Initializing the Core Data Stack

The Core Data stack is a collection of framework objects that are accessed as part of the initialization of Core Data and that mediate 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 four primary objects: the managed object context (`NSManagedObjectContext`), the persistent store coordinator (`NSPersistentStoreCoordinator`), the managed object model (`NSManagedObjectModel`), and the persistent container (`NSPersistentContainer`).

You initialize the Core Data stack prior to accessing your application data. The initialization of the stack prepares Core Data for data requests and the creation of data. Here's an example of how to create that Core Data stack.

Listing 3-1 Example Core Data stack creation

OBJECTIVE-C

```
@interface MyDataController : NSObject

@property (strong, nonatomic, readonly) NSPersistentContainer *persistentContainer;

- (id)initWithCompletionBlock:(CallbackBlock)callback;

@end

@implementation MyDataController

- (id)init
{
    self = [super init];
    if (!self) return nil;

    self.persistentContainer = [[NSPersistentContainer alloc]
  initWithName:@"DataModel"];
    [self.persistentContainer
  loadPersistentStoresWithCompletionHandler:^(NSPersistentStoreDescription
  *description, NSError *error) {
        if (error != nil) {
            NSLog(@"Failed to load Core Data stack: %@", error);
            abort();
        }
        callback();
    }];

    return self;
}
```

SWIFT

```
import UIKit
import CoreData
class DataController: NSObject {
    var managedObjectContext: NSManagedObjectContext
    init(completionClosure: @escaping () -> ()) {
        persistentContainer = NSPersistentContainer(name: "DataModel")
        persistentContainer.loadPersistentStores() { (description, error) in
            if let error = error {
                fatalError("Failed to load Core Data stack: \(error)")
            }
            completionClosure()
        }
    }
}
```

On This Page

This example creates a controller object that represents the persistence layer of the application. The controller is initialized with a default `init` call. As part of that `init` call, the `initializeCoreData` method is called, and it then proceeds to create the Core Data stack.

NSPersistentContainer

Starting in iOS 10 and macOS 10.12, the `NSPersistentContainer` handles the creation of the Core Data stack and offers access to the `NSManagedObjectContext` as well as a number of convenience methods. Prior to iOS 10 and macOS 10.12, the creation of the Core Data stack was more involved.

OBJECTIVE-C

```
- (id)initWithCompletionBlock:(CallbackBlock)callback;
{
    self = [super init];
    if (!self) return nil;

    //This resource is the same name as your xcdatamodeld contained in your project
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"Workspace"
  withExtension:@"momd"];
    NSAssert(modelURL, @"Failed to locate momd bundle in application");
    // The managed object model for the application. It is a fatal error for the
  application not to be able to find and load its model.
    NSManagedObjectModel *mom = [[NSManagedObjectModel alloc]
  initWithContentsOfURL:modelURL];
    NSAssert(mom, @"Failed to initialize mom from URL: %@", modelURL);

    NSPersistentStoreCoordinator *coordinator = [[NSPersistentStoreCoordinator
  alloc] initWithManagedObjectModel:mom];

    NSManagedObjectContext *moc = [[NSManagedObjectContext alloc]
  initWithConcurrencyType:NSMainQueueConcurrencyType];
    [moc setPersistentStoreCoordinator:coordinator];
    [self setManagedObjectContext:moc];
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0),
  ^{
        NSPersistentStoreCoordinator *psc = [[self managedObjectContext]
  persistentStoreCoordinator];
        NSFileManager *fileManager = [NSFileManager defaultManager];
        NSURL *documentsURL = [[fileManager
  URLsForDirectory:NSDocumentationDirectory inDomains:NSUserDomainMask] lastObject];
        // The directory the application uses to store the Core Data store file.
  This code uses a file named "DataModel.sqlite" in the application's documents
  directory.
        NSURL *storeURL = [documentsURL
  URLByAppendingPathComponent:@"DataModel.sqlite"];

        NSError *error = nil;
        NSPersistentStore *store = [psc addPersistentStoreWithType:NSSQLiteStoreType
  configuration:nil URL:storeURL options:nil error:&error];
        if (!store) {
            NSLog(@"Failed to initalize persistent store: %@\n%@", [error
  localizedDescription], [error userInfo]);
            abort();
            //A more user facing error message may be appropriate here rather than
  just a console log and an abort
        }
        if (!callback) {
            //If there is no callback block we can safely return
            return;
        }
```

```
        //The callback block is expected to complete the User Interface and therefore should be presented back on the main queue so that the user interface does not need to be concerned with which queue this call is coming from.

            callback();
        });
    });
    return self;
}
```

```swift
init(completionClosure: @escaping () -> ()) {
    //This resource is the same name as your xcdatamodeld contained in your project
    guard let modelURL = Bundle.main.url(forResource: "DataModel", withExtension:"momd") else {
        fatalError("Error loading model from bundle")
    }
    // The managed object model for the application. It is a fatal error for the application not to be able to find and load its model.
    guard let mom = NSManagedObjectModel(contentsOf: modelURL) else {
        fatalError("Error initializing mom from: \(modelURL)")
    }

    let psc = NSPersistentStoreCoordinator(managedObjectModel: mom)

    managedObjectContext = NSManagedObjectContext(concurrencyType: NSManagedObjectContextConcurrencyType.mainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = psc

    let queue = DispatchQueue.global(qos: DispatchQoS.QoSClass.background)
    queue.async {
        guard let docURL = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask).last else {
            fatalError("Unable to resolve document directory")
        }
        let storeURL = docURL.appendingPathComponent("DataModel.sqlite")
        do {
            try psc.addPersistentStore(ofType: NSSQLiteStoreType, configurationName: nil, at: storeURL, options: nil)
            //The callback block is expected to complete the User Interface and therefore should be presented back on the main queue so that the user interface does not need to be concerned with which queue this call is coming from.
            DispatchQueue.main.sync(execute: completionClosure)
        } catch {
            fatalError("Error migrating store: \(error)")
        }
    }
}
```

In addition to simplifying the construction of the Core Data stack, the `NSPersistentContainer` also has a number of convenience methods that assist the developer when working with multithreaded applications.

NSManagedObjectModel

The `NSManagedObjectModel` instance describes the data that is going to be accessed by the Core Data stack. During the creation of the Core Data stack, the `NSManagedObjectModel` is loaded into memory as the first step in the creation of the stack. The example code above resolves an `NSURL` from the main application bundle using a known filename (in this example `DataModel.momd`) for the `NSManagedObjectModel`. After the `NSManagedObjectModel` object is initialized, the `NSPersistentStoreCoordinator` object is constructed.

NSPersistentStoreCoordinator

The `NSPersistentStoreCoordinator` sits in the middle of the Core Data stack. The coordinator is responsible for realizing inst
the model, and i
store can be on disk or in memory. Depending on the structure of the application, it is possible, although uncommon, to have more than one persistent store being coordinated by the `NSPersistentStoreCoordinator`.

Whereas the `NSManagedObjectModel` defines the structure of the data, the `NSPersistentStoreCoordinator` realizes objects from the data in the persistent store and passes those objects off to the requesting `NSManagedObjectContext`. The `NSPersistentStoreCoordinator` also verifies that the data is in a consistent state that matches the definitions in the `NSManagedObjectModel`.

The call to add the `NSPersistentStore` to the `NSPersistentStoreCoordinator` is performed asynchronously. A few situations can cause this call to block the calling thread (for example, integration with iCloud and Migrations). Therefore, it is better to execute this call asynchronously to avoid blocking the user interface queue.

NSManagedObjectContext

The managed object context (`NSManagedObjectContext`) is the object that your application will interact with the most, and therefore it is the one that is exposed to the rest of your application. Think of the managed object context as an intelligent scratch pad. 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 however you like. 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.

If you choose to save the changes you have made, the context ensures that your objects are in a valid state. If they are, 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.

Without Core Data, you have to write methods to support archiving and unarchiving of data, to keep track of model objects, and to interact with an undo manager to support undo. In the Core Data framework, most of this functionality is provided for you automatically, primarily through the managed object context.