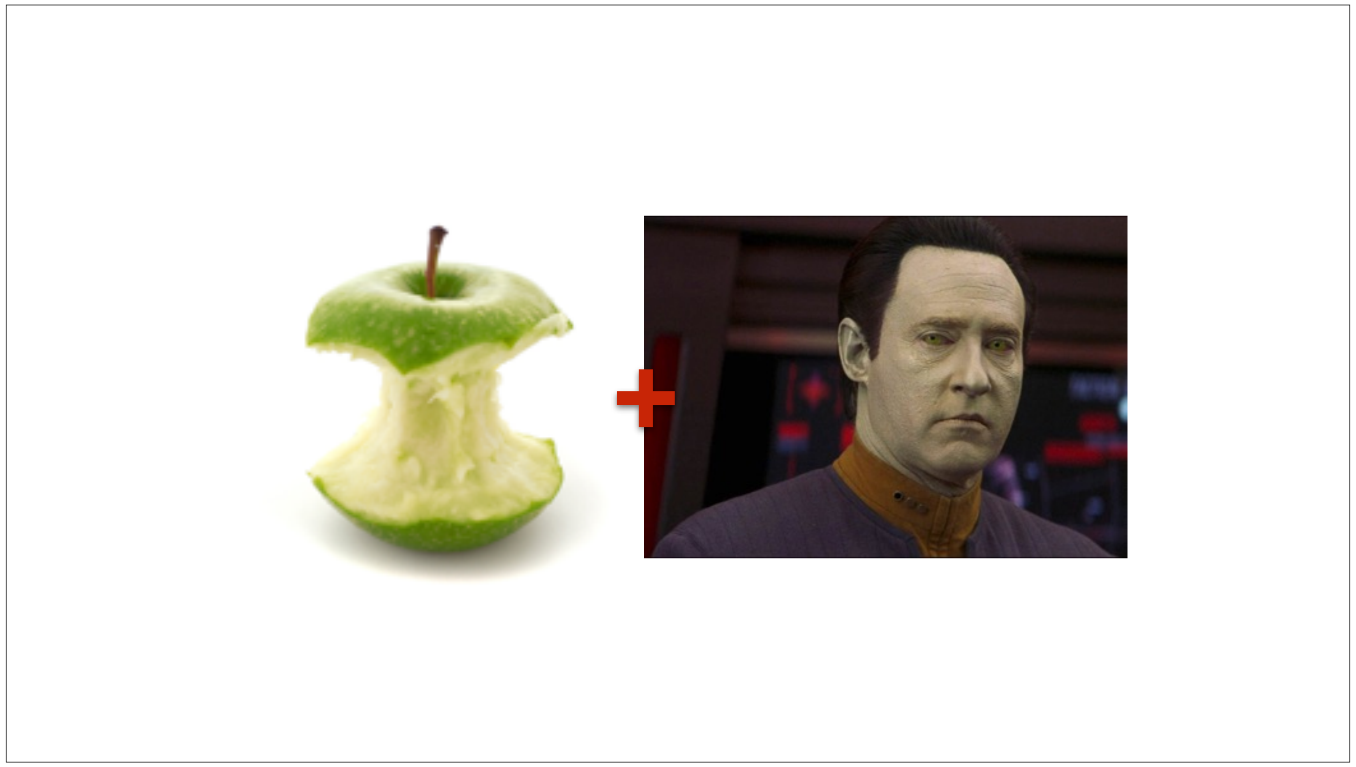
Without data, this app is useless….it’s just a bunch of shiny pixels but not useful for much else

So, as iOS developers we need a way to handle data and make it work well together with our gorgeous user interfaces…that's where a data framework comes in…

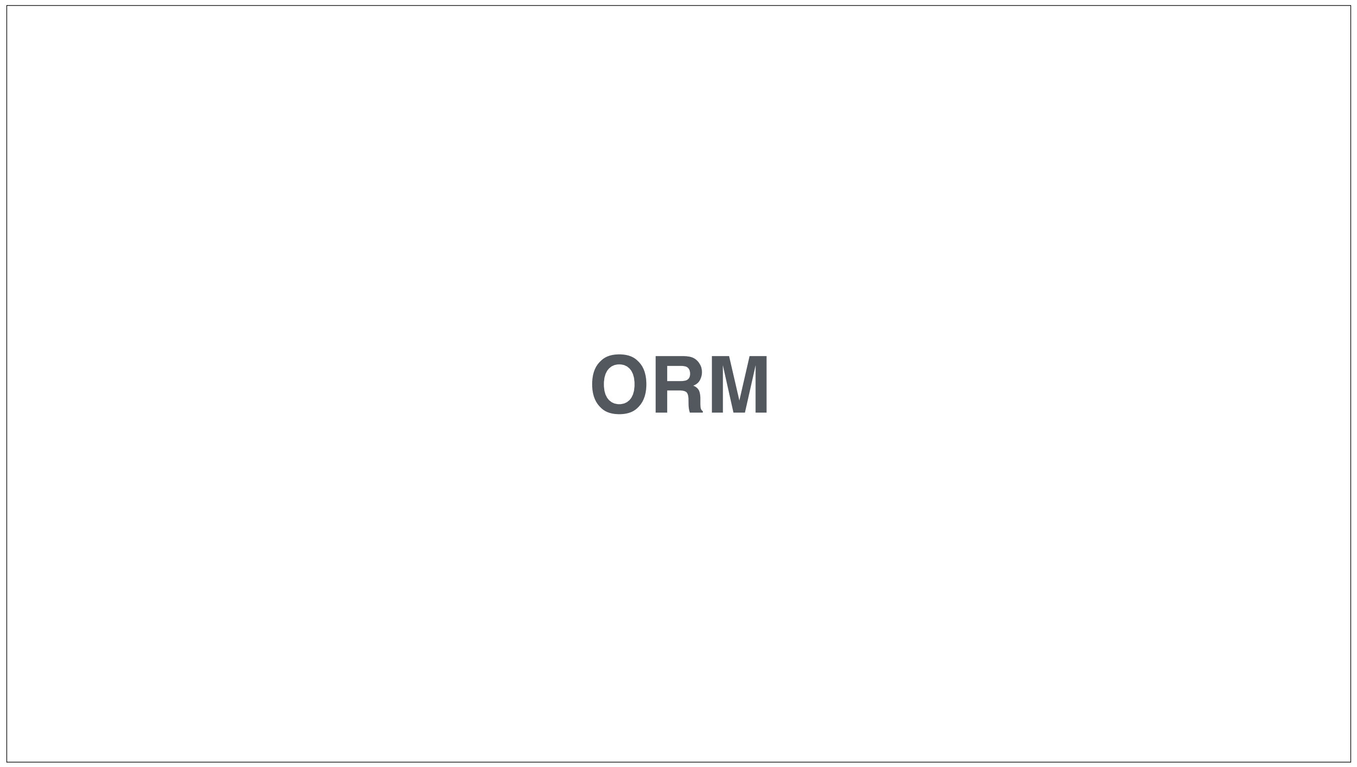
As you start working, you find you're most often working in **objects**. But you store your data in **tables and relations**. You make a lot of glue code for this…

Data Code
- Glue code
- DAOs. Yuck
- Can easily mix the "database" code with "business model" code

When it comes to writing code for that data, it can become messy quite quickly. When you mix two different languages together, boundaries begin to blur, and "data logic" melds with "business logic".



But what exactly is Core Data?
Core Data is what you would build if you wrote this framework yourself. The API certainly is cause for many questions and confusion, but I've come to realize that Core Data is a framework, or the lowest level of abstraction Apple is comfortable in providing you in order to keep the object oriented ideals in tact.



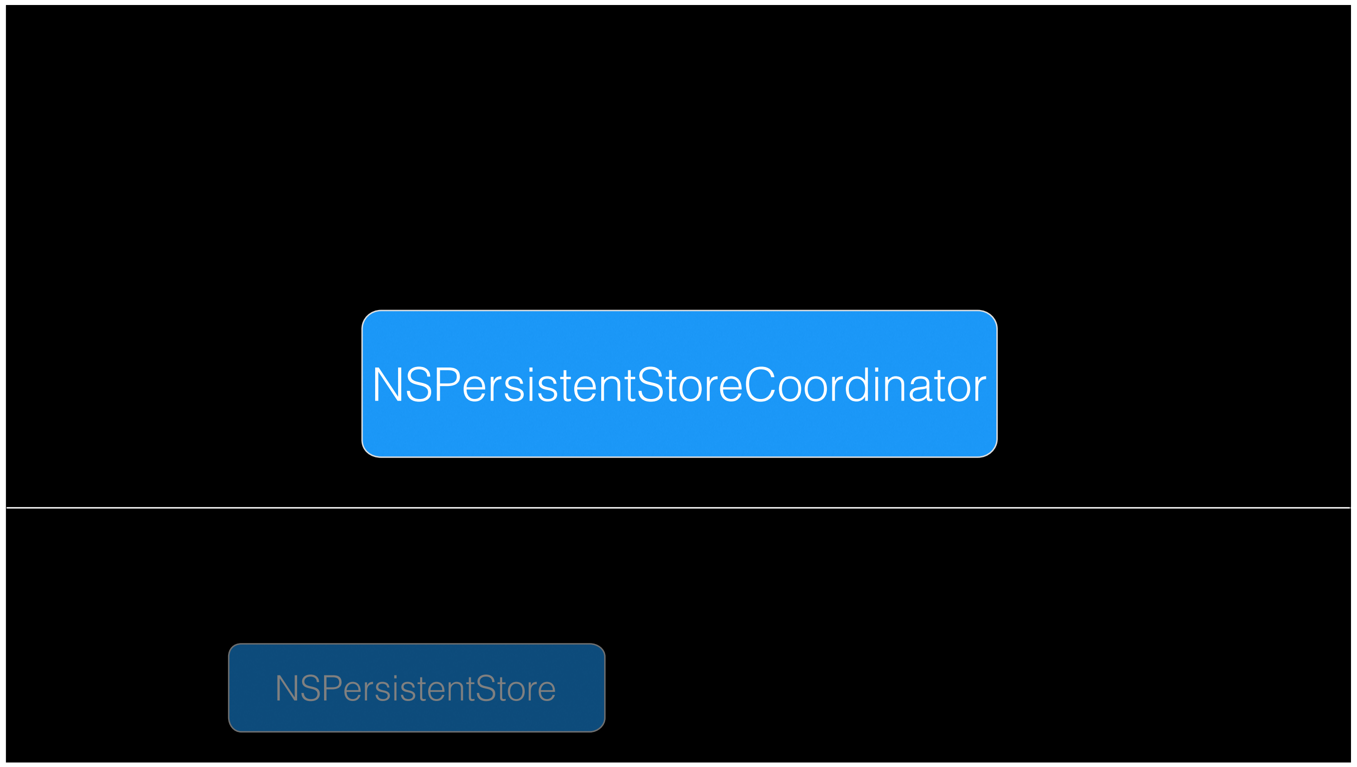
Eventually, you build an ORM. ORMs have a few shortcomings of their own:

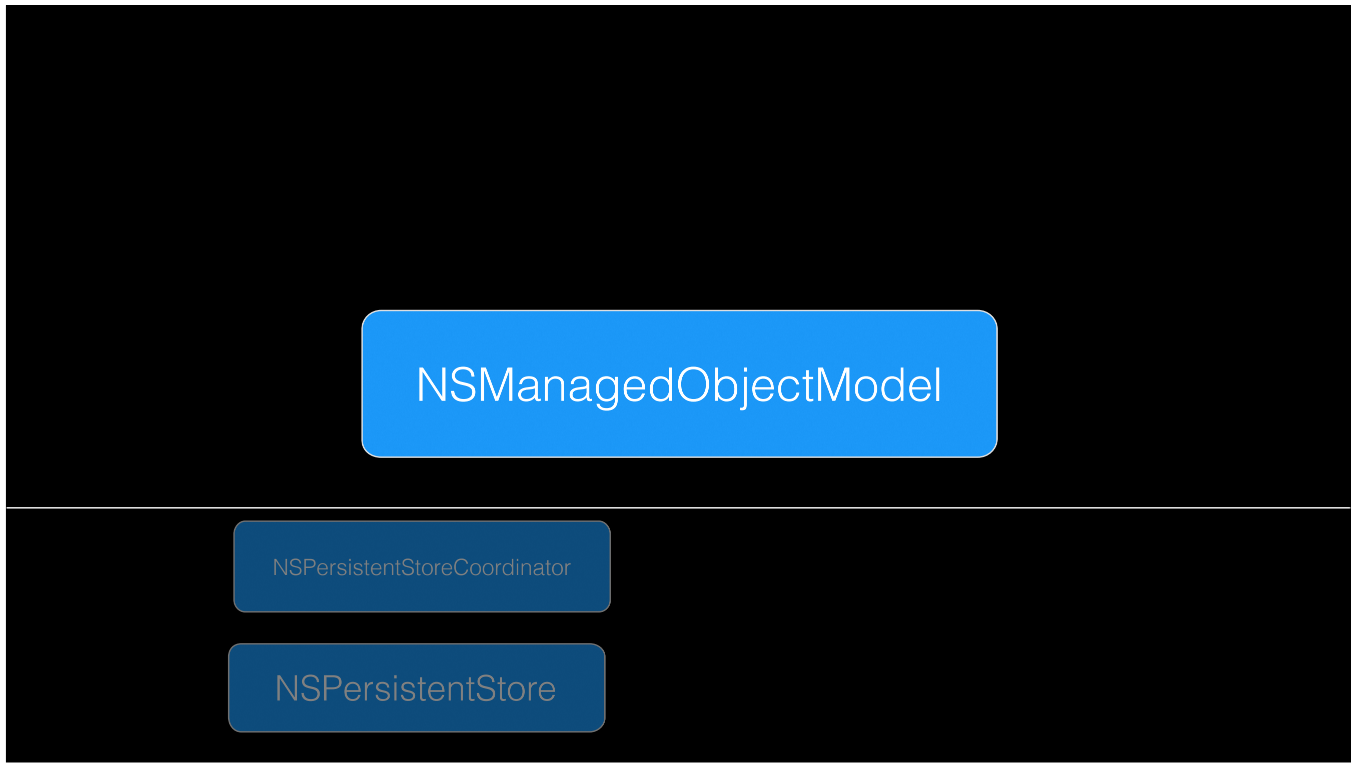
- Abstractions SQL leak into your object code

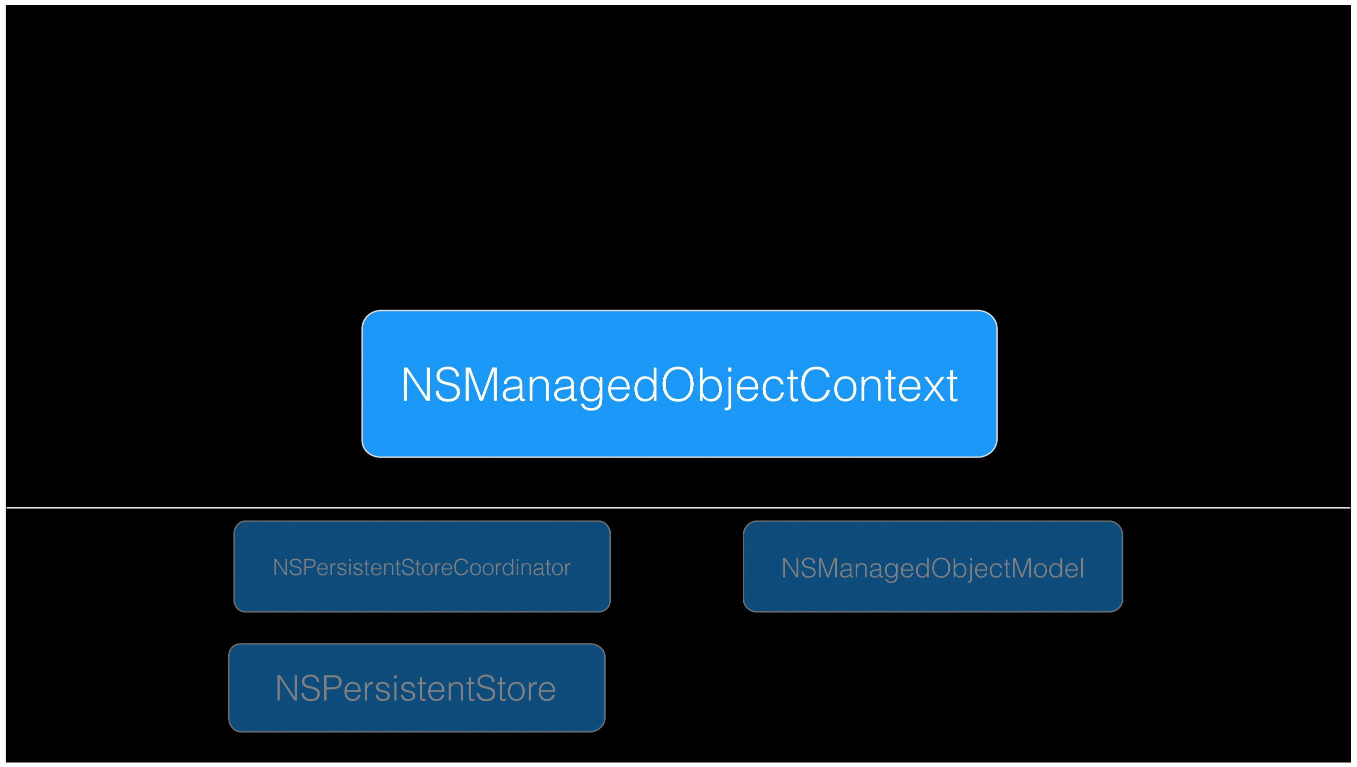
ORMs are a **leaky abstraction**.
CoreData is less leaky, but still there are a few places where those boundaries appear. The bottom line is, while CoreData is not an ORM, it's an **Object Graph Persistence Framework**, it is something you would write anyway.

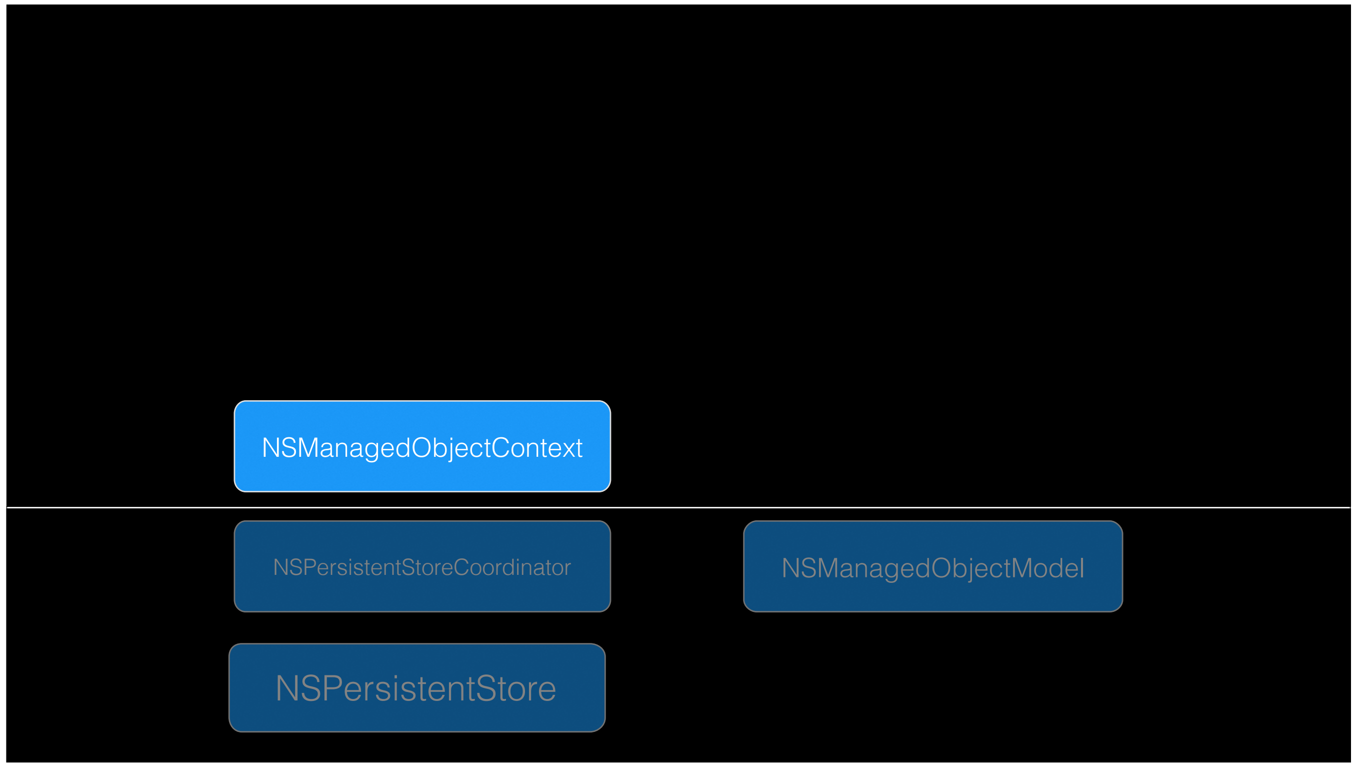
CoreData is an Object Graph Persistence Framework…

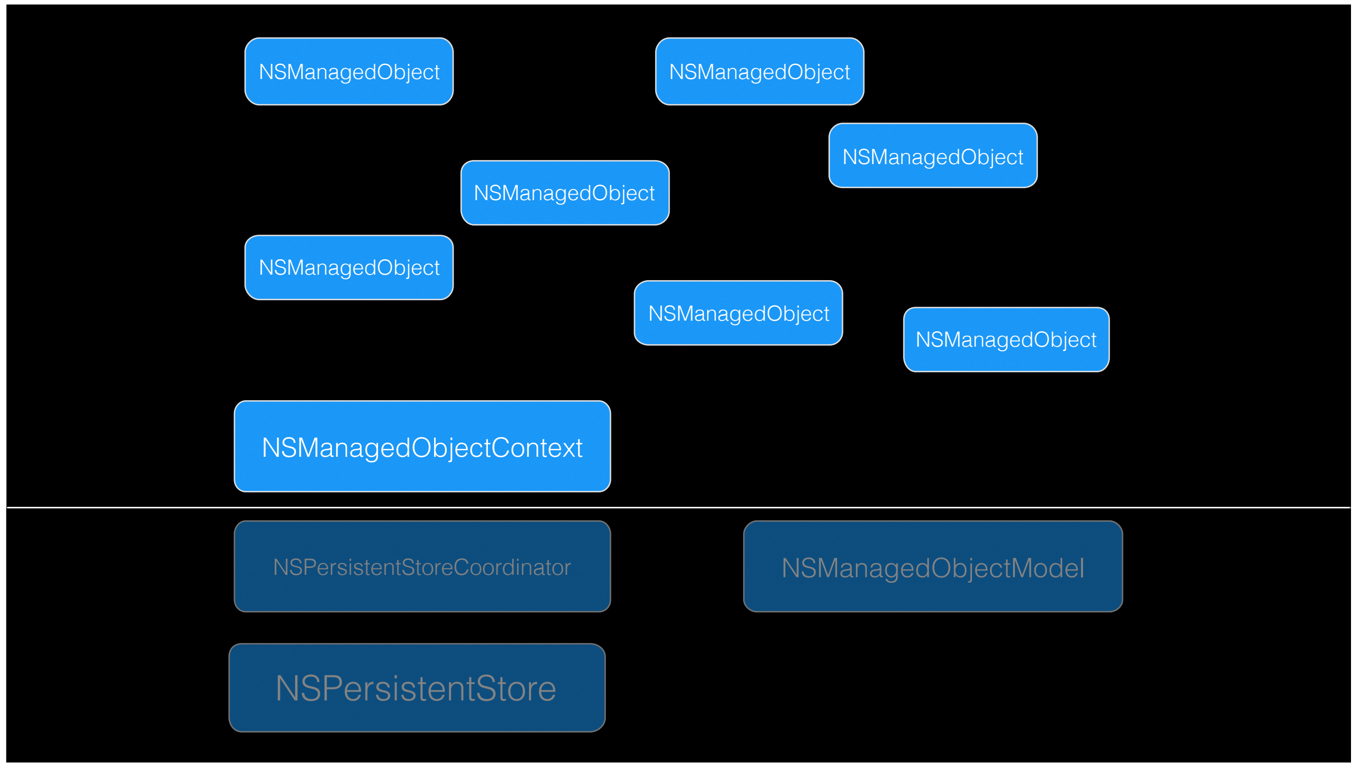
NSPersistentStore

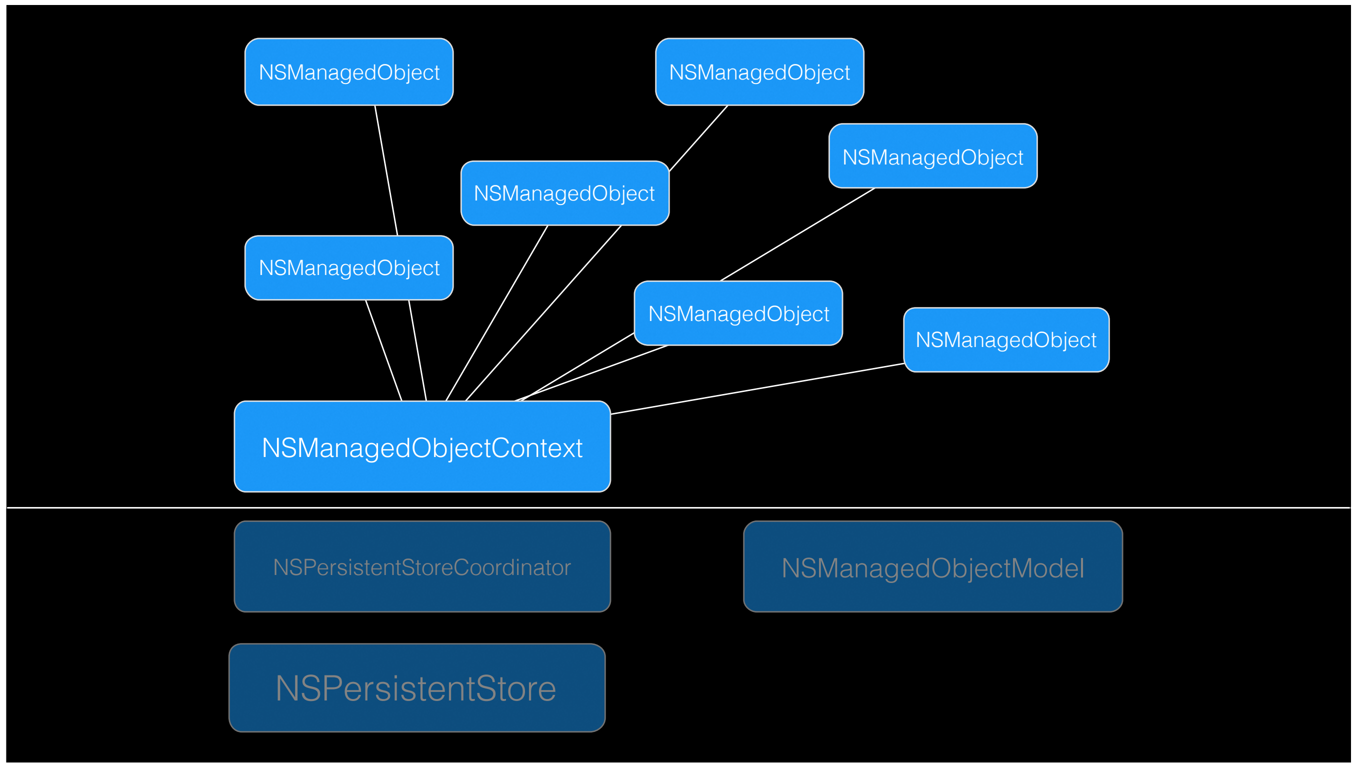

NSPersistentStoreCoordinator
NSPersistentStore

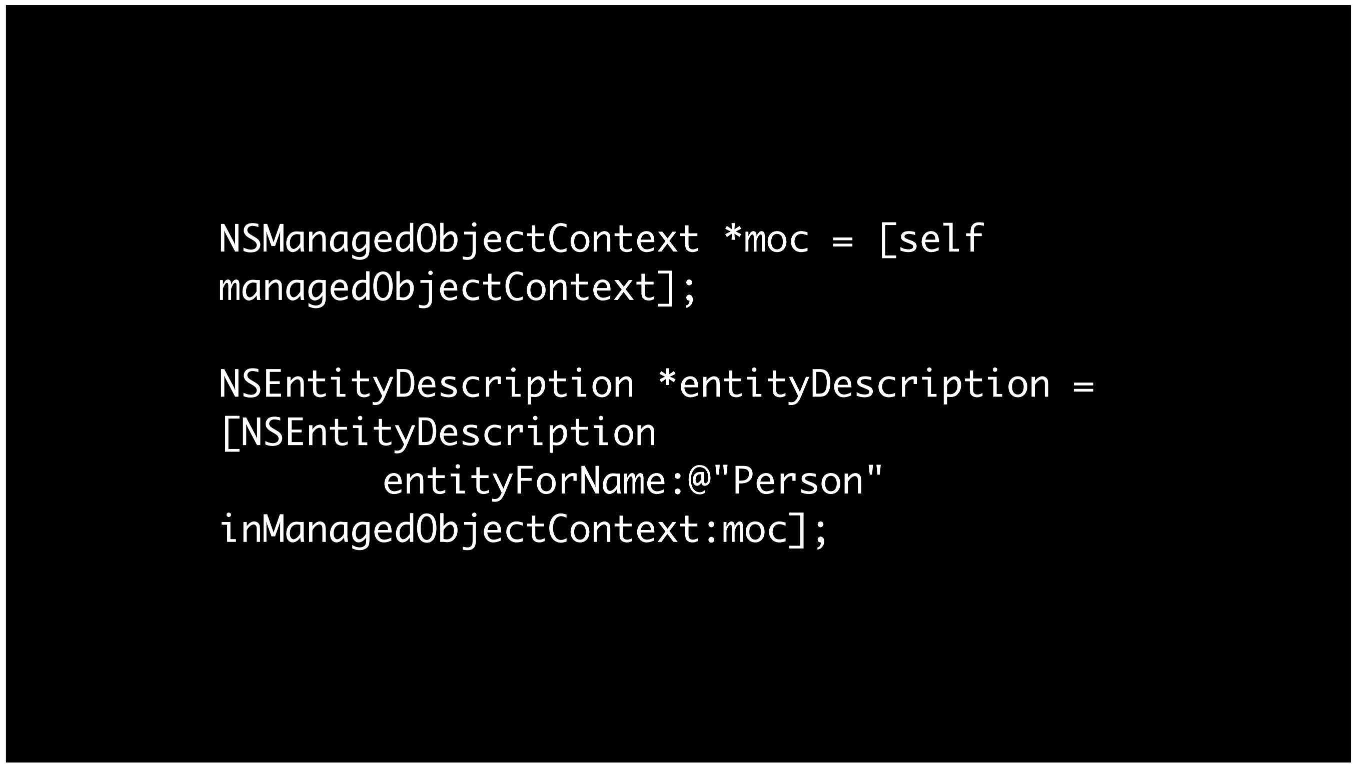

NSManagedObjectModel
NSPersistentStoreCoordinator
NSPersistentStore


NSManagedObjectContext
NSPersistentStoreCoordinator
NSManagedObjectModel
NSPersistentStore


NSManagedObjectContext
NSPersistentStoreCoordinator
NSManagedObjectModel
NSPersistentStore

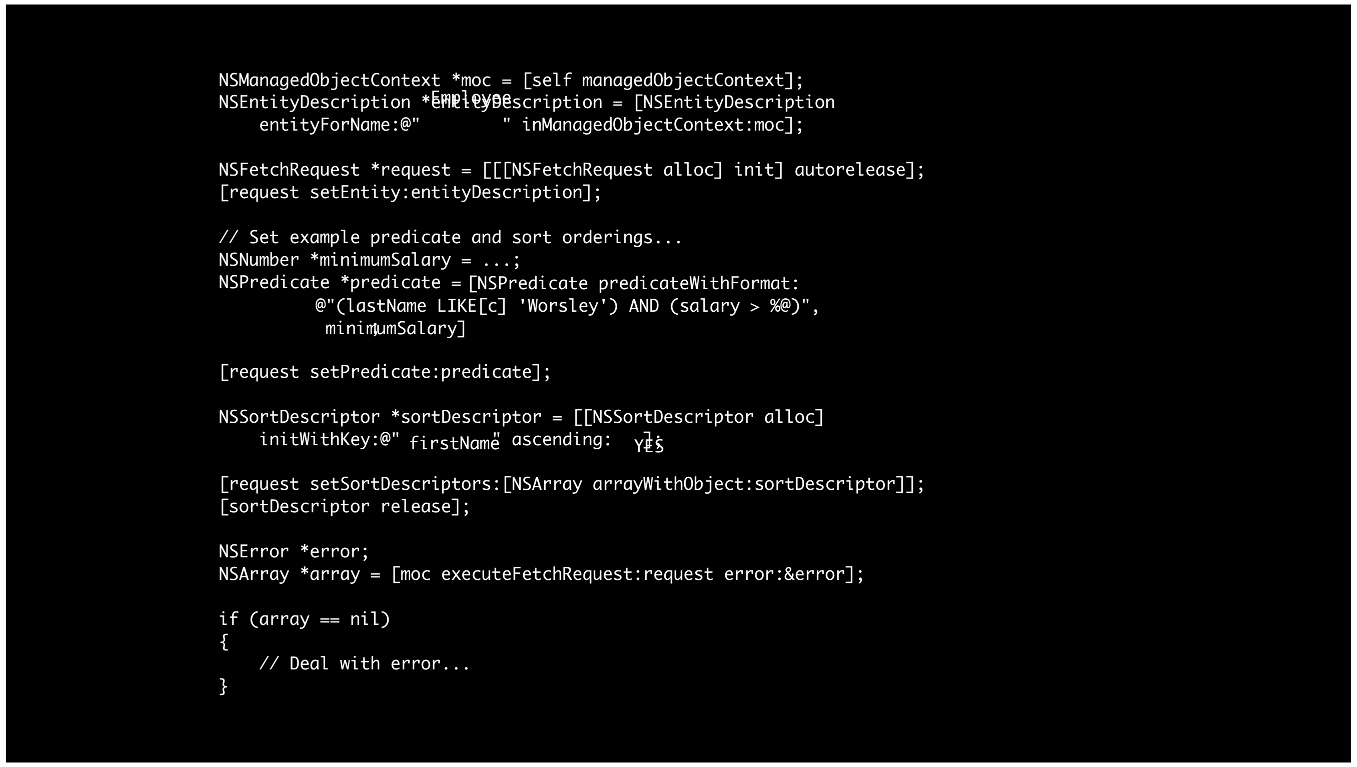

NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObjectContext
NSPersistentStoreCoordinator
NSManagedObjectModel
NSPersistentStore


NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObject
NSManagedObjectContext
NSPersistentStoreCoordinator
NSManagedObjectModel
NSPersistentStore



```
NSManagedObjectContext *moc = [self
managedObjectContext];

NSEntityDescription *entityDescription =
[NSEntityDescription
       entityForName:@"Person"
inManagedObjectContext:moc];
```

Creating new NSManagedObjects from another container object



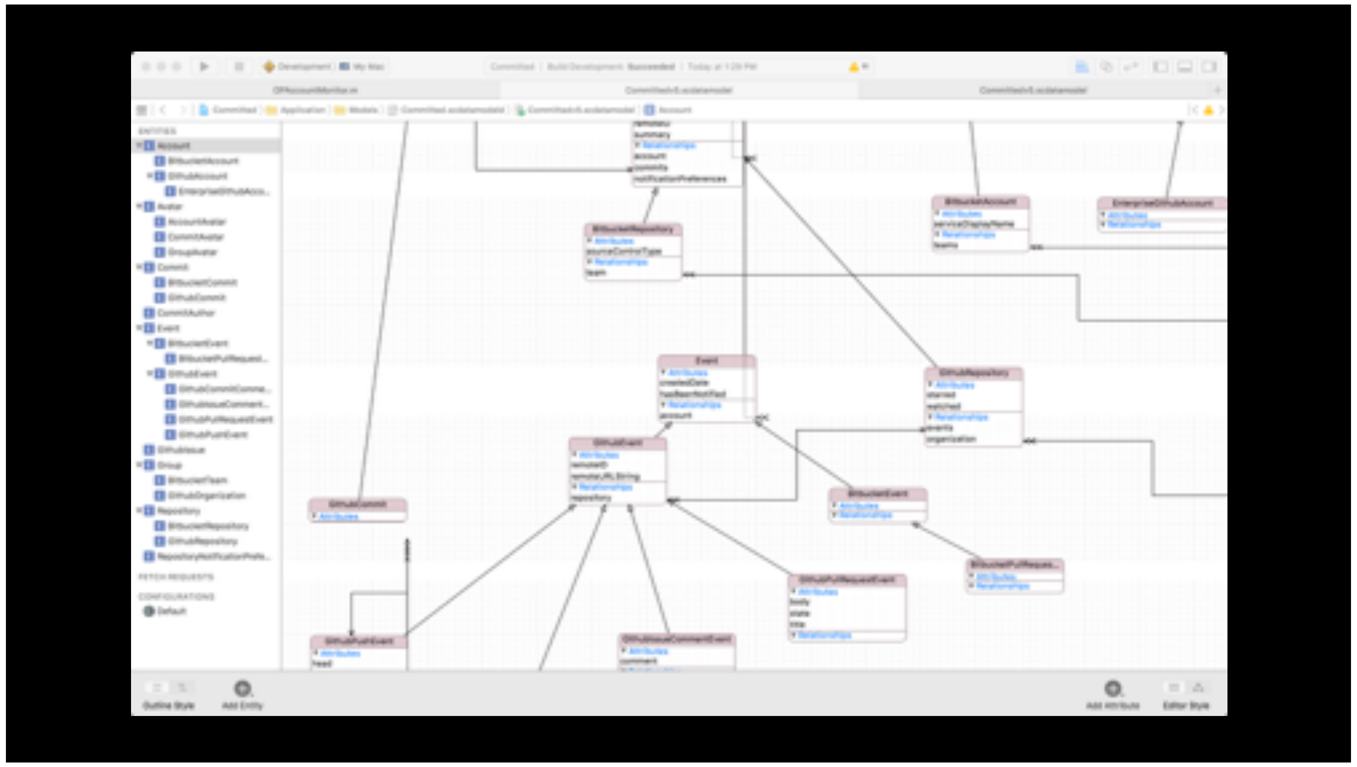
```
NSManagedObjectContext *moc = [self managedObjectContext];
NSEntityDescription *entityDescription = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:moc];

NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
[request setEntity:entityDescription];

// Set example predicate and sort orderings...
NSNumber *minimumSalary = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
          @"(lastName LIKE[c] 'Worsley') AND (salary > %@)",
           minimumSalary];

[request setPredicate:predicate];

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
    initWithKey:@"firstName" ascending:YES];

[request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];
[sortDescriptor release];

NSError *error;
NSArray *array = [moc executeFetchRequest:request error:&error];

if (array == nil)
{
    // Deal with error...
}
```

Fetch requests usually take a lot of code
Favorite part of the Apple Sample fetch request:
- Deal with error

```
NSManagedObjectContext *moc = [self managedObjectContext];
[moc performBlock:^{
    NSManagedObject *originalCustomer = //found this from a previous fetch

    NSManagedObject *customer = [moc objectWithID: [originalCustomer objectID]];

    [customer setValue:@”John Appleseed” forKey:@“name”];
    //set more properties

    NSError *error = nil;
    BOOL saved = [moc save:&error];
    if (!saved) {
        // deal with save issue
    }
}];
```

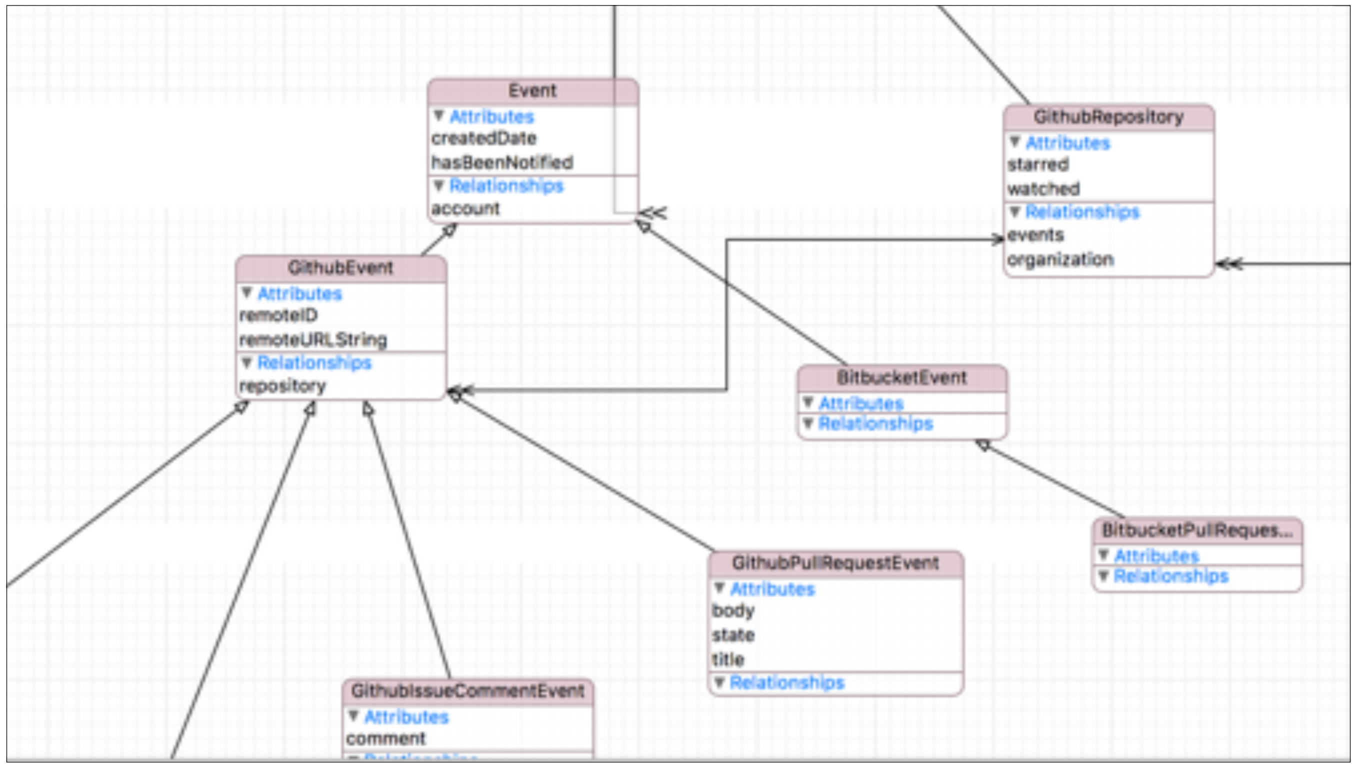
Saving changes in code

Notice that we set values for keys, so a very dynamic, Key Value coding approach to saving data. Core Data is dynamic and can adapt to whatever data comes through



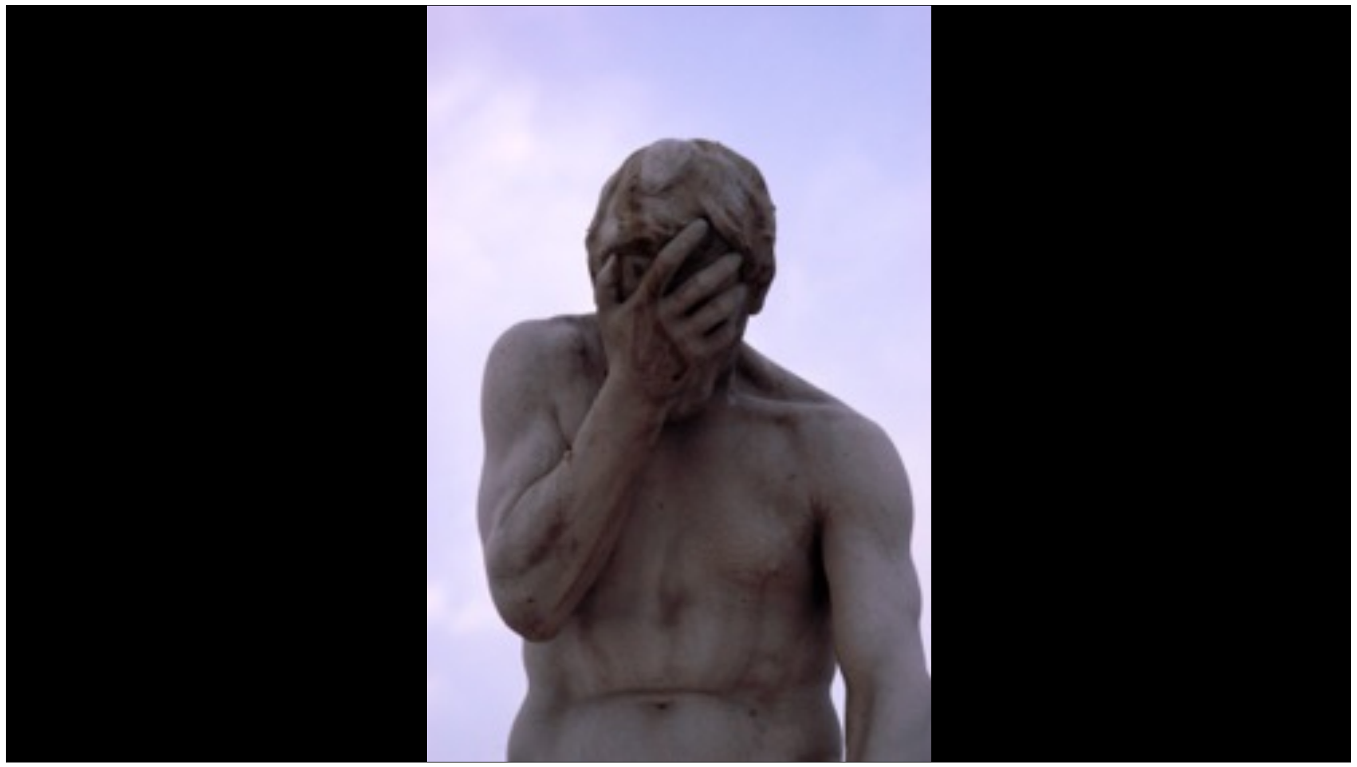
Whenever you add a new property to an entity, you’ll need to:

1) Find your entity in code
2) Manually enter your new property with the correct type information
3) make sure you don’t overwrite the old implementation of entity/business logic code by mistake.



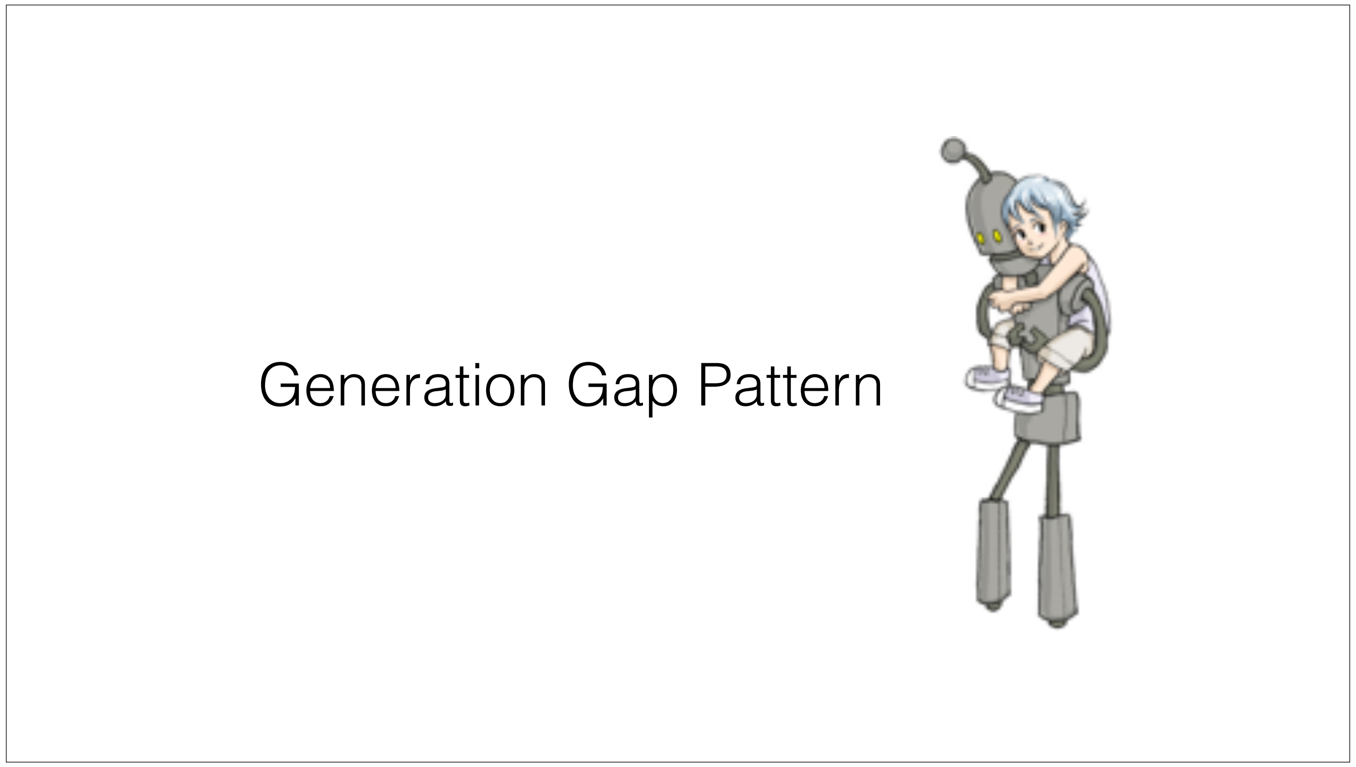
Whenever you add a new property to an entity, you'll need to:

1) Find your entity in code
2) Manually enter your new property with the correct type information
3) make sure you don't overwrite the old implementation of entity/business logic code by mistake.



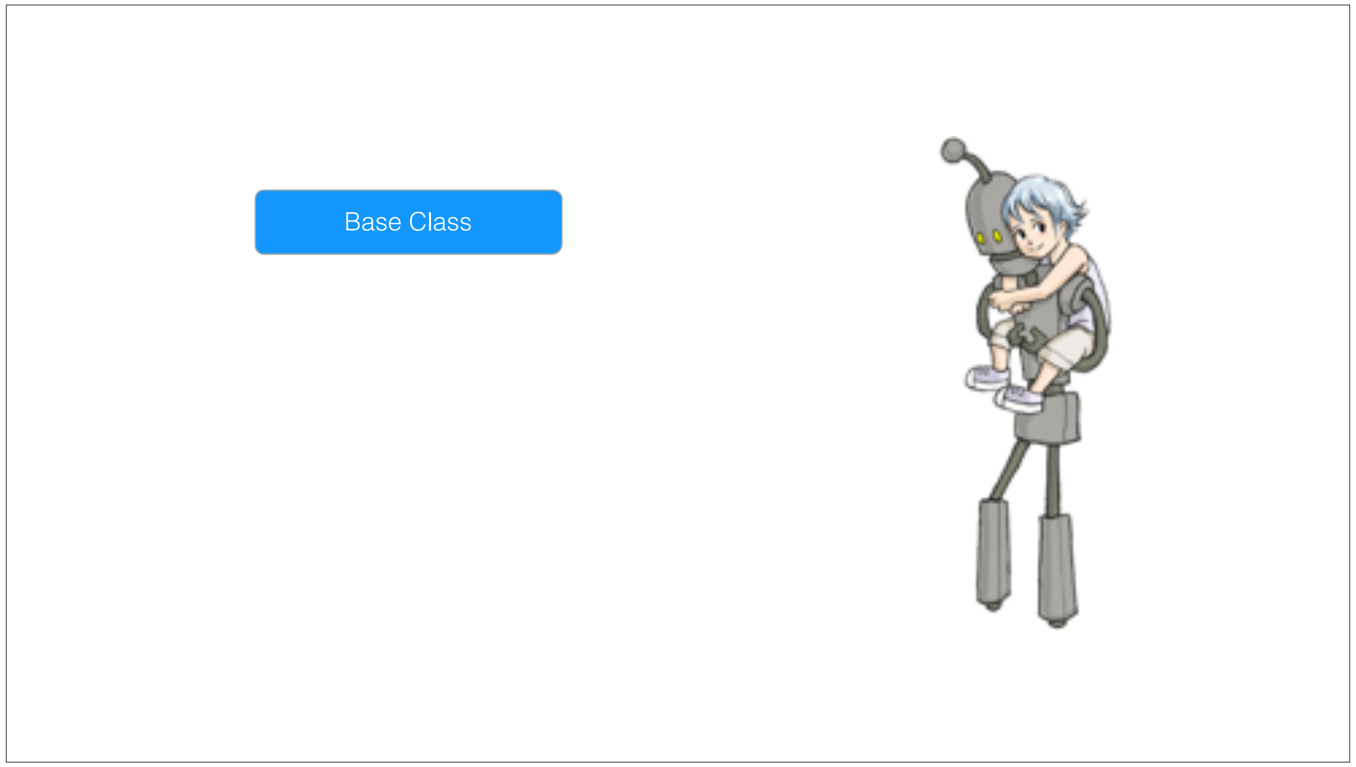
We have the technology to make this better

mogenerator implements the Generation Gap pattern

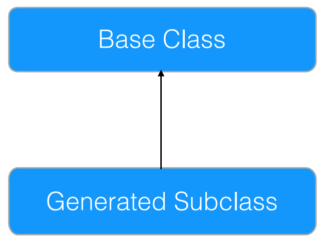


## Generation Gap Pattern

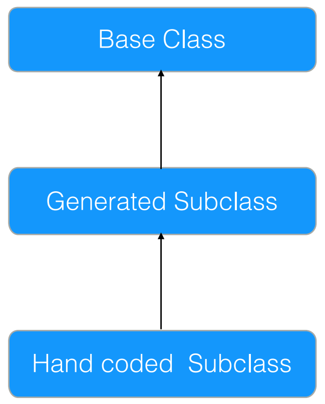
mogenerator implements the Generation Gap pattern



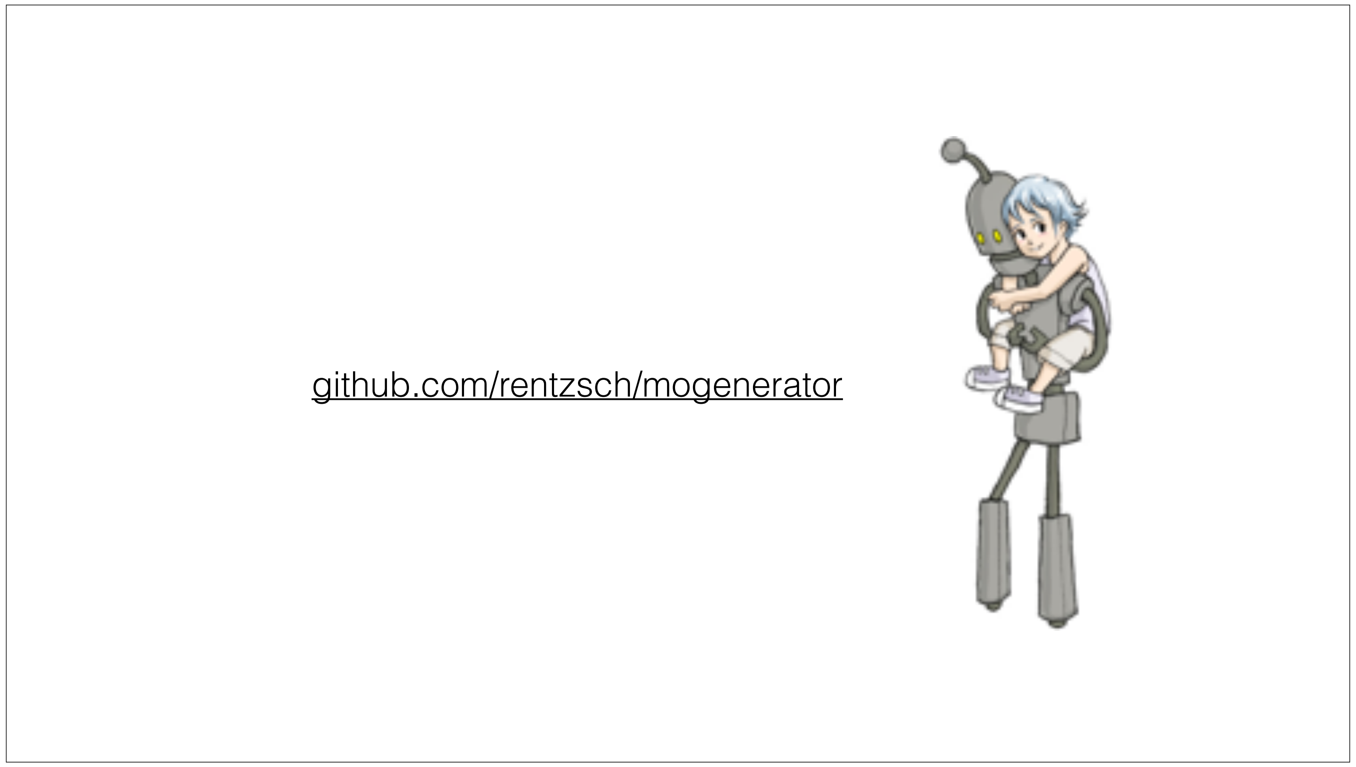
mogenerator implements the Generation Gap pattern



mogenerator implements the Generation Gap pattern

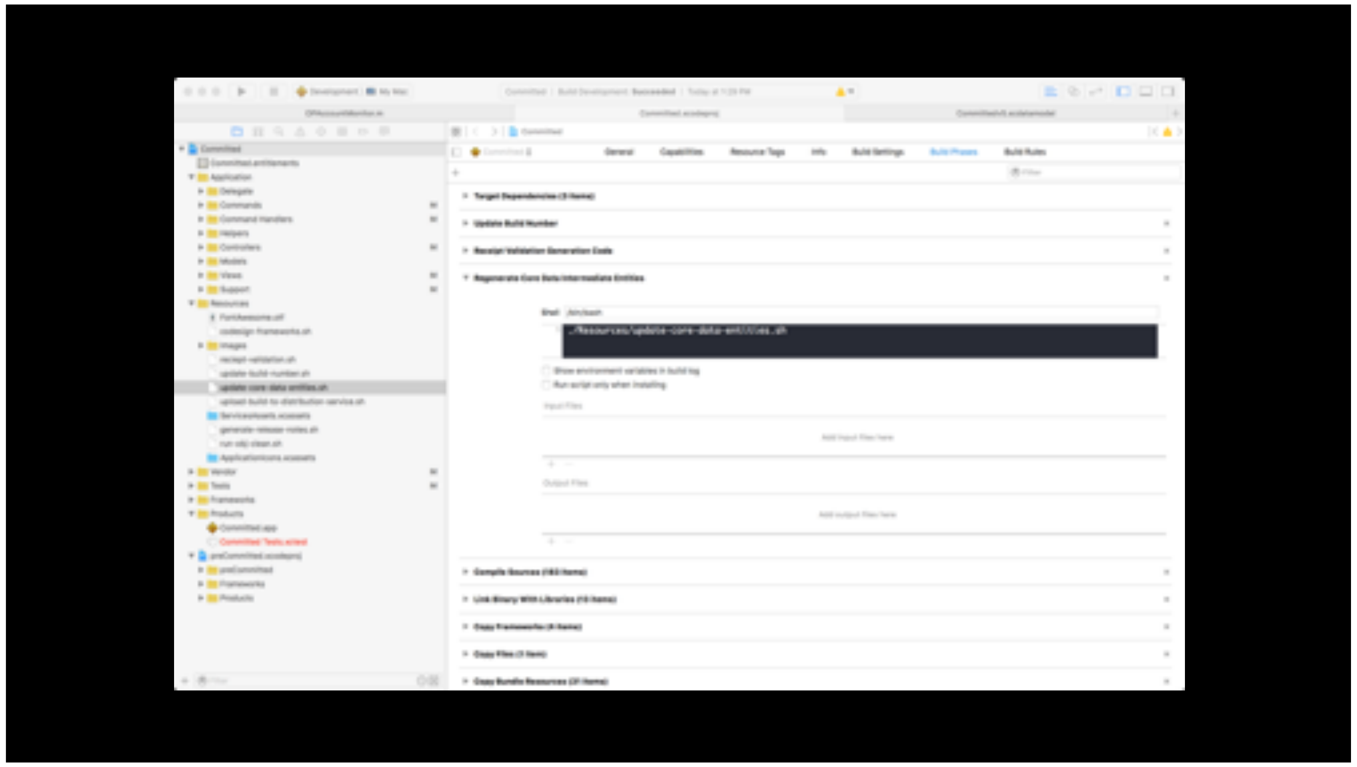


mogenerator implements the Generation Gap pattern



github.com/rentzsch/mogenerator

mogenerator implements the Generation Gap pattern



Target Dependencies (3 items)

Update Build Number

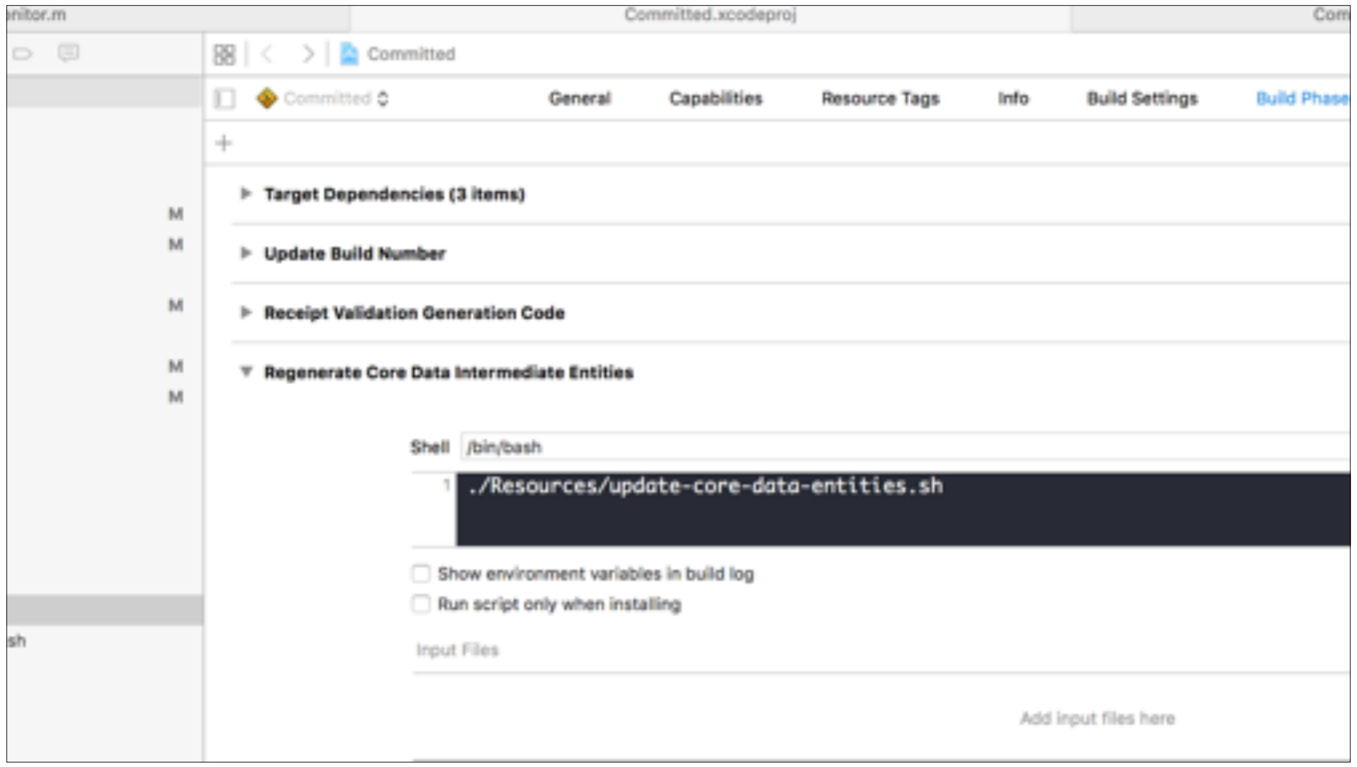
Receipt Validation Generation Code

Regenerate Core Data Intermediate Entities

Shell: /bin/bash

```
./Resources/update-core-data-entities.sh
```

Click the plus button to Add a New Run Script Phase to your project
Move it to complete just prior to your Compile Sources phase.
Now al your generated code will be up to date on every compile!



Shell /bin/bash

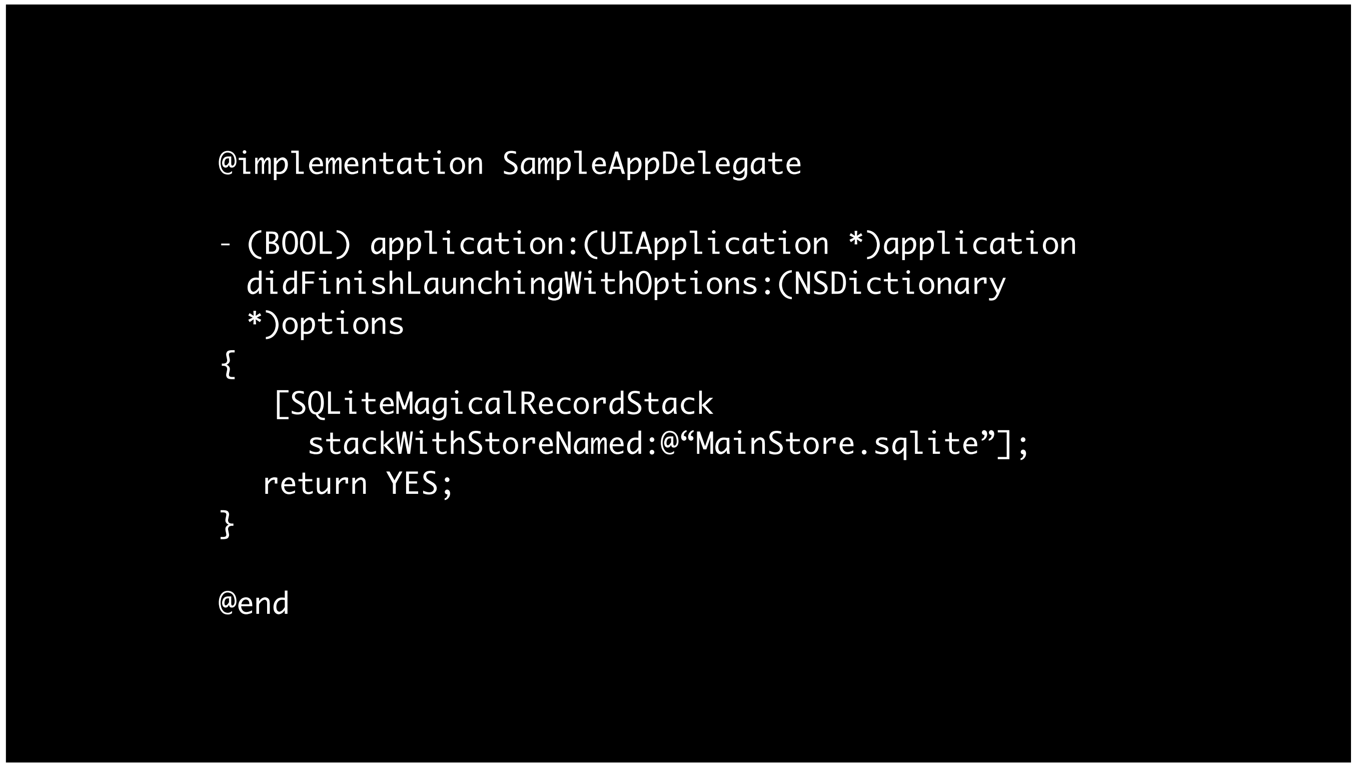
```
./Resources/update-core-data-entities.sh
```

☐ Show environment variables in build log
☐ Run script only when installing

Script source: https://gist.github.com/casademora/f115102b2e6530e2554c

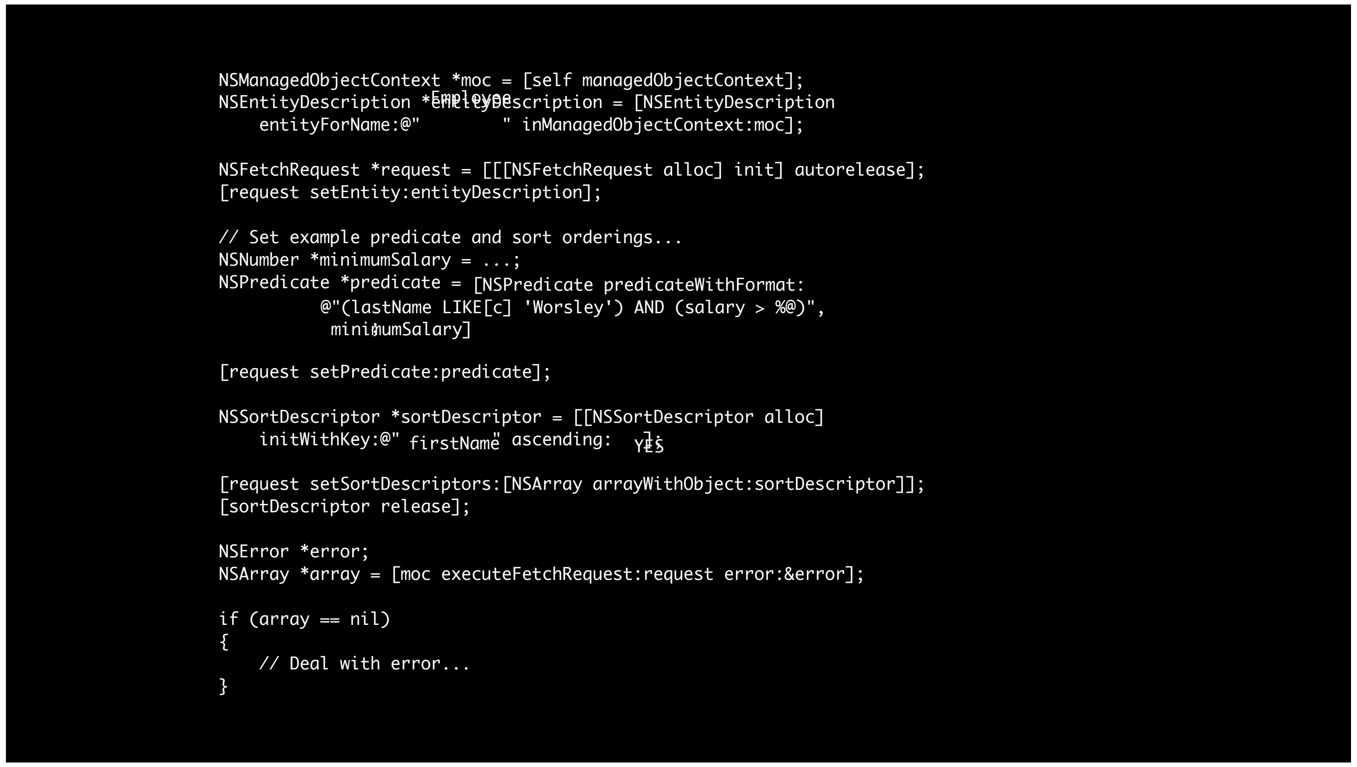
棒 MAGICALRECORD

3.0



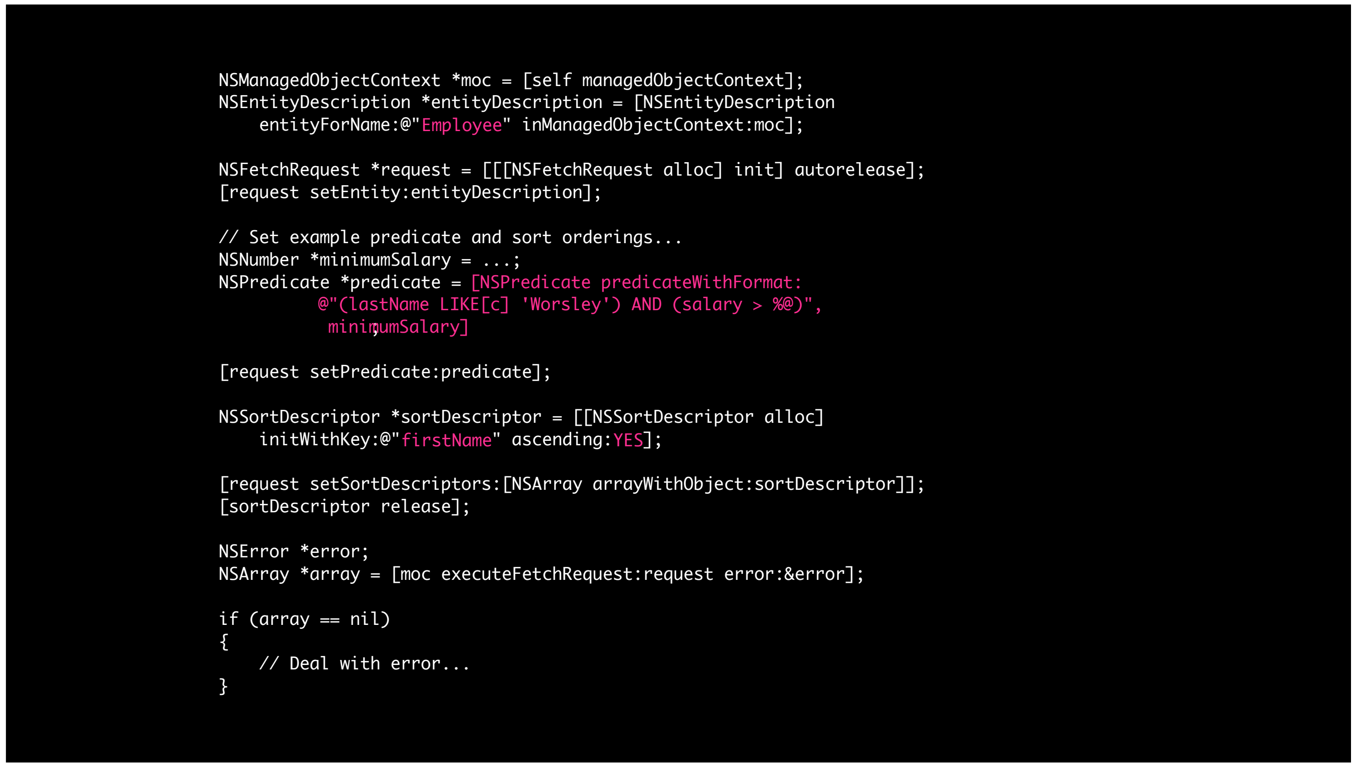
```
@implementation SampleAppDelegate

- (BOOL) application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary
  *)options
{
    [SQLiteMagicalRecordStack
      stackWithStoreNamed:@“MainStore.sqlite”];
   return YES;
}

@end
```

setting up a simple sqlite stack



```
NSManagedObjectContext *moc = [self managedObjectContext];
NSEntityDescription *entityDescription = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:moc];

NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
[request setEntity:entityDescription];

// Set example predicate and sort orderings...
NSNumber *minimumSalary = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
          @"(lastName LIKE[c] 'Worsley') AND (salary > %@)",
           minimumSalary];

[request setPredicate:predicate];

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
    initWithKey:@"firstName" ascending:YES];

[request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];
[sortDescriptor release];

NSError *error;
NSArray *array = [moc executeFetchRequest:request error:&error];

if (array == nil)
{
    // Deal with error...
}
```

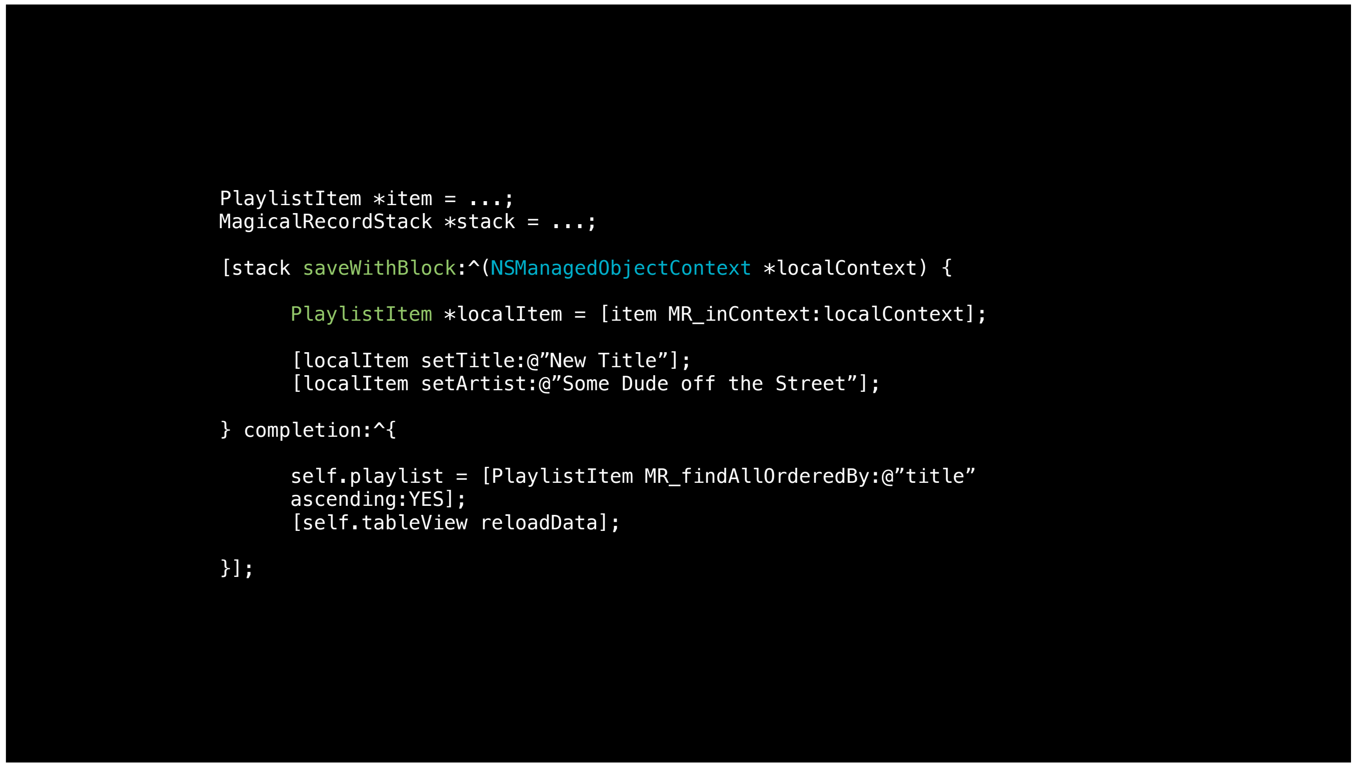
Fetch requests usually take a lot of code
There is a pretty clear pattern here, and a lot of repeated boilerplate code…
Let’s see exactly what is unique about this fetch request as opposed to others



```
NSManagedObjectContext *moc = [self managedObjectContext];
NSEntityDescription *entityDescription = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:moc];

NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
[request setEntity:entityDescription];

// Set example predicate and sort orderings...
NSNumber *minimumSalary = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
          @"(lastName LIKE[c] 'Worsley') AND (salary > %@)",
           minimumSalary]

[request setPredicate:predicate];

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
    initWithKey:@"firstName" ascending:YES];

[request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];
[sortDescriptor release];

NSError *error;
NSArray *array = [moc executeFetchRequest:request error:&error];

if (array == nil)
{
    // Deal with error...
}
```

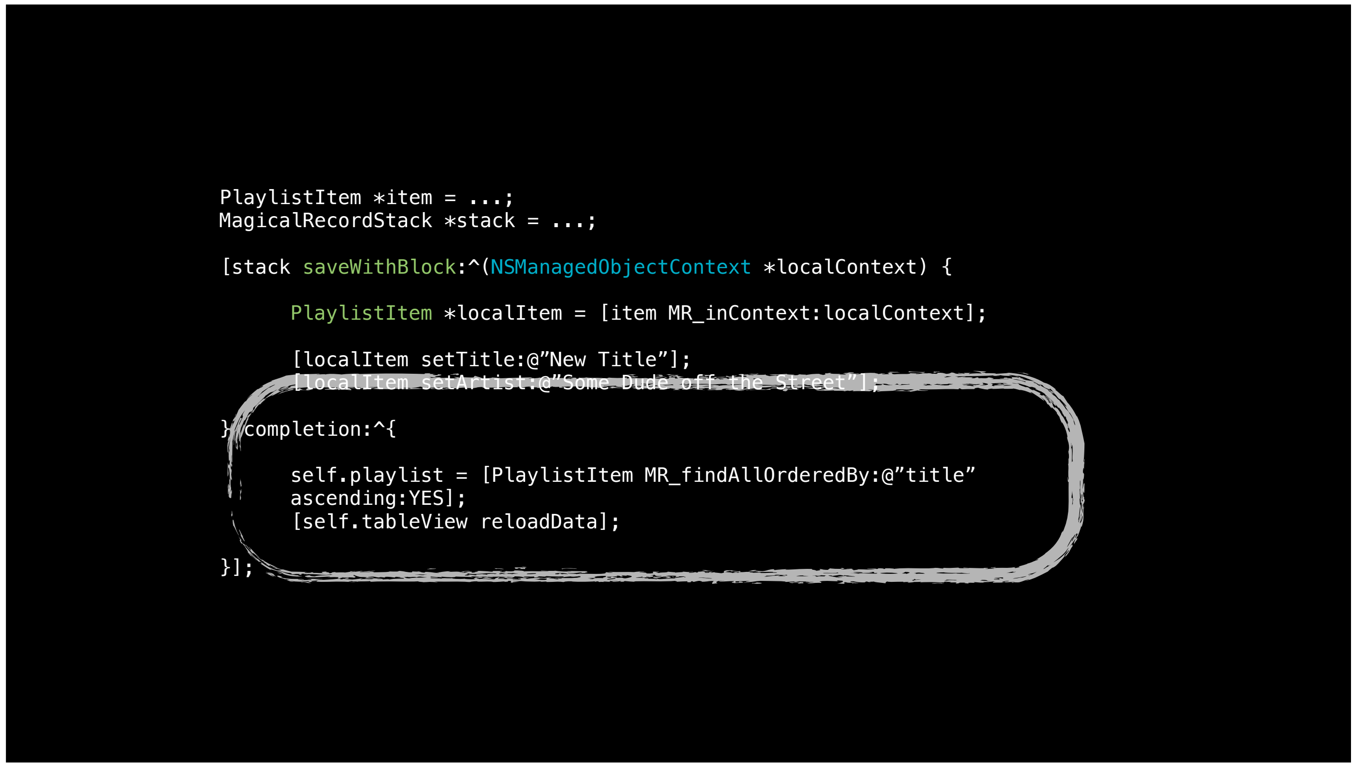
What we want is a way to make this code less boilerplate...we want our code to express the intent of this fetch request...

```
[Employee MR_findAllWithPredicate:[NSPredicate predicateWithFormat:
                  @"(lastName LIKE[c] 'Worsley') AND (salary > %@)",
                   minimumSalary]
             sortedBy:@"firstName"
            ascending: YES
            inContext:moc];
```



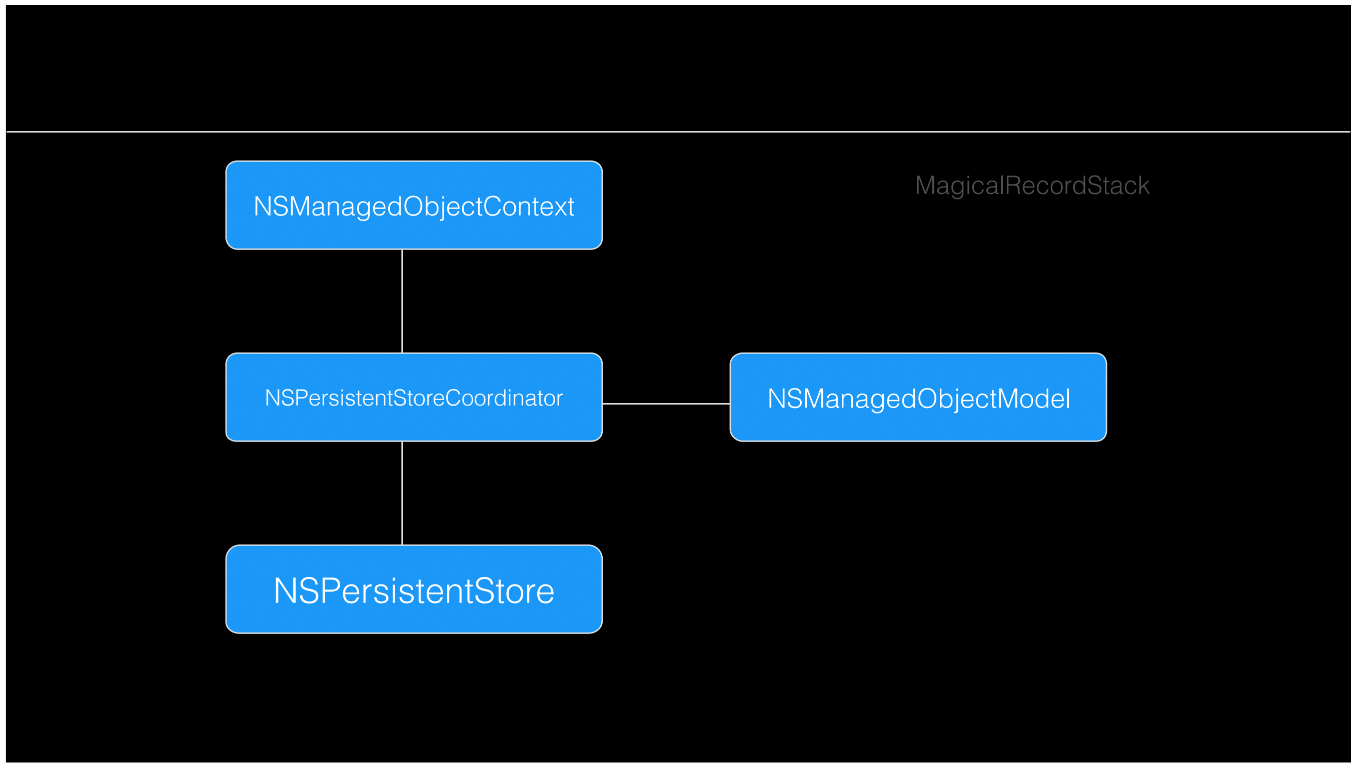
```
PlaylistItem *item = ...;
MagicalRecordStack *stack = ...;

[stack saveWithBlock:^(NSManagedObjectContext *localContext) {

      PlaylistItem *localItem = [item MR_inContext:localContext];

      [localItem setTitle:@"New Title"];
      [localItem setArtist:@"Some Dude off the Street"];

} completion:^{

      self.playlist = [PlaylistItem MR_findAllOrderedBy:@"title"
      ascending:YES];
      [self.tableView reloadData];

}];
```

What if you had a UIView animation style completion handler so that when your data is saved, you get a callback to update your UI! MagicalRecord does that for you!

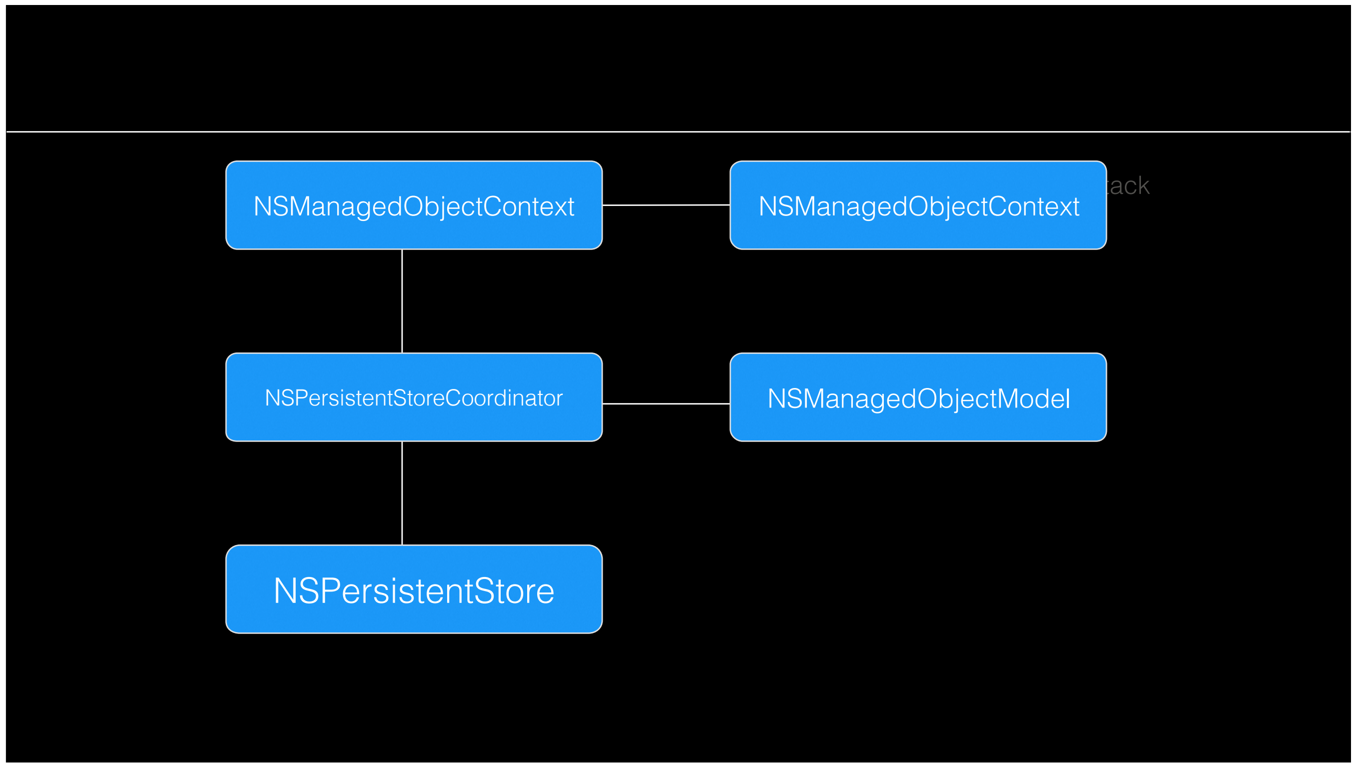


```
PlaylistItem *item = ...;
MagicalRecordStack *stack = ...;

[stack saveWithBlock:^(NSManagedObjectContext *localContext) {

      PlaylistItem *localItem = [item MR_inContext:localContext];

      [localItem setTitle:@"New Title"];
      [localItem setArtist:@"Some Dude off the Street"];

} completion:^{

      self.playlist = [PlaylistItem MR_findAllOrderedBy:@"title"
      ascending:YES];
      [self.tableView reloadData];

}];
```

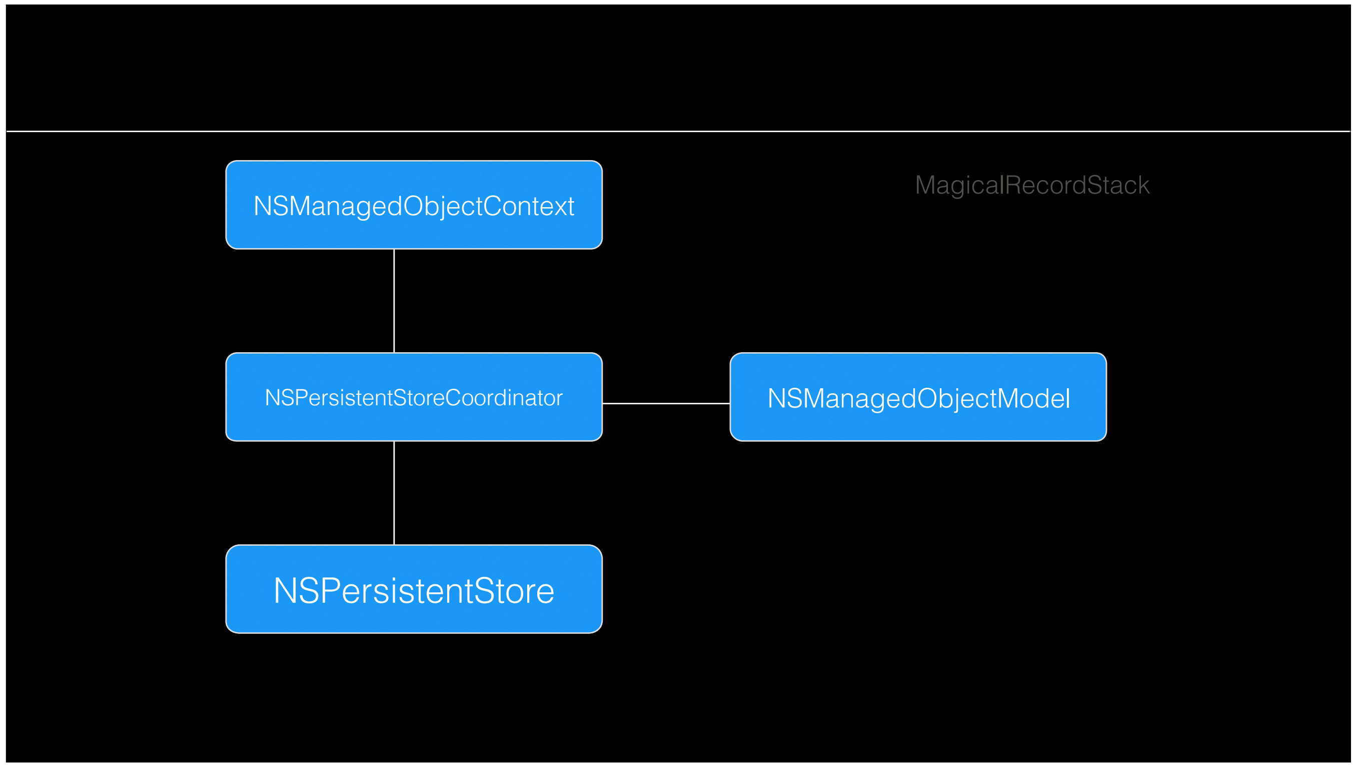
What if you had a UIView animation style completion handler so that when your data is saved, you get a callback to update your UI! MagicalRecord does that for you!

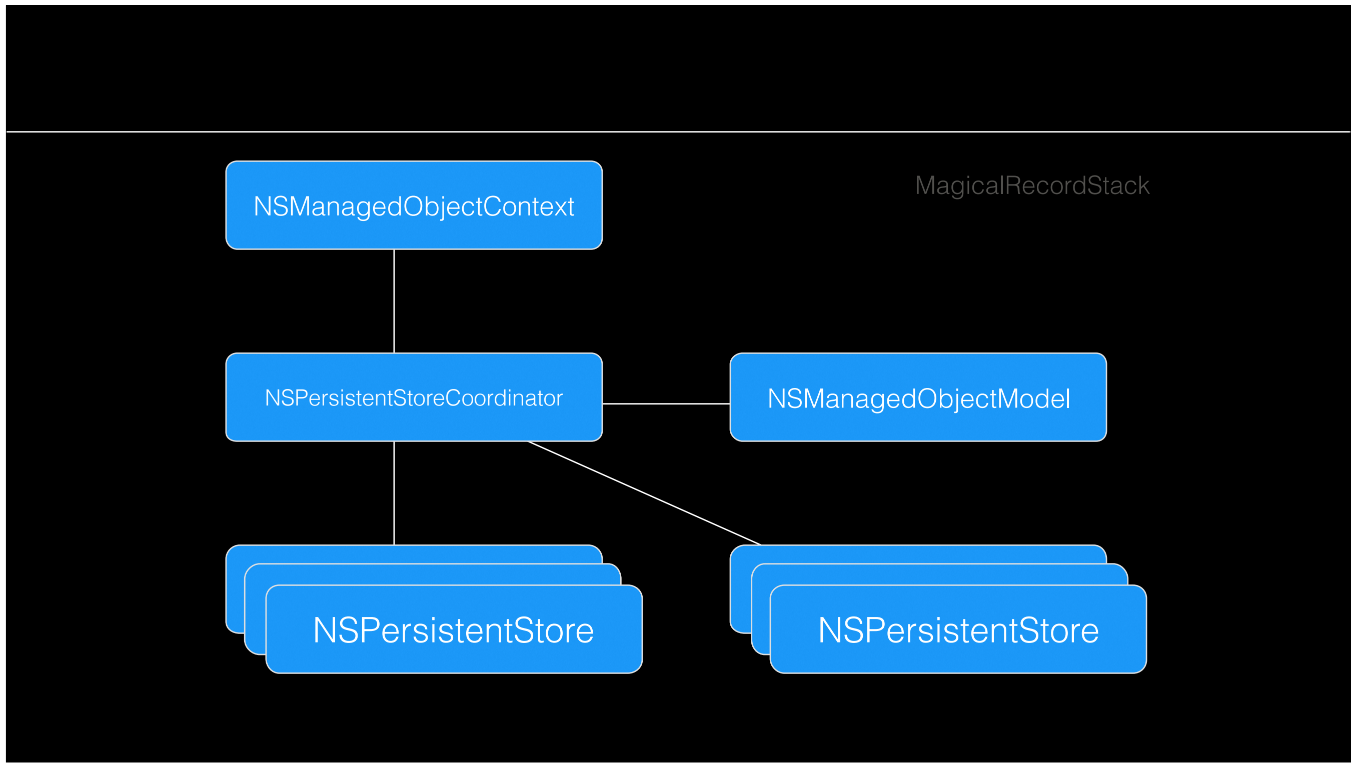


Let’s talk about how to configure Core Data differently

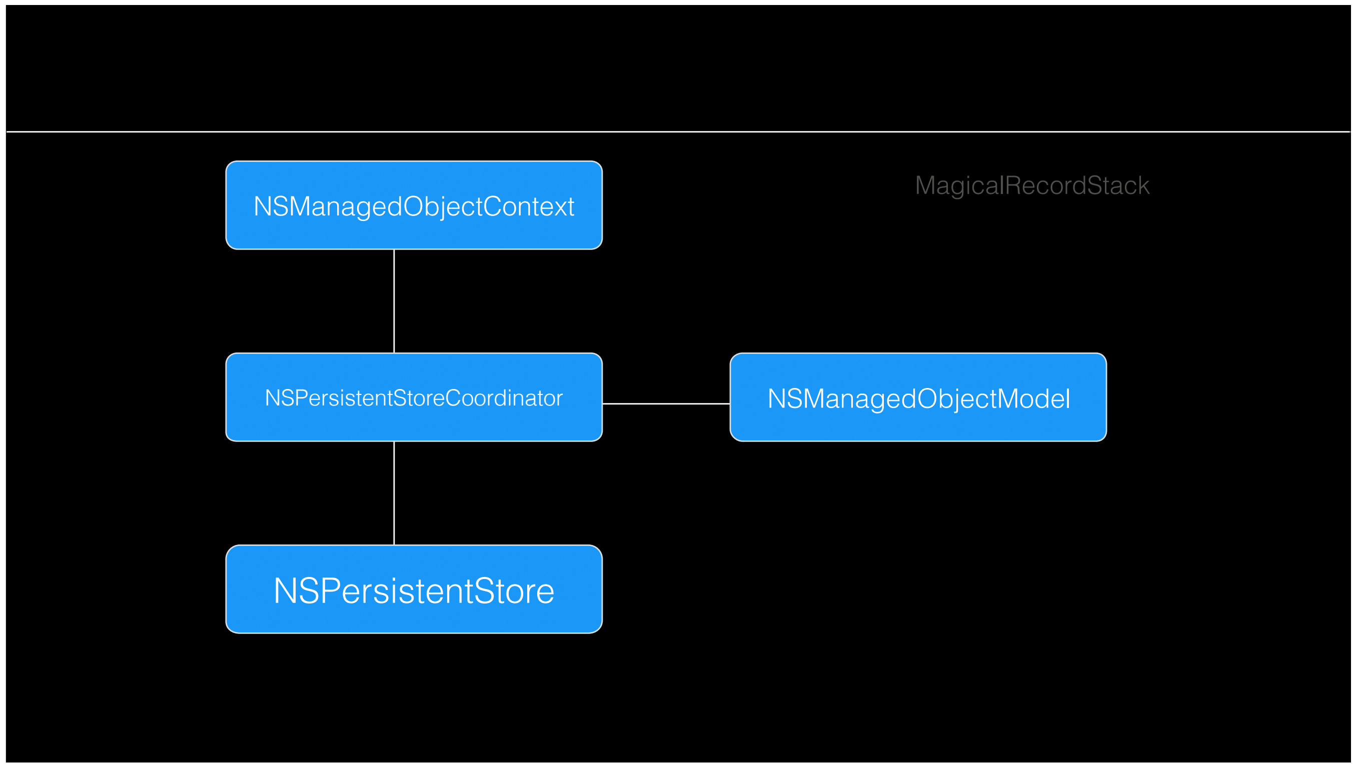


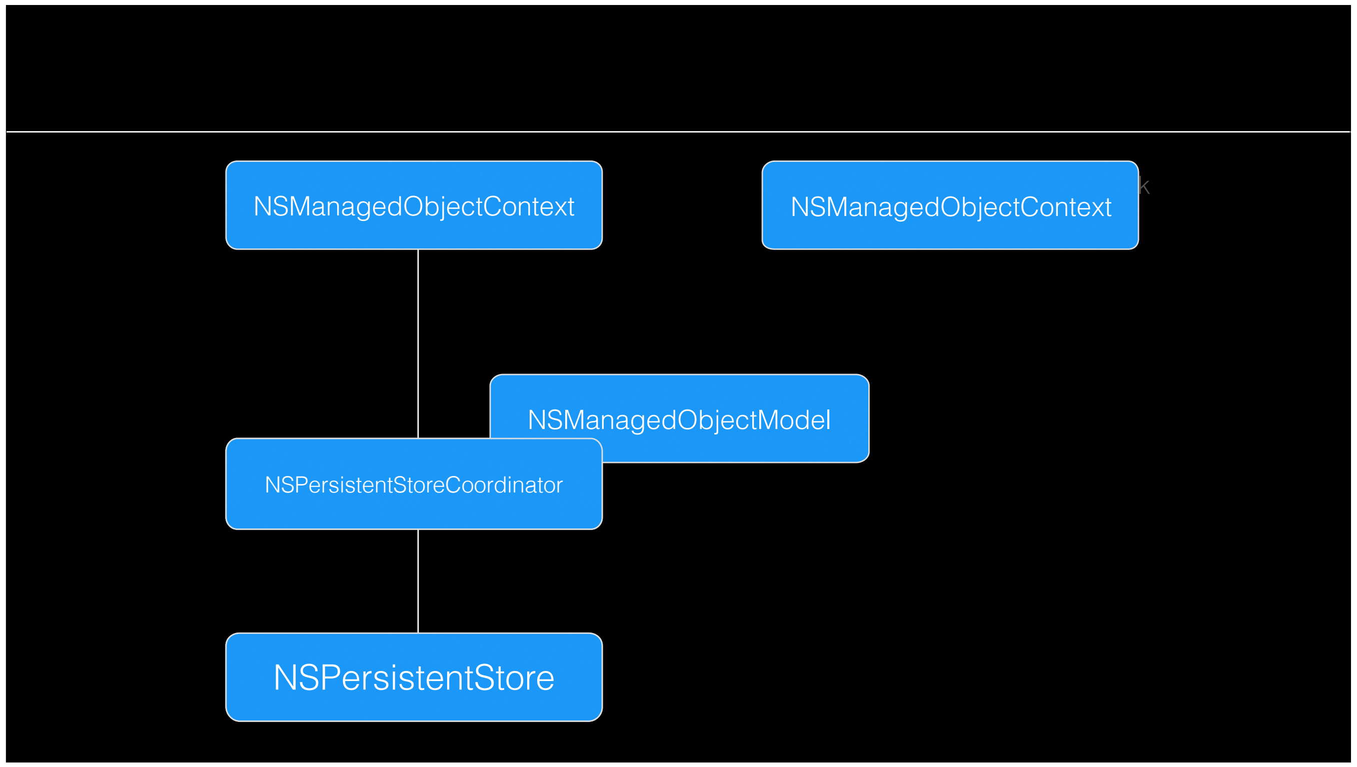
The second context is for saving in the background using a private queue context

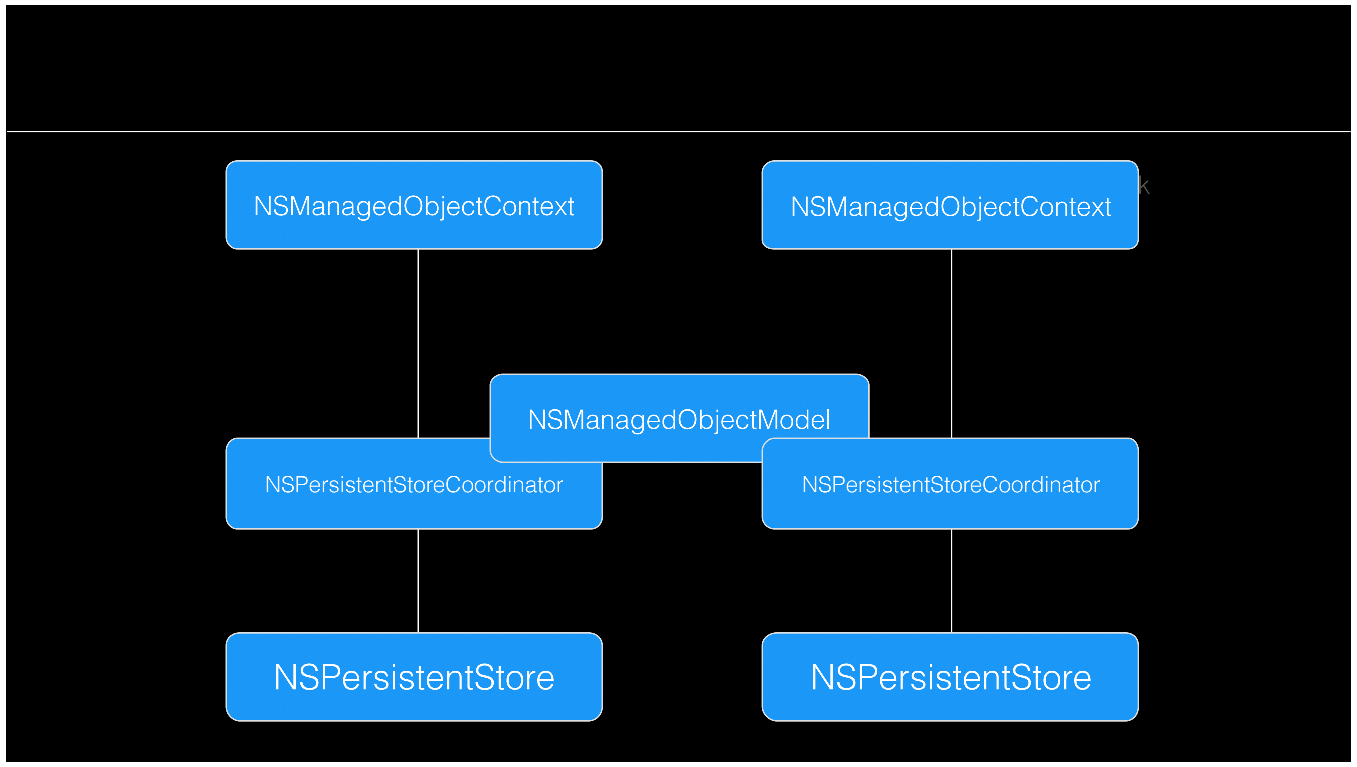

MagicalRecordStack
NSManagedObjectContext
NSPersistentStoreCoordinator
NSManagedObjectModel
NSPersistentStore

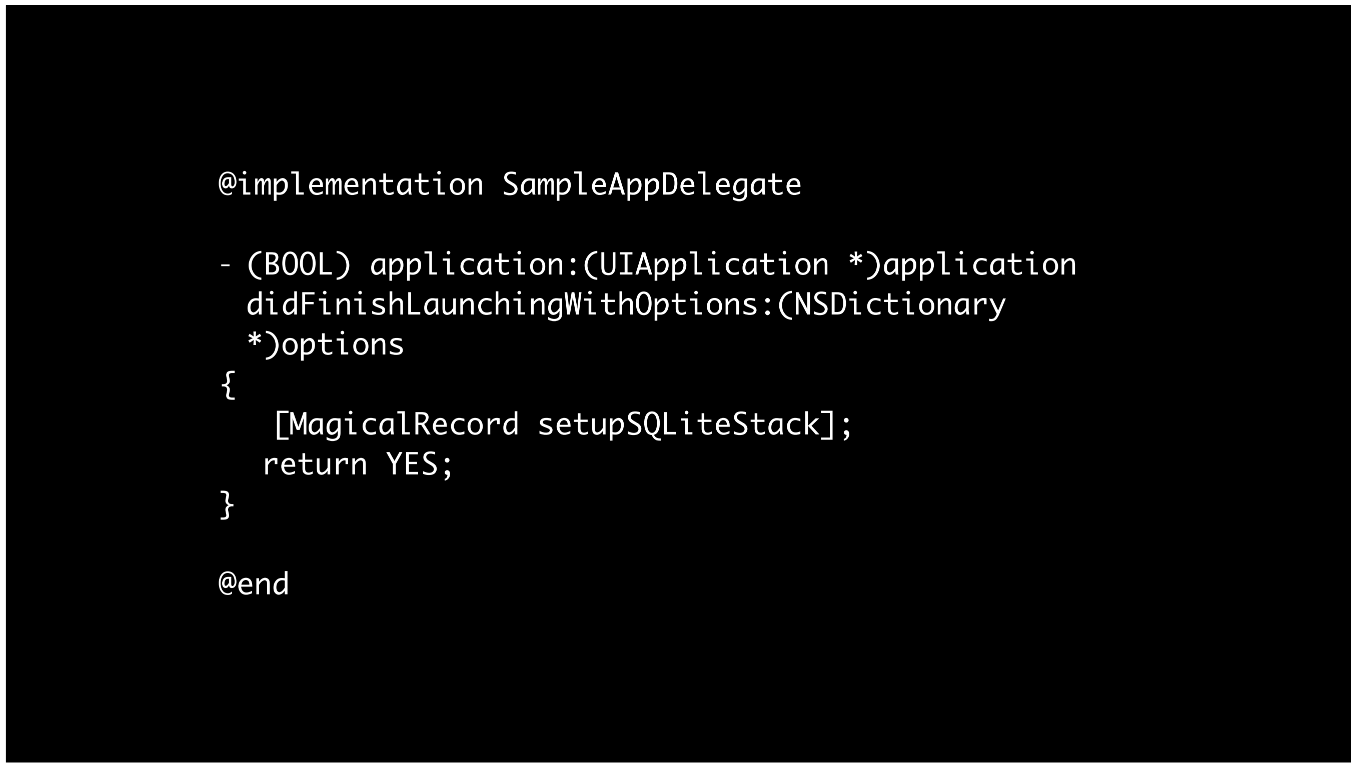


Multiple Persistent Stores is great for having separate “documents”


MagicalRecordStack
NSManagedObjectContext
NSPersistentStoreCoordinator
NSManagedObjectModel
NSPersistentStore

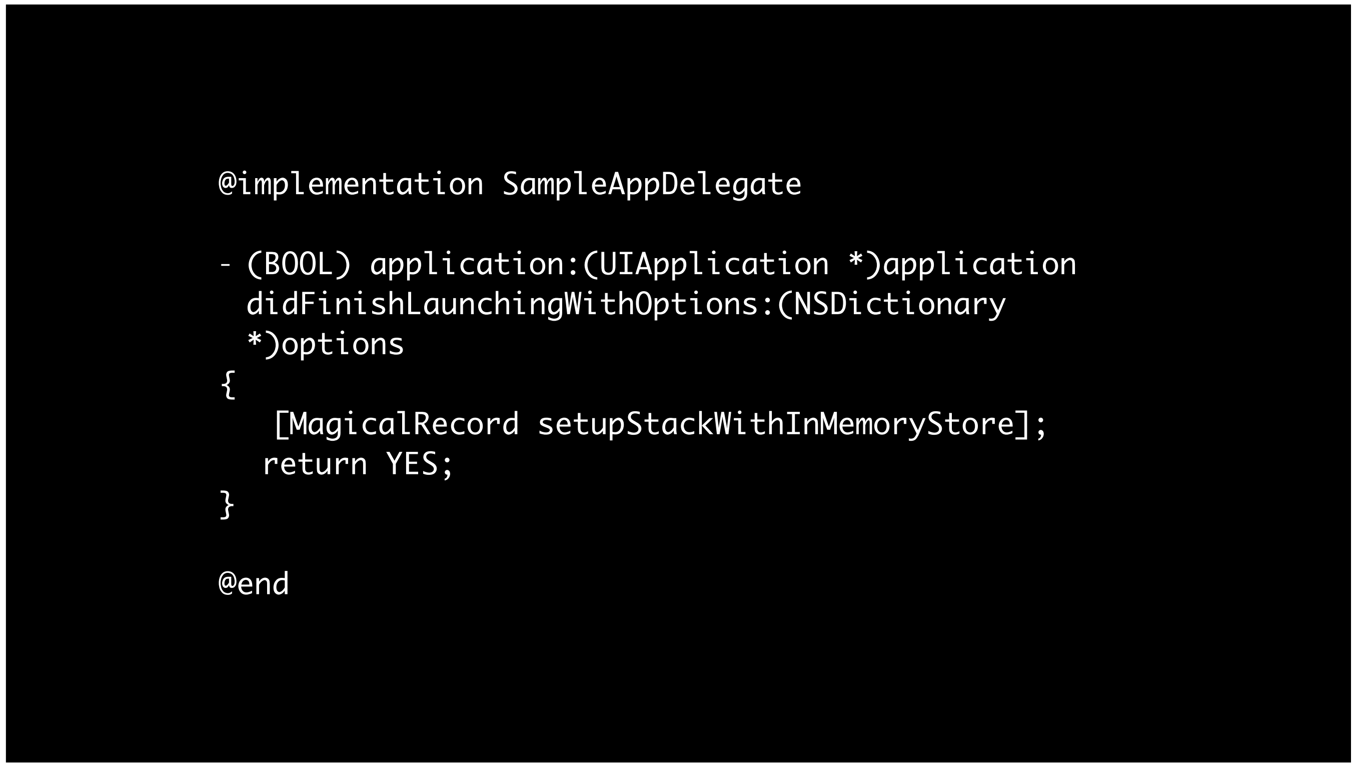

NSManagedObjectContext
NSManagedObjectContext
NSManagedObjectModel
NSPersistentStoreCoordinator
NSPersistentStore


NSManagedObjectContext
NSManagedObjectContext
NSManagedObjectModel
NSPersistentStoreCoordinator
NSPersistentStoreCoordinator
NSPersistentStore
NSPersistentStore



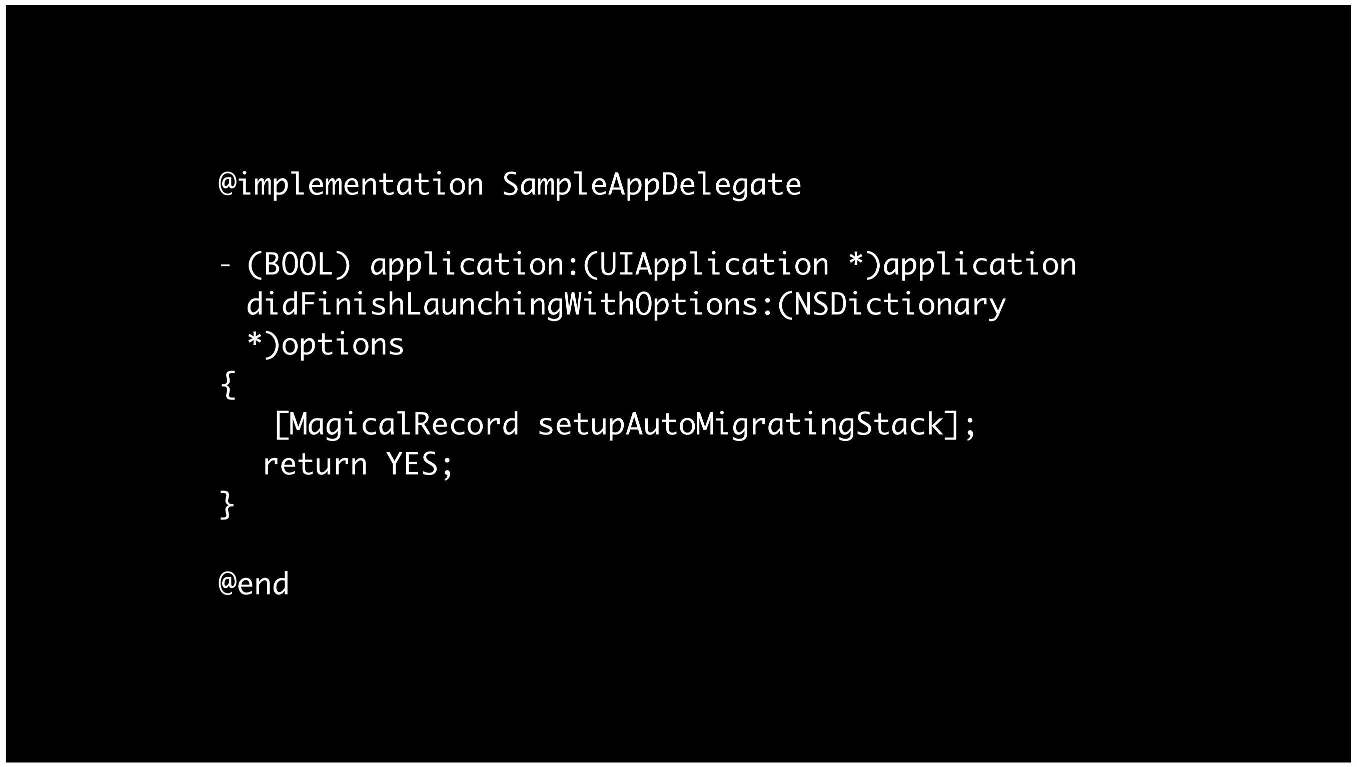
```
@implementation SampleAppDelegate

- (BOOL) application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary
  *)options
{
    [MagicalRecord setupSQLiteStack];
   return YES;
}

@end
```

setting up a simple sqlite stack



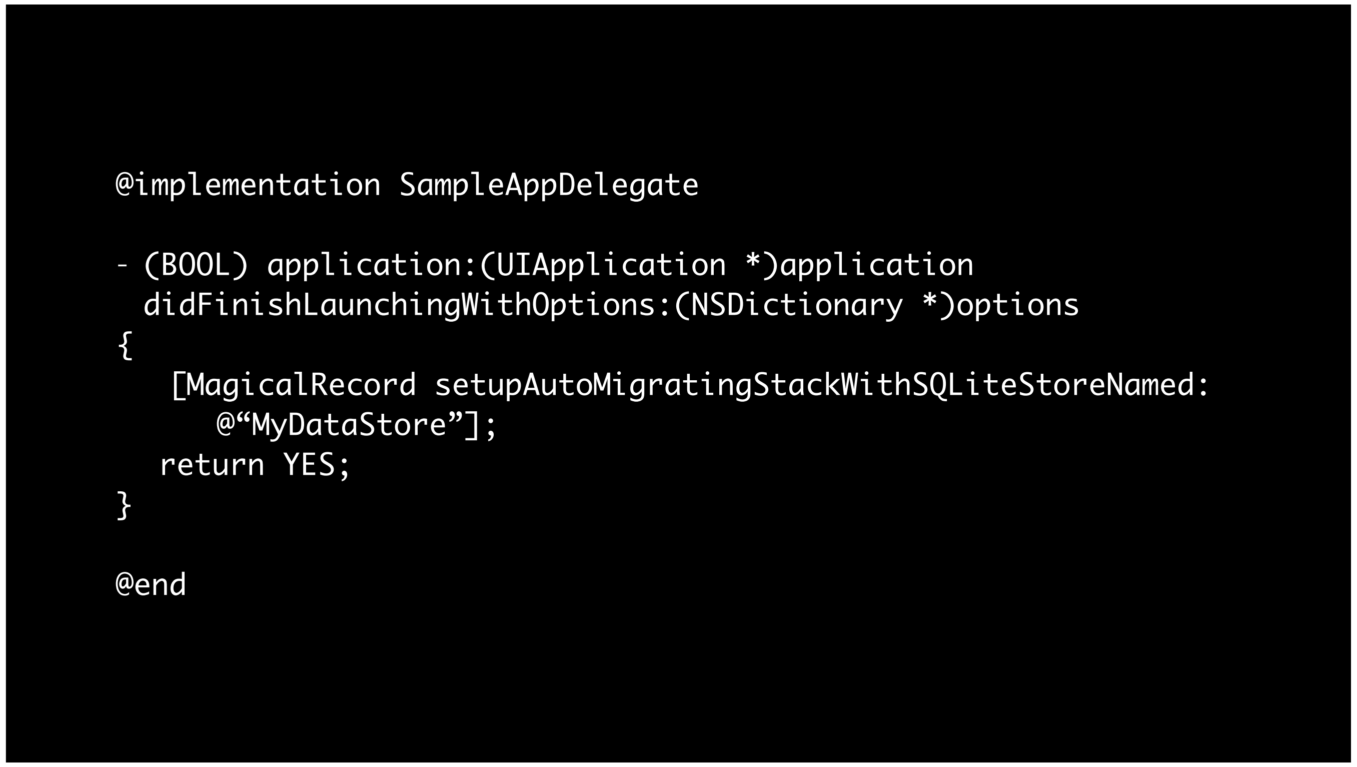
```
@implementation SampleAppDelegate

- (BOOL) application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary
  *)options
{
    [MagicalRecord setupStackWithInMemoryStore];
   return YES;
}

@end
```

setting up an in memory stack



```
@implementation SampleAppDelegate

- (BOOL) application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary
  *)options
{
    [MagicalRecord setupAutoMigratingStack];
   return YES;
}

@end
```

setting up an auto migrating sqlite stack



```
@implementation SampleAppDelegate

- (BOOL) application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)options
{
    [MagicalRecord setupAutoMigratingStackWithSQLiteStoreNamed:
       @“MyDataStore”];
   return YES;
}

@end
```

setting up an auto migrating sqlite stack with a custom store name

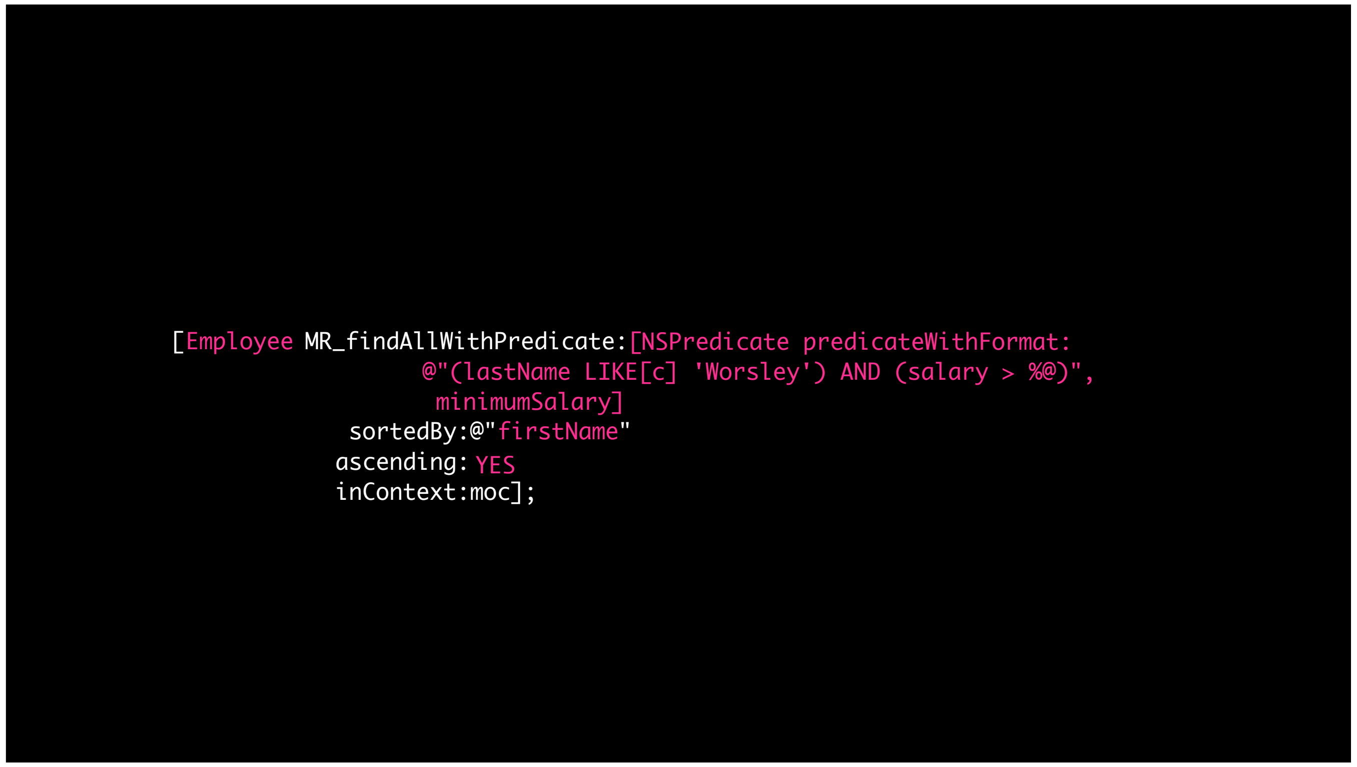
MagicalRecord is open source

But what about swift? This whole presentation has been showing you code in Objective C..

Objective C had a huge influence on the MagicalRecord API design. Making MagicalRecord fit into swift isn’t just simply a rewrite/reimplement operation.

- Logging Macros
- Data import library

- fetching method naming conventin



```
[Employee MR_findAllWithPredicate:[NSPredicate predicateWithFormat:
                   @"(lastName LIKE[c] 'Worsley') AND (salary > %@)",
                    minimumSalary]
             sortedBy:@"firstName"
            ascending: YES
            inContext:moc];
```

- MR_ prefix naming convention is due to the nature of categories in ObjC…

## Some Code Reads Strangely

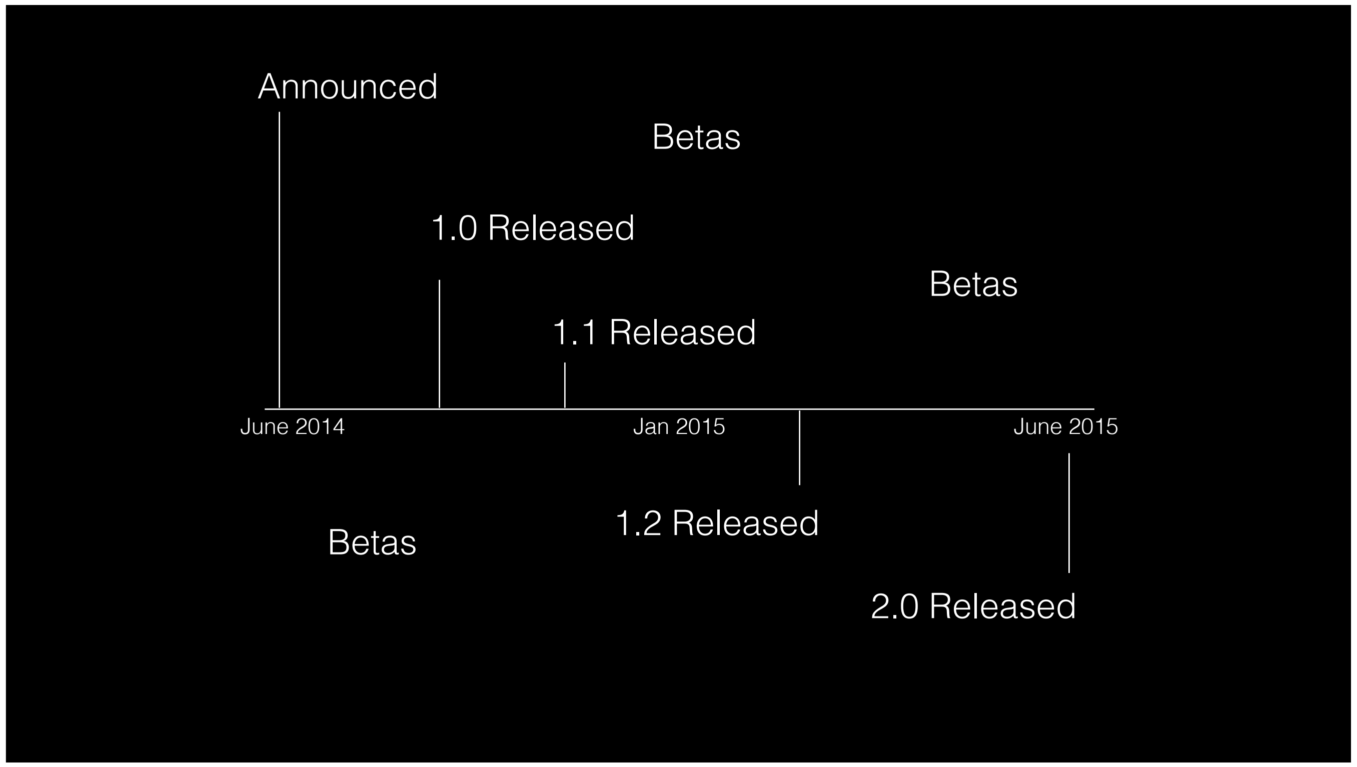
Examples like the MR_ prefix. It was a hack in ObjC, and carried over to Swift it makes some APIs feel like they don’t fit in

Magical Record is LARGE. It’s more than just a collection of simple helper methods around CoreData now…

MR has a lot of useful features.

- Stack management
- Thread/Queue/Context management
- Request helpers
- Logging
- Simple Data Importing

Swift moves fast



The Swift Language has moved super fast…

<Show timeline of swift releases with features and breaking changes>

Announced June 2, 2014

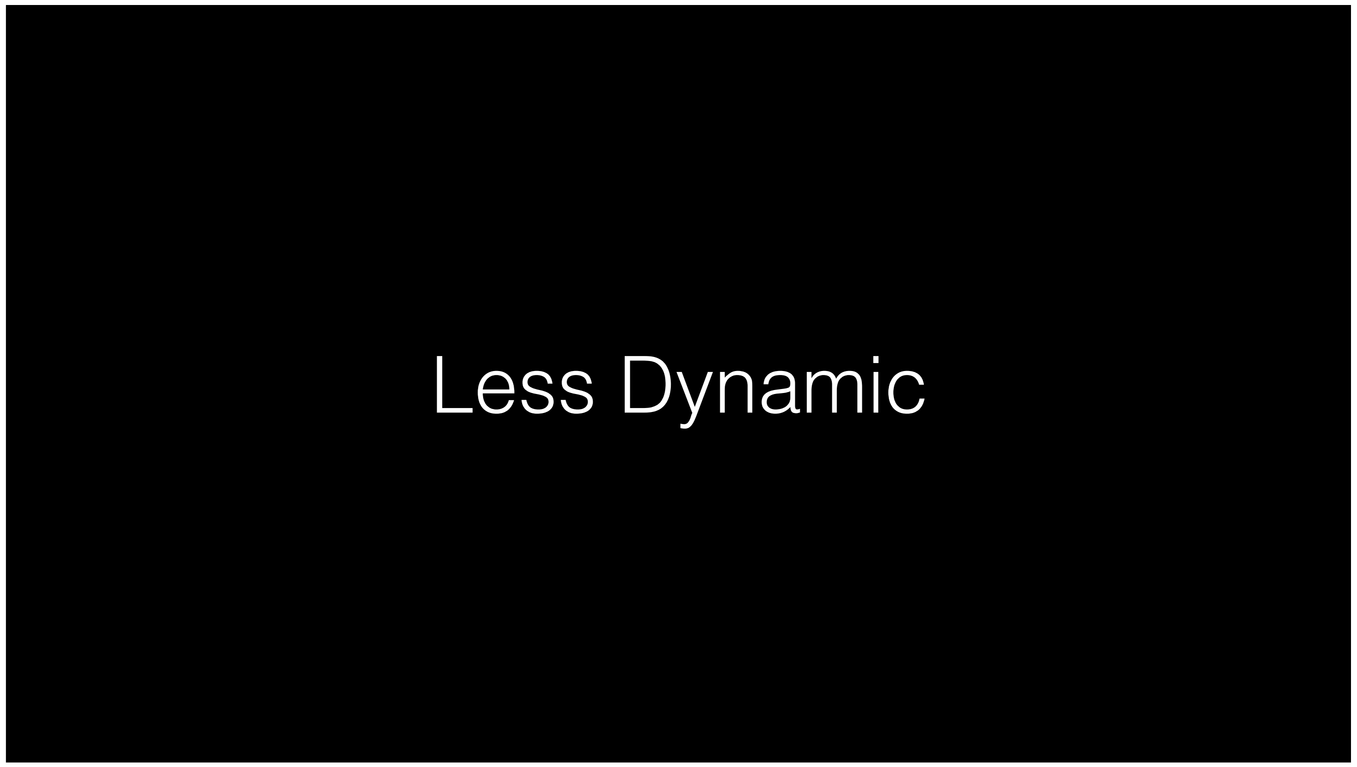
1.0 - Sept 9, 2014

1.1 - Oct 2, 2014 (failable initializers)

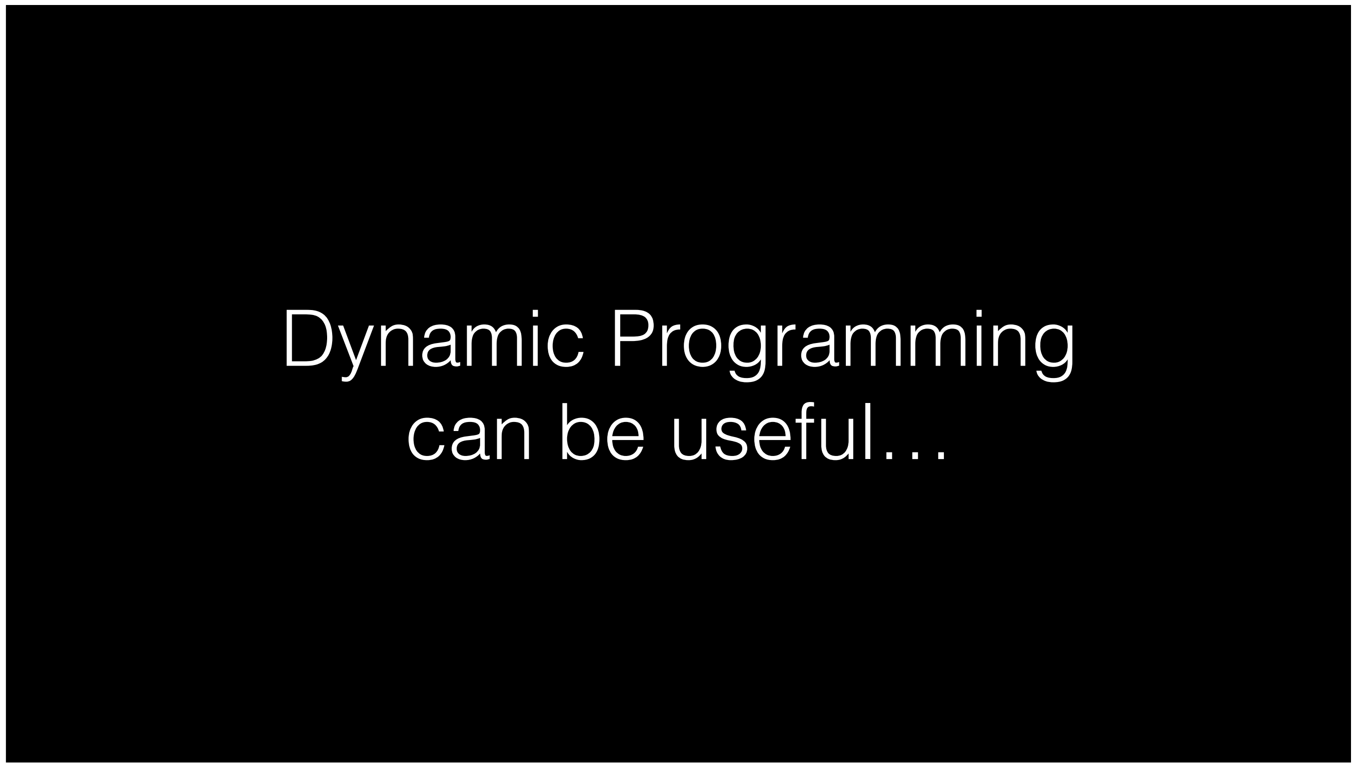
1.2 - Feb 9, 2105 (compiler improvements, as!, nullability for ObjC code, compound if let statements)

2.0 - June 2015 (guard, protocol extensions)

2.1 - already on the horizon with Xcode 7.1 beta2

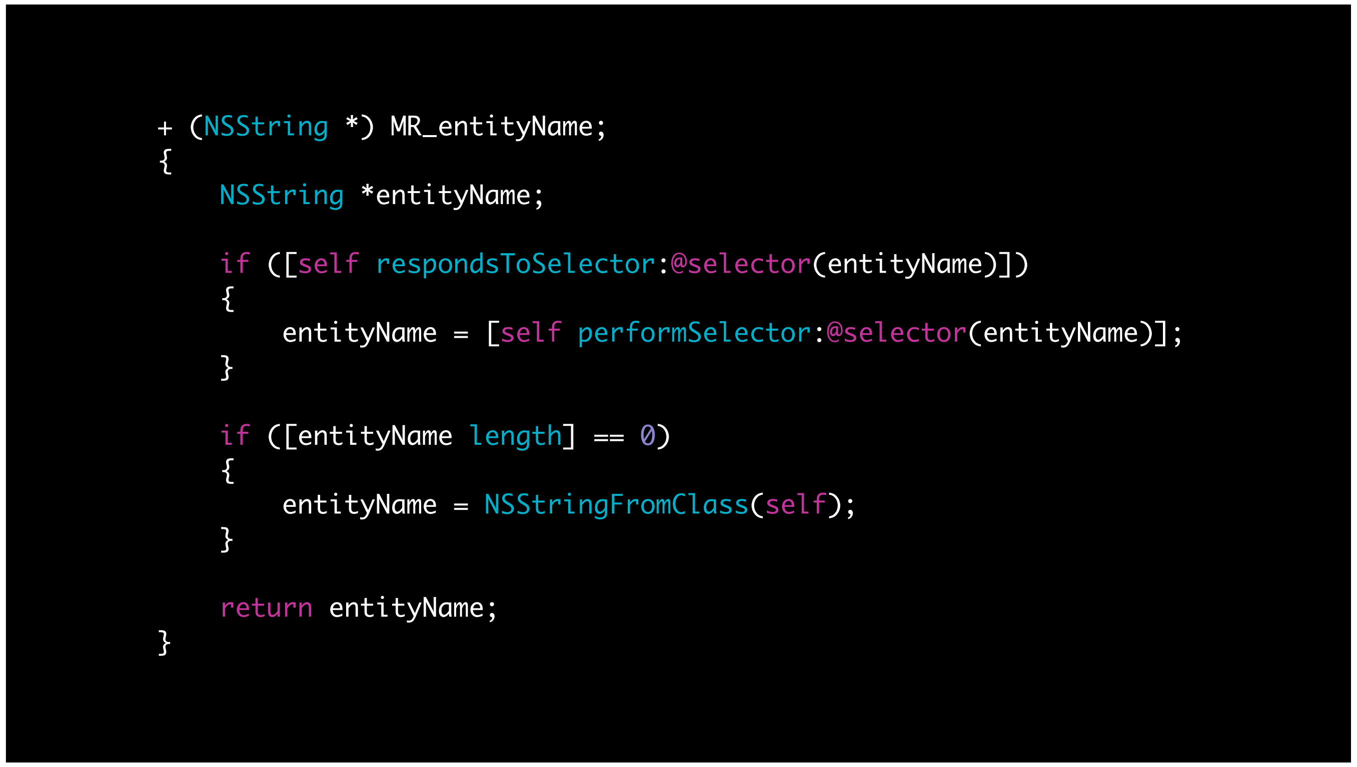

## Less Dynamic

Swift has started as a less dynamic language despite being implemented on the ObjC runtime. Swift calls have been optimized to remove the dynamic dispatch calls that made ObjC flexible…but also more easier to write bad bugs. MagicalRecord has taken advantage of the dynamic nature of ObjC.



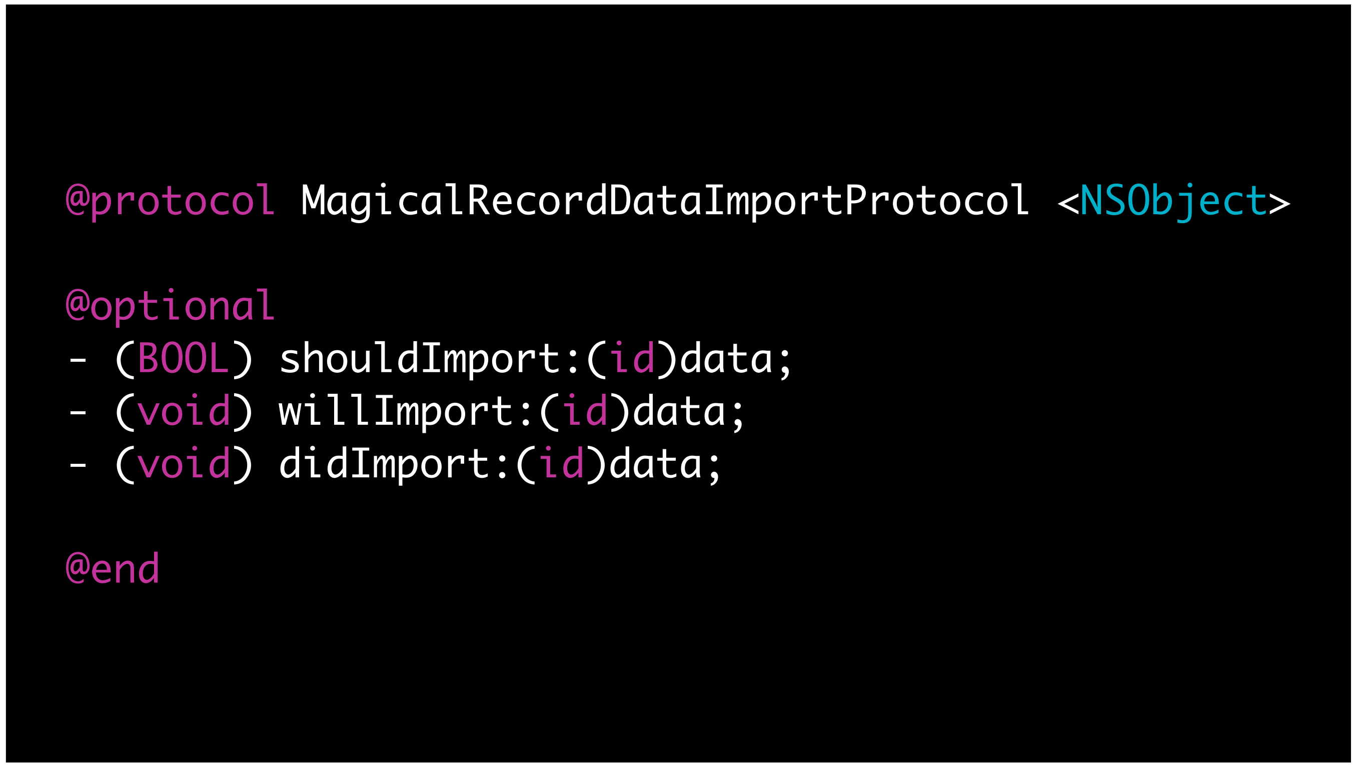
## Dynamic Programming can be useful…

Some solutions are just more simple to implement using Dynamic Runtime Programming. Like the data import library callbacks…
Simpler is relative because you still need to make it work. But as far as code goes, the amount of effort to dynamically call a method based on a property name is far simpler to write and consume using dynamic programming.



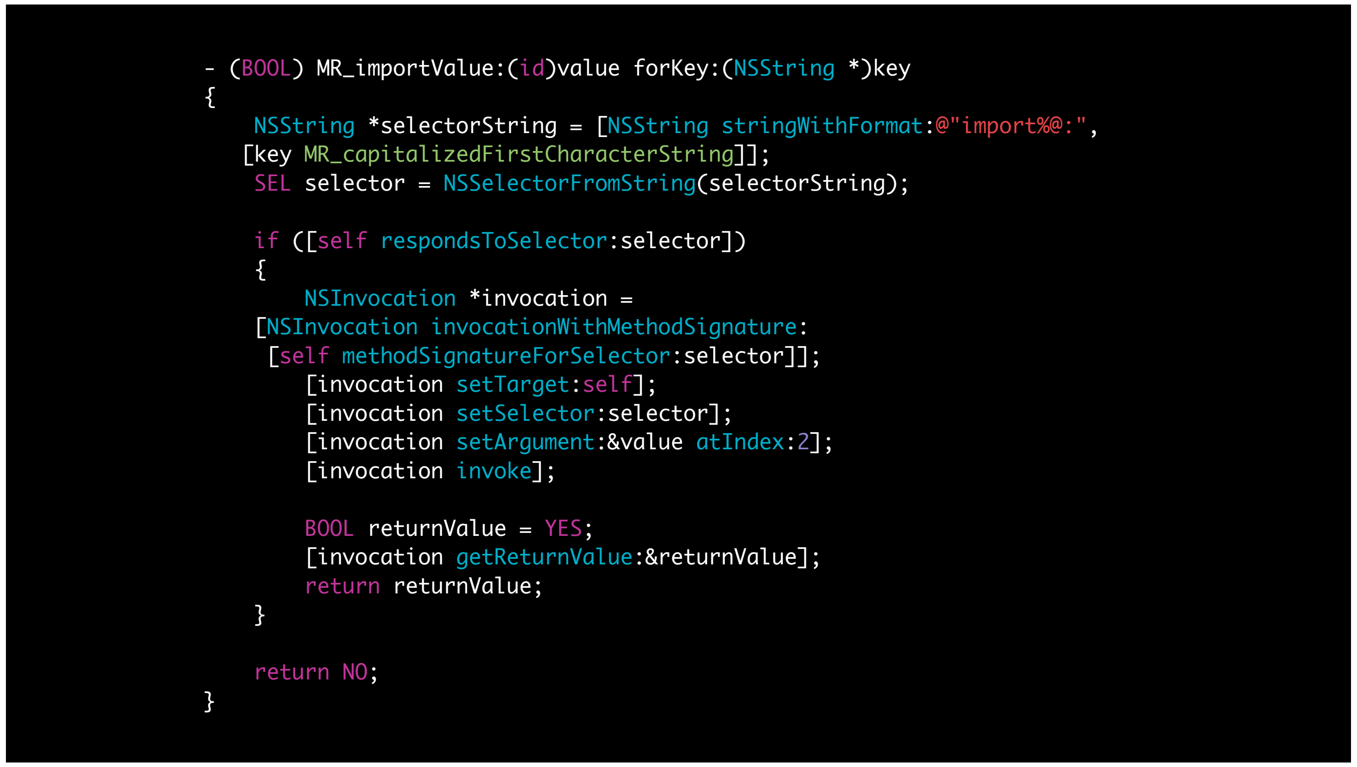
```
+ (NSString *) MR_entityName;
{
    NSString *entityName;

    if ([self respondsToSelector:@selector(entityName)])
    {
        entityName = [self performSelector:@selector(entityName)];
    }

    if ([entityName length] == 0)
    {
        entityName = NSStringFromClass(self);
    }

    return entityName;
}
```

The mapping between the entity name and the class name is a simple solution in MagicalRecord.



```objc
@protocol MagicalRecordDataImportProtocol <NSObject>

@optional
- (BOOL) shouldImport:(id)data;
- (void) willImport:(id)data;
- (void) didImport:(id)data;

@end
```

When importing data, we can create a simple protocol to call back a the client that triggered the import at the proper time. This protocol can be implemented and triggered using swift. So it doesn’t matter if it’s written in ObjC or Swift. It’s protocol oriented



```
- (BOOL) MR_importValue:(id)value forKey:(NSString *)key
{
    NSString *selectorString = [NSString stringWithFormat:@"import%@:",
   [key MR_capitalizedFirstCharacterString]];
    SEL selector = NSSelectorFromString(selectorString);

    if ([self respondsToSelector:selector])
    {
        NSInvocation *invocation =
    [NSInvocation invocationWithMethodSignature:
     [self methodSignatureForSelector:selector]];
        [invocation setTarget:self];
        [invocation setSelector:selector];
        [invocation setArgument:&value atIndex:2];
        [invocation invoke];

        BOOL returnValue = YES;
        [invocation getReturnValue:&returnValue];
        return returnValue;
    }

    return NO;
}
```

Category on NSManagedObject to import a single value

This is where dynamic programming is useful. There is a convention on the NSManagedObjects so that if you have custom import logic, you can implement it one attribute at a time and the import method is named for the attribute. You can see an example of how this is useful here…

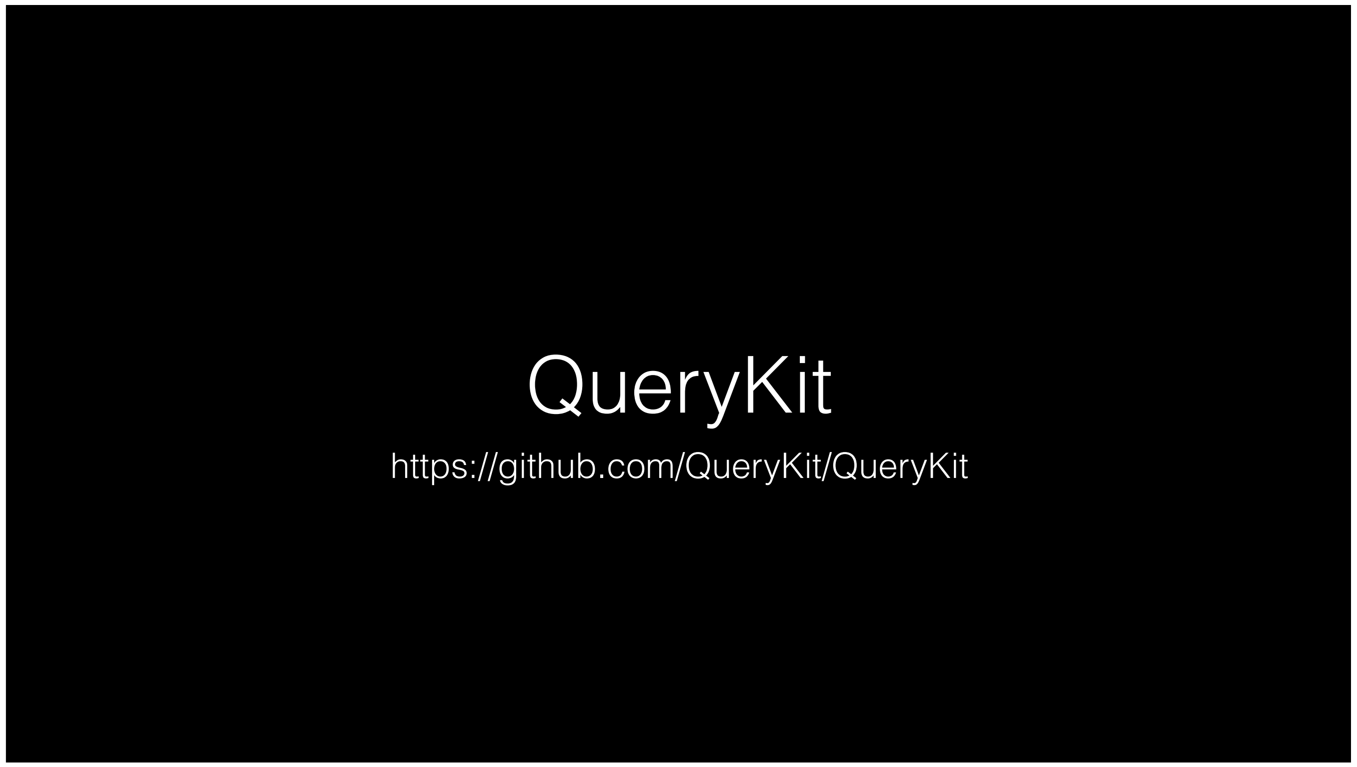
```
- (BOOL) importName:(id)name;
{
    NSArray *components = [name componentsSeparatedByString:@"/"];
    self.name = [components lastObject];
    self.fullName = name;
    if ([components count] > 1)
    {
        self.owner = [components firstObject];
    }
    return self.name != nil;
}
```

Custom import logic to read a persons name from a Bitbucket Repository

## SugarRecord

https://github.com/SugarTeam/SugarRecord

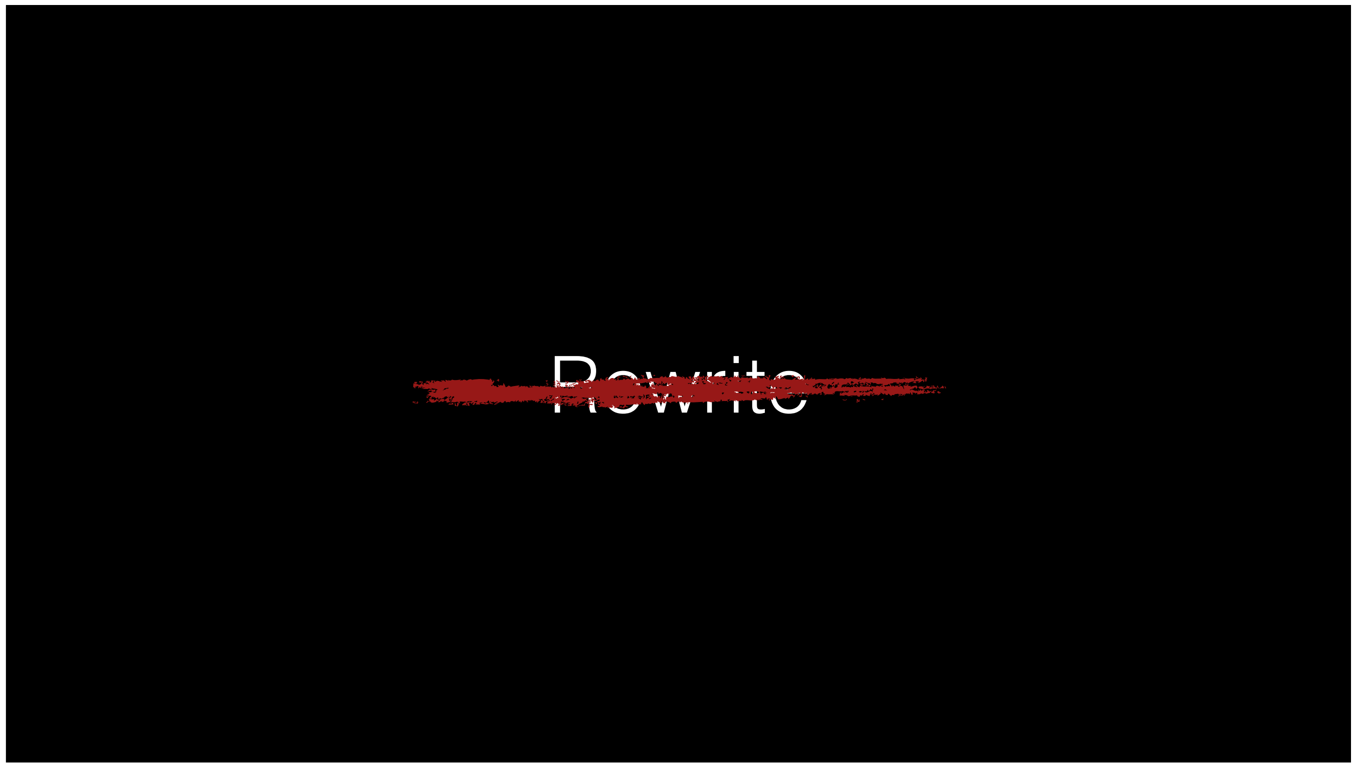
SugarRecord came out with a good fresh start



QueryKit

https://github.com/QueryKit/QueryKit

QueryKit resembles something like Linq. The approach feels like a Domain Specific Language.

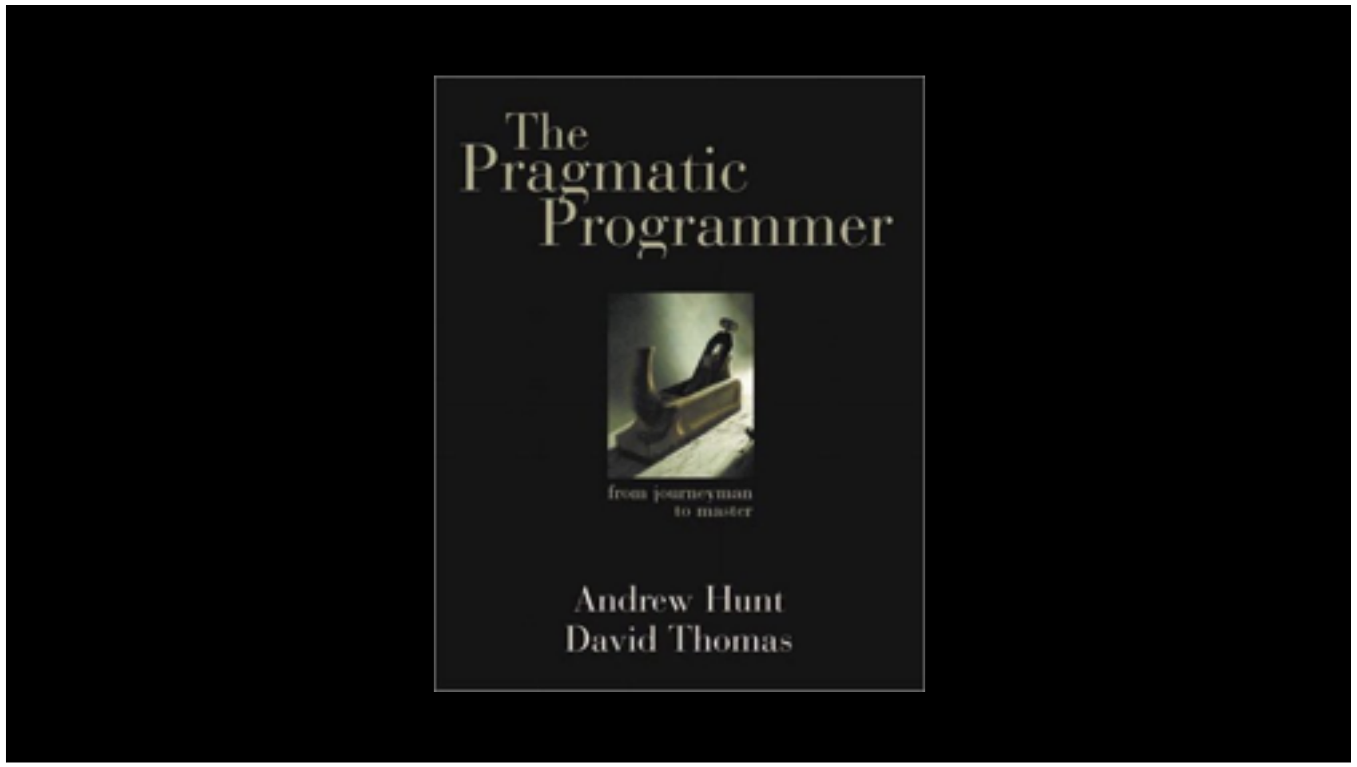


~~Rewrite~~

Decided not to rewrite the codebase!

Instead we will move forward with some Swift only extensions that will not be visible to ObjC code.

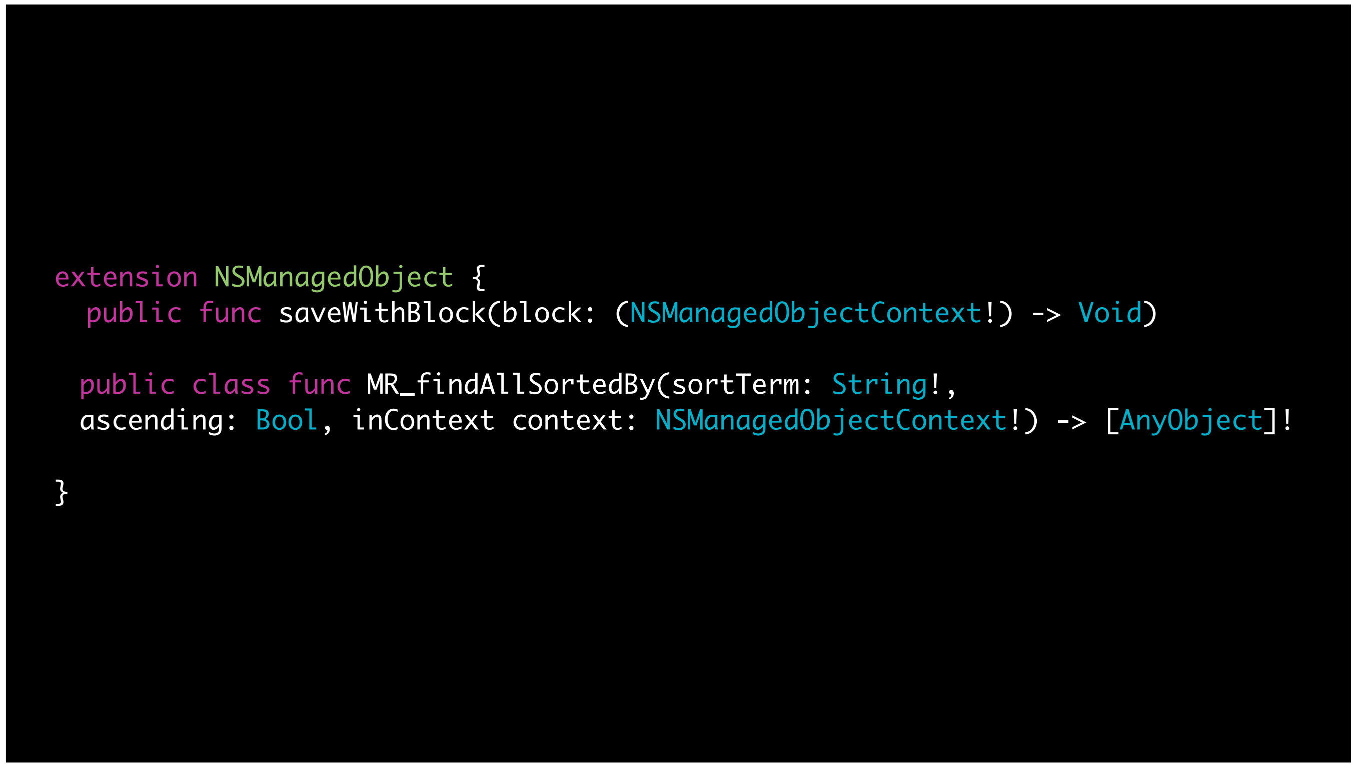
Discuss API design decisions here.



Trying to be a pragmatic programmer myself, I didn’t want to rewrite what wasn't broken. MagicalRecord is great for CoreData. Remember CoreData is a very dynamic, and key value based approach to reading and writing data.

## Interoperability

Apple spent a lot of time and effort in WWDC 2014 to promote Swift/ObjC interoperability. In fact, it works amazingly well because of the architecture of the language implementation being built on top of the Objective C runtime. In this case, there isn't a strong need to rewrite, since the codebase still works. What is needed is a way to fit the library into the new language paradigm without breaking backward compatibility with old code.

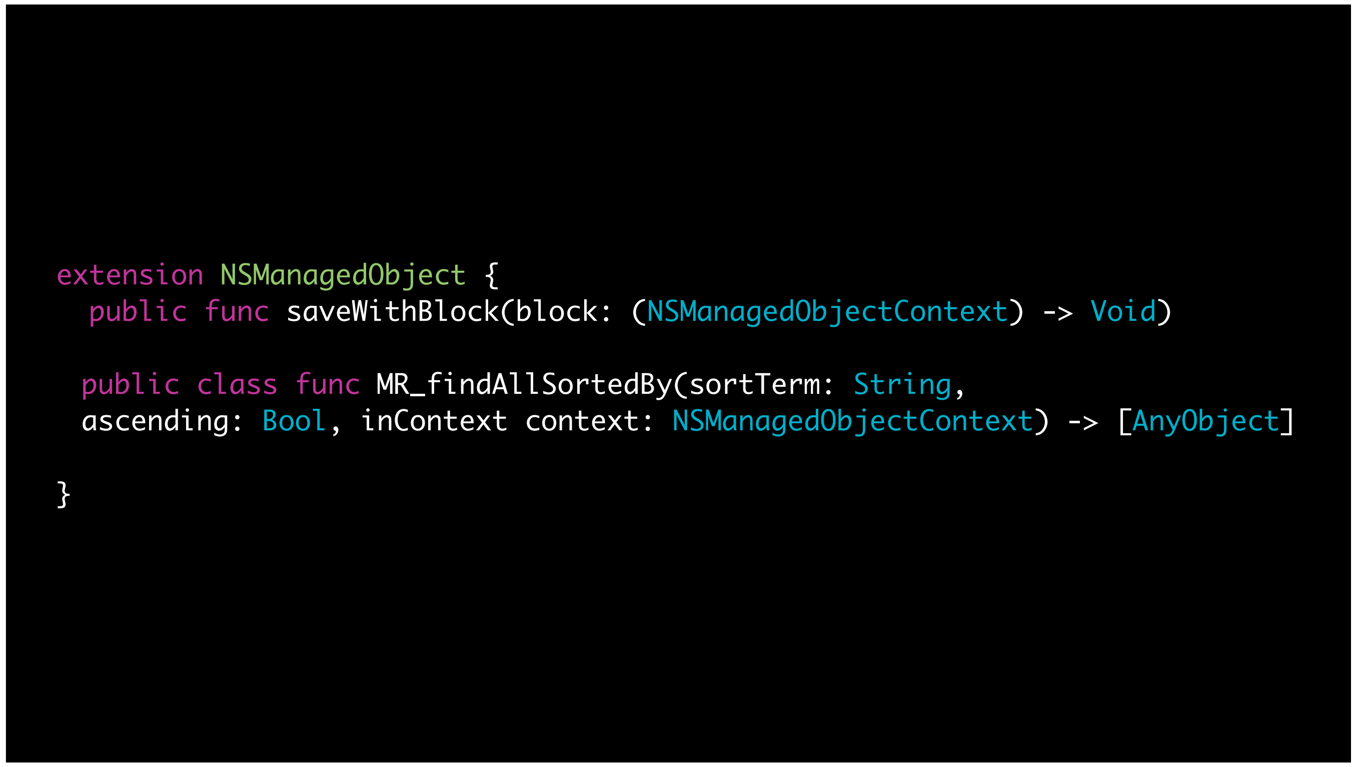


```
extension NSManagedObject {
  public func saveWithBlock(block: (NSManagedObjectContext!) -> Void)

  public class func MR_findAllSortedBy(sortTerm: String!,
  ascending: Bool, inContext context: NSManagedObjectContext!) -> [AnyObject]!

}
```

But, one of the problems with the first attempts at interoperability was all the crash operators being placed by default.

```
NS_ASSUME_NONNULL_BEGIN

NS_ASSUME_NONNULL_END
```

```
NS_ASSUME_NONNULL_BEGIN
@interface MagicalRecord (Setup)

+ (MagicalRecordStack *) setupSQLiteStack;
+ (MagicalRecordStack *)
setupSQLiteStackWithStoreAtURL:(NSURL *)url;
+ (MagicalRecordStack *)
setupSQLiteStackWithStoreNamed:(NSString
*)storeName;

- (instancetype) findWithPredicate:
(NSPredicate * __nullable)searchTerm;

@end
NS_ASSUME_NONNULL_END
```



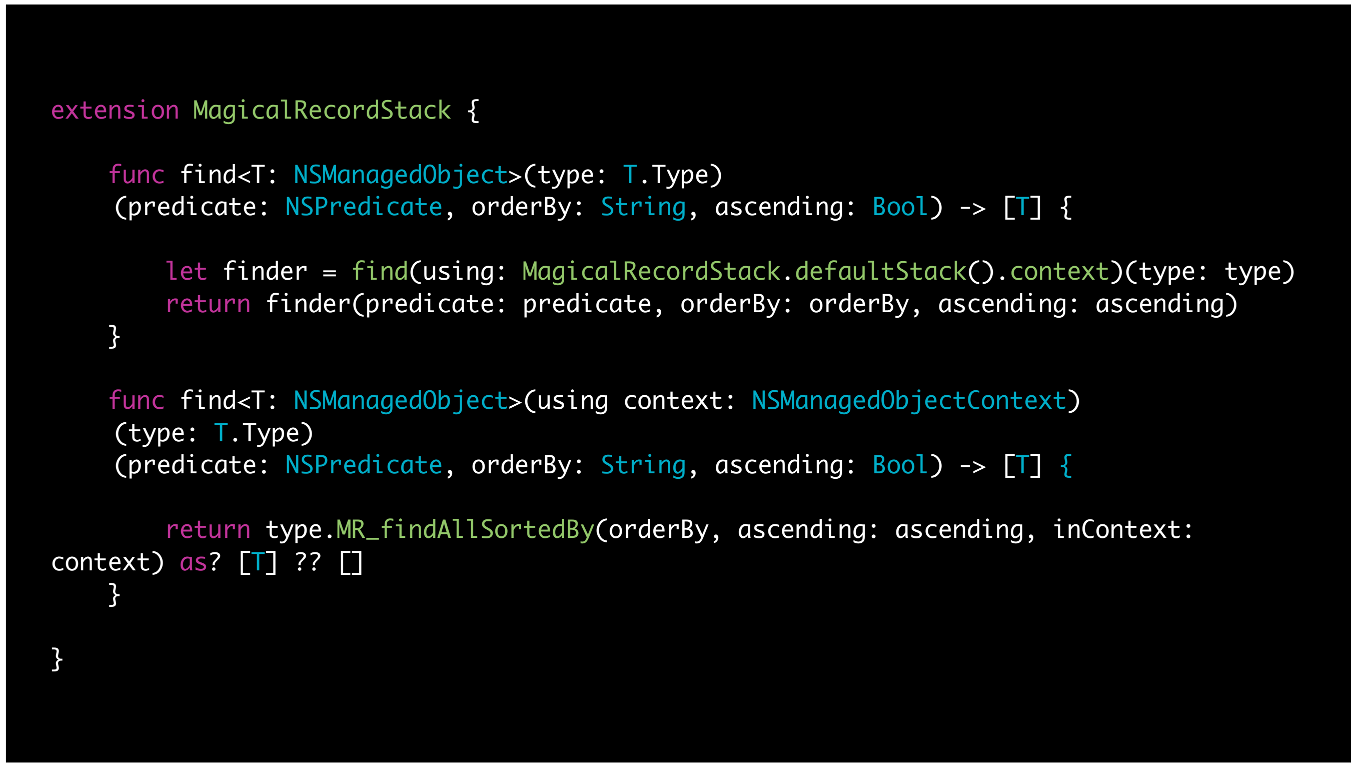
```
extension NSManagedObject {
  public func saveWithBlock(block: (NSManagedObjectContext) -> Void)

 public class func MR_findAllSortedBy(sortTerm: String,
 ascending: Bool, inContext context: NSManagedObjectContext) -> [AnyObject]

}
```

Now, we don’t have those crash operators in our generated swift compatibility headers.

extensions

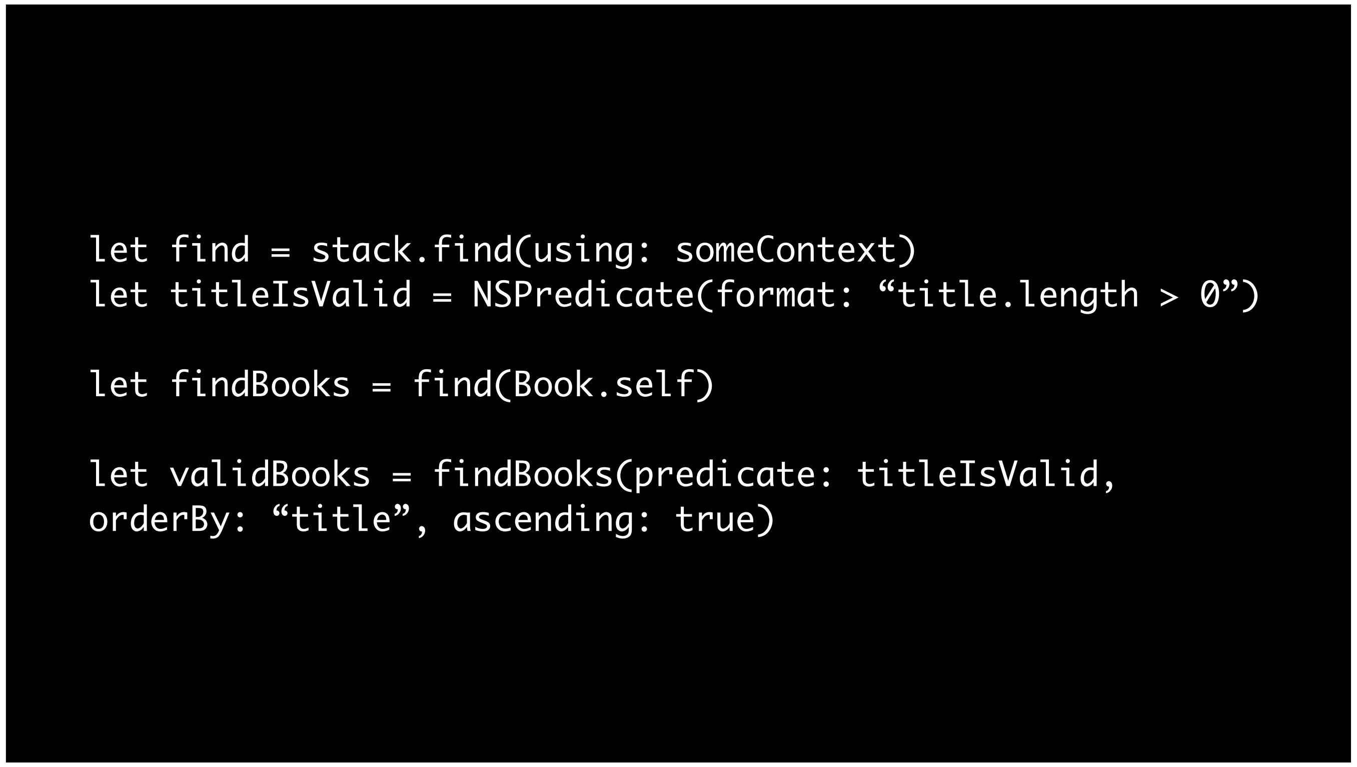
Being pragmatic, I decided that it’s still important to support the Objective C APIs. At the same time, the Swift compatibility layer with ObjC means I can write extensions in Swift only and not expose them to Objective C, and still have the Swift layer use code that is proven to still work, and work well for the key/value storage paradigm

## ~~Category Prefixes~~



```
extension MagicalRecordStack {

    func find<T: NSManagedObject>(type: T.Type)
    (predicate: NSPredicate, orderBy: String, ascending: Bool) -> [T] {

        let finder = find(using: MagicalRecordStack.defaultStack().context)(type: type)
        return finder(predicate: predicate, orderBy: orderBy, ascending: ascending)
    }

    func find<T: NSManagedObject>(using context: NSManagedObjectContext)
    (type: T.Type)
    (predicate: NSPredicate, orderBy: String, ascending: Bool) -> [T] {

        return type.MR_findAllSortedBy(orderBy, ascending: ascending, inContext:
context) as? [T] ?? []
    }

}
```

Curried functions for partially applying a method



```
let find = stack.find(using: someContext)
let titleIsValid = NSPredicate(format: “title.length > 0”)

let findBooks = find(Book.self)

let validBooks = findBooks(predicate: titleIsValid,
orderBy: “title”, ascending: true)
```

Curried functions let you have functions that return functions with simple syntax.

```
let originalObject: NSManagedObject = …

stack.save { usingContext in
  let localObject = other.inContext(usingContext)

  localObject.firstName = “Saul”
  localObject.lastName = “Mora”
}
```

And since MagicalRecord isn’t going to be a rewrite, some of the previous compatibility and call back conventions still work. Mogenerator still works, as does the importing. This is because Swift and ObjC are running on the same runtime, albeit with some optimizations for the Swift language

http://www.raywenderlich.com/store/core-data-by-tutorials

Classes
Protocols
Generics
Playgrounds
Swift Apprentice
iOS Apprentice
iOS 9 by Tutorials
watchOS 2 by Tutorials
Core Data by Tutorials
iOS Animations by Tutorials
2D iOS & tvOS Games by Tutorials

goto; conference

Please

Remember to rate this session

Thank you!

Let us know what you think

Join the conversation #gotocph