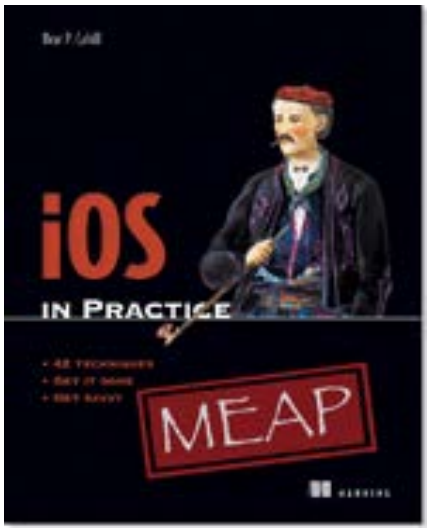
MANNING PUBLICATIONS



iOS in Practice
By Bear Cahill

The iPhone, iPod, and iPad are great music playing devices and allow you to create playlists. Creating playlists is useful, but being able to play them is necessary to really make it worthwhile. So using the iPod access to select tracks, CoreData to store the selections, and the music player to play the music makes for a usable app. To start, you need to design a database within Xcode and have the related code generated for you. In this article based on chapter 7 of iOS in Practice, author Bear Cahill shows you how to set up the project to support a table view navigation.

Creating a Table View Project

All apps deal with some form of data and many need to store that data in a meaningful way. Often, the solution is a database including tables, rows, and relationships between them. A common way to display the data to the user in a meaningful way is with rows in a table.

Technique 1: Creating a project with table navigation

Data-driven apps typically need an interface to allow the user to select items and/or details for a given item. While the data is stored in a database with CoreData handling the storing and retrieving, what the user sees in the UI is often a tableview representing the data. And often with the user of tableviews, when the user taps on a given row, they expect to see details slide in from the right, which is the functionality of a navigation controller.

Problem

We need to create a project with a table view and a navigation controller.

Solution

We will create a Master-Detail application project in Xcode using Storyboard and CoreData. Then, we'll set up the UI to have a tableview embedded in a navigation controller.

Discussion

Open Xcode and select to create a new project. From the pop window that displays, select **Master-Detail Application** and click Next (see figure 1).

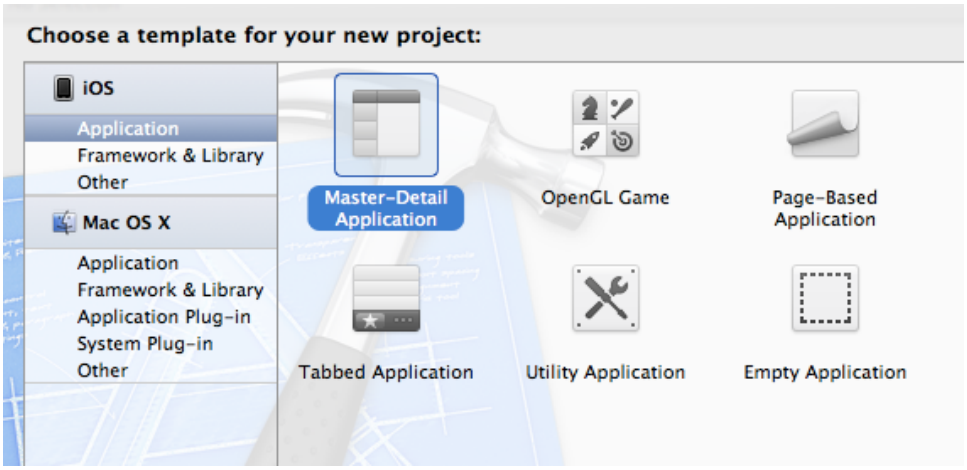


Figure 1 Creating a new Master-Detail Application Xcode Project

Continuing on with the project creation, set the product name (PlayMyLists) and company identifier (see figure 2). Notice that we'll use Storyboard and CoreData for this project. One thing to note is that Storyboard requires iOS version 5, so if you want to support past versions of iOS, you should use Storyboard.

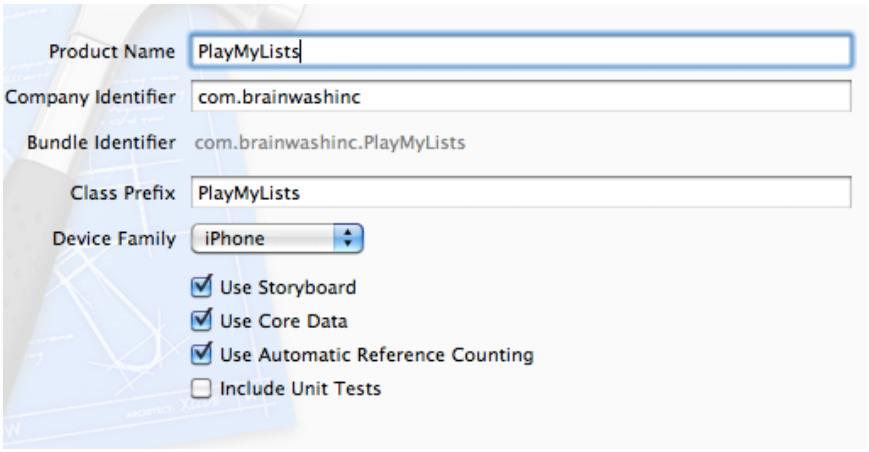


Figure 2 Project options including Storyboard and CoreData

Continue with the project creation by selecting the location and click **Create** to create the project and view the **Summary** (see figure 3).

Figure 3 Newly created project summary data

Open the Storyboard file using the Navigator. As you can see, the base view controller is created as a navigation controller with a root controller of a table view controller which leads to a view controller for displaying data details (see figure 4). This helps to confirm our assumption that this hierarchy style data display and drilling down into details is a commonly used design.

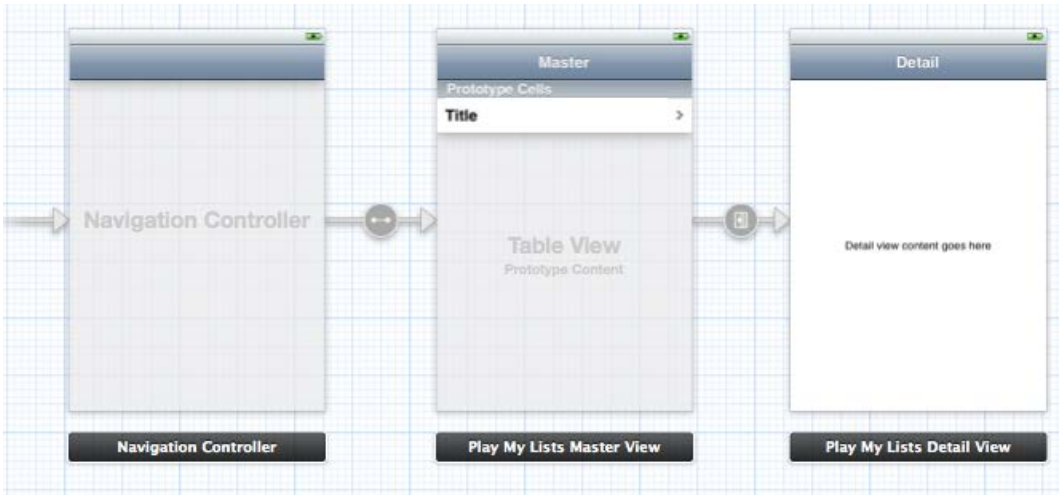


Figure 4 Storyboard of Master-Detail Application user interface

At this point, Xcode has handled most of the heavy (and light) lifting for us. It included the CoreData framework to our project and it created our app delegate, master view controller, detail view controller classes, as well as the storyboard to use them. On top of that, our master view controller is a table view controller. Finally, both the app delegate and our master view controller have a lot of code relating to common table view controller and CoreData generated for us.

If you run this app in the simulator, you can create new database data using the add **(+)** button, delete them using the **Edit** butto,n and view details for the items by tapping on the listed rows (see figure 5).

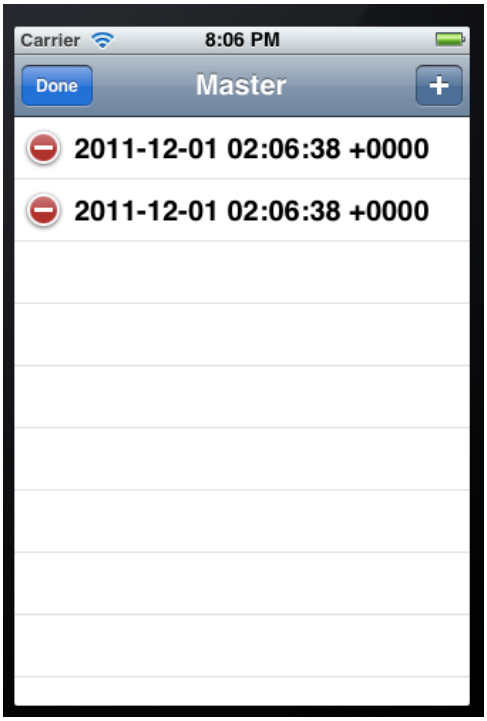


Figure 5 Executing the template code in the Simulator

So we're set up on the UI side, but what about the data side? We need to configure our project to be ready to store and access data using CoreData.

Technique 2: Defining entities in CoreData

Our entities will be fairly simple, but more complex entities are possible. We will have two entities for our playlists: the playlist and its member tracks. As you may have guessed, playlists will contain tracks so we'll need to set up the relationship between playlists and tracks later. In this technique, we'll focus on just the entities: creating and adding attributes.

Built into Xcode is the interface to develop database entities. An entity maps to a table and the attributes are the fields in the table. We'll use this interface to enter all of the data we'll need for our database.

Problem

We need to create two CoreData entities for our app, AppPlayList and PlayListTrack, to map to our database tables. The playlist needs to have a name and the track needs a persistent id which we'll later relate to the persistent id in the iPod.

Solution

Using the data model editor, we will define our entities and their attributes.

Discussion

Select the PlayMyLists.xcdatamodel item in the **Groups & Files** area on the left side in Xcode. Notice the Event template entity is listed in the **Entities** area. Select and delete this item so we can start fresh. Now that our list is empty, click the **Add Entity** button at the bottom of Xcode. A new entity is created and now you need to enter the name: AppPlayList.

> **ENTITY NAMING WARNING**
>
> Be careful with your naming choices for your entities. If they duplicate existing class names within iOS frameworks, you may wind up with very confusing errors at compile time.

Next, add an attribute to the entity. Click the **+** button at the bottom left of the **Attributes** area and type the name *name* then set its type to String (see figure 6).

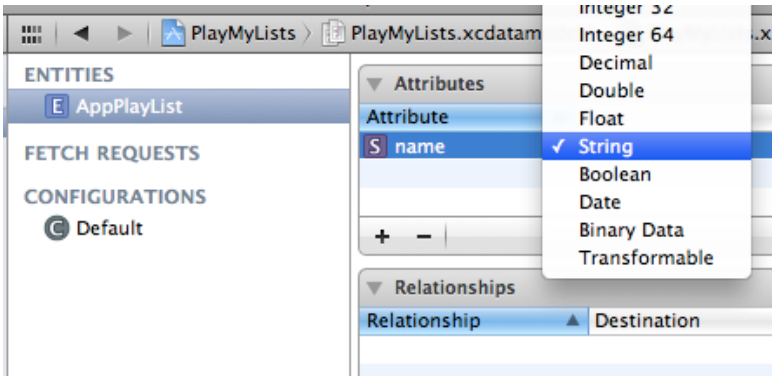


Figure 6 AppPlayList entity with the name attribute of type String

Now create another new Entity with the button at the bottom of the frame. Set its name to PlayListTrack. Add a new attribute and name it *persistentID* and make it of type String (see figure 7).

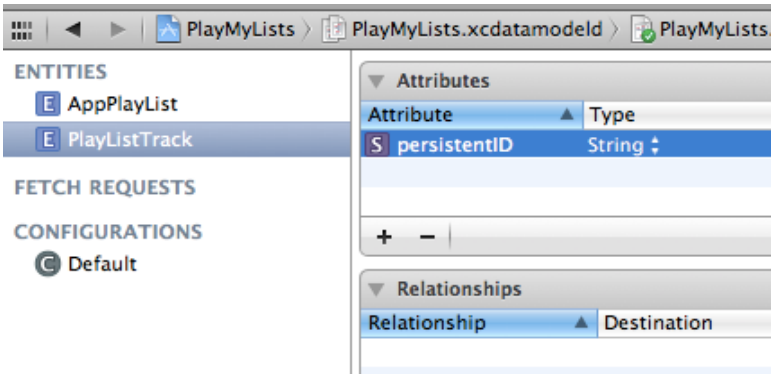


Figure 7 PlayListTrack entity with persistentID attribute of type String

We now have our two entities. However, they are not related. How does the app know which tracks goes with which playlists? We need to define the relationship between these two entities. Let's do that now.

Technique 3: Creating relationships in CoreData

Like most cases, our database needs relationships. Playlists need to have access to multiple tracks and tracks should know which playlist they belong to. So the relationship from the playlists to the tracks is a *to-many* relationship: one playlist can point to many tracks.

The model editor lets us build the relationships including to-many relationships. Also, the relationship can be set to handle object deletion in different ways: whether to delete the object or the objects it's related to in the database.

Problem

We need to define a to-many relationship between the playlists and tracks.

Solution

Using the CoreModel editor, we can define relationships between the two entities. Also, we can specify the relationship from the AppPlayList entity to the PlayListTrack is a to-many relationship. We'll also define the relationship from the PlayListTrack as the inverse of the AppPlayList's relationship.

Discussion

Select the same xcdatamodel as before from the left pane. Select the AppPlayList entity in the definition. Click and hold the **Add Attribute** button for the drop-down menu to display and select **Add Relationship**.

A new relationship will be defined for the AppPlayList entity. Name the new relationship *tracks*. To the right of it, select PlayListTrack as the **Destination**. Do not specify an inverse.

Bring up the **Utilities** view on the right for the new relationship and make sure the **Data Model Inspector** is selected (at the top of **Utilities**). Check the **To-Many Relationship** in the Plural area (see figure 8). Also notice we set Cascade as the **Delete Rule**. This means that if the playlist is deleted, so is the track data. We don't want tracks orphaned in the database.

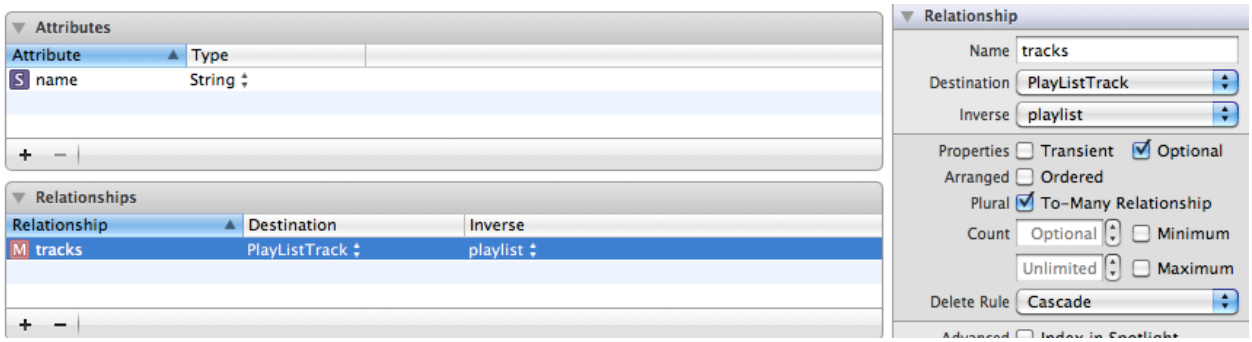


Figure 8 Creating a to-many relationship between AppPlayList and PlayListTrack

Now select the PlayListTrack entity and create a new relationship from the drop-down menu as you did before. Name it *playlist* since it points to the given track's playlist. Set its Destination to be AppPlayList. Set its inverse to *tracks*, which is the relationship you defined for AppPlayList (see figure 9). This means it's the other end of that same relationship.

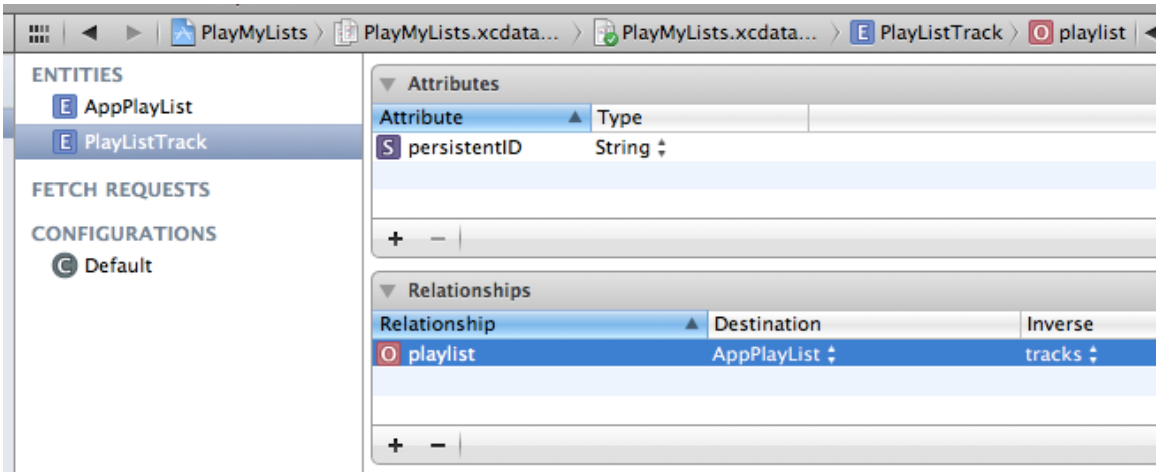


Figure 9 Creating the inverse relationship from PlayListTrack to AppPlayList

Had we not set this as the inverse (or even created it at all), the playlist would have had a relationship to the tracks. However, the track wouldn't know what playlist it belongs to. Depending on your app, this aspect may be useful or not. We've created our entities and the relationship between the two. To get a high-level view of the entities, select them both on the right (see figure 10).

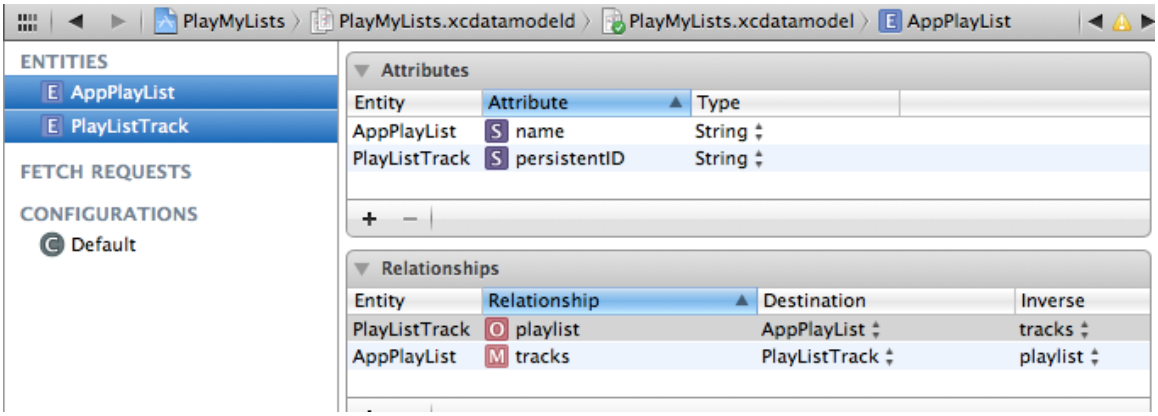


Figure 10 Viewing all entities defined and their attributes and relationships

To see a visual representation of the database defined by our entities, click the Table/Graph toggle on the bottom right (see figure 11).

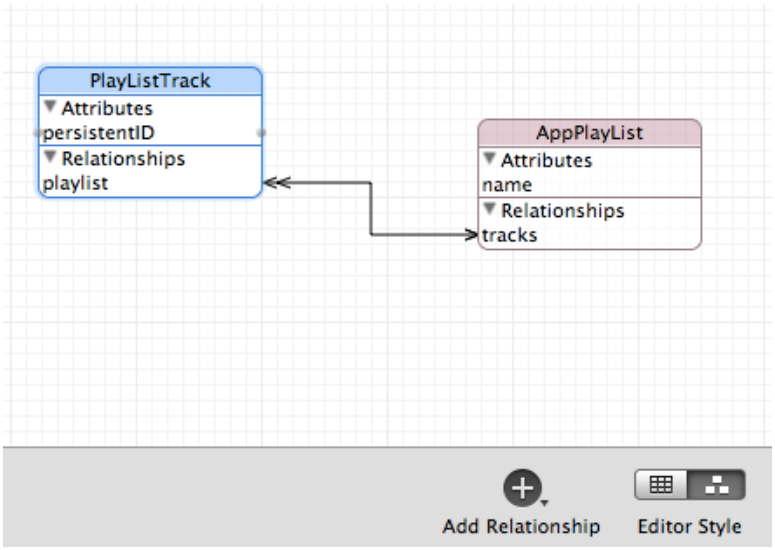


Figure 11 Graph representation of the database entities and the relationship

At this point, your entities are defined and usable. The iOS framework provides the means to access your objects as `NSManageableObject` instances and the attributes through key-value coding. Let's look at how we can do this with the existing code in our project.

Technique 4: Inserting and deleting CoreData objects

We've changed our database definition so the app is no longer valid—it still uses the previous defined entity and attribute. However, with only a few changes we can see how CoreData allows us to access the data.

Since the default project already handled some of the CoreData functions including creating new objects, listing them in the table view and deleting them, we can reuse some of that code. However, we need to update the code to reflect the changes we made to the Event entity.

Problem

We need to access our AppPlayList database entity from the code and display its attribute.

Solution

We will change the references in the code to use our definition changes.

Discussion

Because the default CoreData code in our project uses key-value coding, we don't need to change any of the class names. Also, because the default entity only had one attribute and so does our AppPlayList entity, we only need to change that one attribute's references.

The app needs to know what entity it is dealing with (creating, accessing, deleting, and so on). In the method that creates and/or returns the object to fetch data, we need to tell it to use the new entity name.

Find the method named `fetchedResultsController`. It has one line that specifies what entity it is dealing with. Find that line and change Event to AppPlayList.

```
NSEntityDescription *entity = [NSEntityDescription
    entityForName:@"AppPlayList"
    inManagedObjectContext:self.managedObjectContext];
```

That's the only place the entity name is used. Now our `NSFetchedResultsController` instance will access that entity. However, we need to tell it the name of the attribute also.

The `fetchedResultsController` instance needs to know how to sort the fetched objects from the database. This is done via the NSSortDescriptor instance created for the `fetchedResultsController`. In the same method, `fetchedResultsController`, set the value from `"timeStamp"` to `"name"` where the sort descriptor is created.

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
                          initWithKey:@"name" ascending:NO];
```

The previous entity had an attribute named `"timestamp"`. If we change all references in the code to our new attribute, `"name"`, the code will access the data correctly.

In the method that configures the table cell, we specify the attribute to display in the table. Change that from `"timeStamp"` to `"name"` (see listing 1).

Listing 1 Specify the name Attribute to be Displayed in the Table Cell

```
- (void)configureCell:(UITableViewCell *)cell
    atIndexPath:(NSIndexPath *)indexPath {

    NSManagedObject *managedObject =
        [self.fetchedResultsController
            objectAtIndexPath:indexPath];
    cell.textLabel.text = [[managedObject
                            valueForKey:@"name"] description];
}
```

Now we're accessing the right attribute, but the app also needs to set the right attribute. Make the same change from `"timeStamp"` to `"name"` in the `insertNewObject` method. Default the value of the name to `"New List"` which is more meaningful in our app.

```
[newManagedObject setValue:@"New List" forKey:@"name"];
```

Now you can run the app and create database items of our entity: AppPlayList. Of course, you can delete them too. However, these playlists aren't really anything yet. They're just entries in the database—all with the same name value and no track (see figure 12). Be sure to delete the old version of the app first to avoid issues with the existing database.

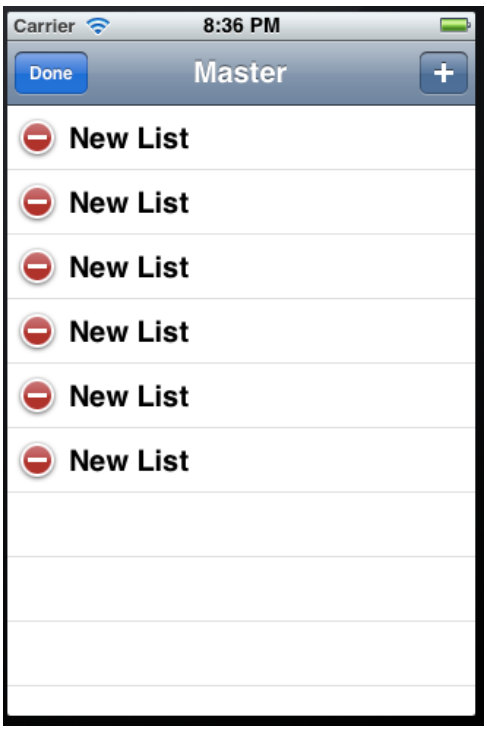


Figure 12 PlayMyLists listing newly created AppPlayList entities

To add tracks to the playlist, we can use the same key-value coding mechanism or we can generate code for our entities and access it all via member items. Let's look at how to generate Objective-C code from our CoreData entities.

Technique 5: Creating classes for CoreData entities

Creating classes based on CoreData entities can be very useful in accessing the data. However, you probably don't want to edit that code after you create it. If you need to change your entity definition later and regenerate the code, it will blow away your changes. Create helper classes or child classes instead.

Helper classes can be classes that inherit from your generated CoreData classes or completely separate. These are classes that can possibly filter our objects, sort them, convert them for transmission or any other functionality you might need.

Problem

We need to generate code based on our CoreData entity definitions.

Solution

We will use the Xcode new file mechanism to generate the code into our project.

Discussion

Open the data model again and select the entities (or just one). You can right-click on the Entity name and get helpful documentation relating to CoreData operations here.

To create the subclass, select the entity or entities you'd like to create source code for in the Entity list. From the Editor menu, select the **Create NSManagedObject Subclass...** option (see figure 13).

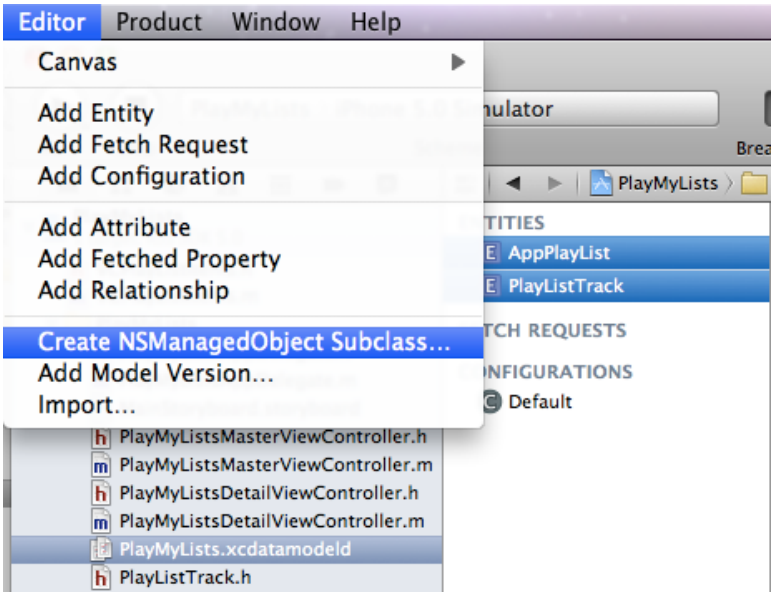


Figure 13 Selecting entities to generate Objective-C code

On the next form, select the location and target (see figure 14).

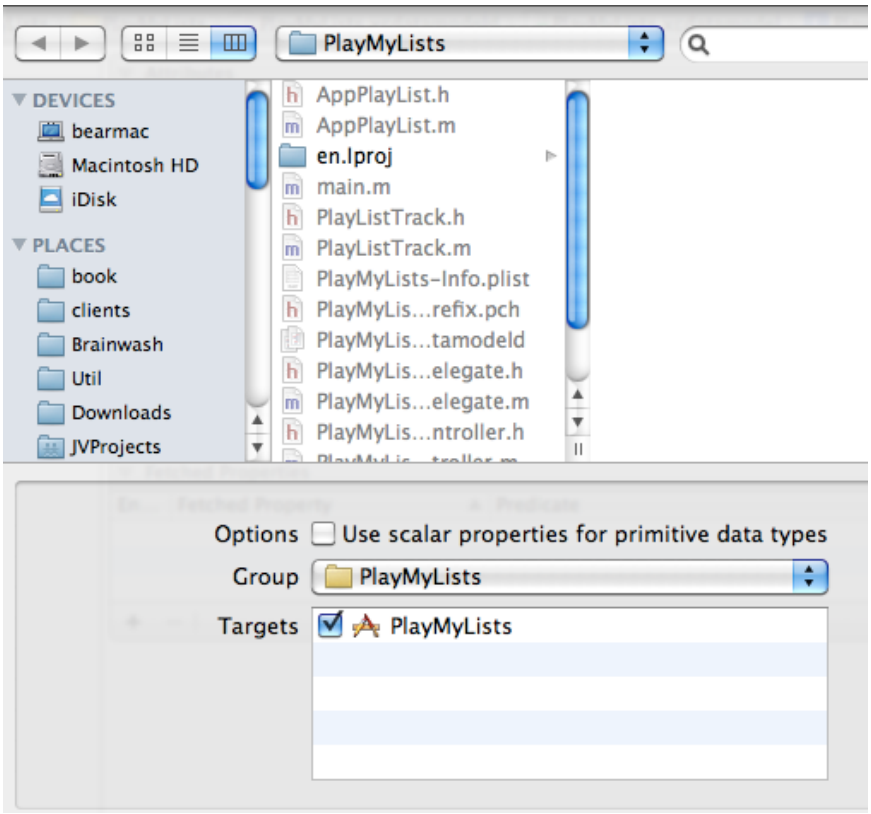


Figure 14 Specifying the directory, group, and target for the new NSManagedClass files

There should be two to four new classes generated based on the entities selected: a header and implementation file for each. The files will have the same names as the entities you created. Again, this can cause problems if you

named your entities something that may already exist somewhere. I created an entity named Message once, and it took a while to figure out that was a problem with the generated code.

By looking at the implementation files, you'll see there's not much to the classes. The header files are more interesting because they tell you the methods you have access to (see listings 2 and 3). Notice how the attributes and relationships both translate into items.

Listing 2 Header file for AppPlayList with name and tracks

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

@interface AppPlayList : NSManagedObject
{
}

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSSet* tracks;            #1

@end

@interface AppPlayList (CoreDataGeneratedAccessors)
- (void)addTracksObject:(NSManagedObject *)value;       #2
- (void)removeTracksObject:(NSManagedObject *)value;
- (void)addTracks:(NSSet *)value;
- (void)removeTracks:(NSSet *)value;

@end
```

#1 NSSet for multiples
#2 Relationship accessors

CoreData objects use NSSet instances for the to-many relationships. NSSet is similar to NSArray but isn't ordered. In our case accessing tracks returns the set of tracks for the given playlist as one or more PlayListTrack instances. There are also generated methods to add and remove single and multiple tracks.

Listing 3 Header file for PlayListTrack with persistentID and playList

```
#import <CoreData/CoreData.h>
@class AppPlayList;

@interface PlayListTrack : NSManagedObject
{
}

@property (nonatomic, retain) NSString * persistentID;
@property (nonatomic, retain) AppPlayList * playList;    #1

@end
```

#1 Relationship is member

Since the playlist has a to-many relationship with the tracks, the accessors need to support that. It uses NSSet to return the tracks and has methods to add additional tracks also.

So that's how you generate the code for CoreData entities. We can use these classes for data in the database and access the attributes as data.

Summary

We've now seen how to create a project using CoreData. We've also looked how to define entities, their attributes, relationships and even inserting and deleting instances of the data and the source code.

Here are some other Manning titles you might be interested in:

Creating iPhone Apps
Lou Franco

Quick & Easy iPhone Programming
Bintu Harwani

Objective-C Fundamentals
Christopher K. Fairbairn, Johannes Fahrenkrug, and Collin Ruffenach

Last updated: January 30, 2012