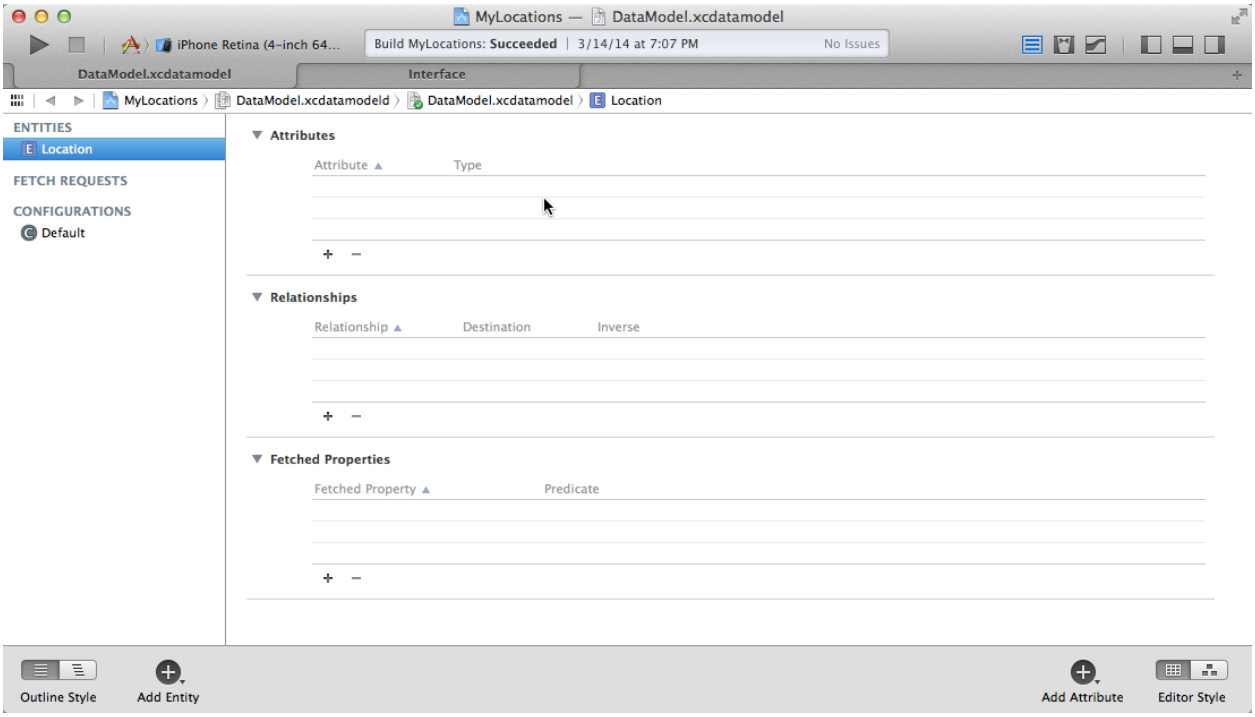
从零开始学iOS7开发系列3-我的地盘我做主-Cha21

说明:

本系列文章的原文及示例代码来自raywenderlich store中的iOS Apprentice 系列3教程，经过翻译和改编。

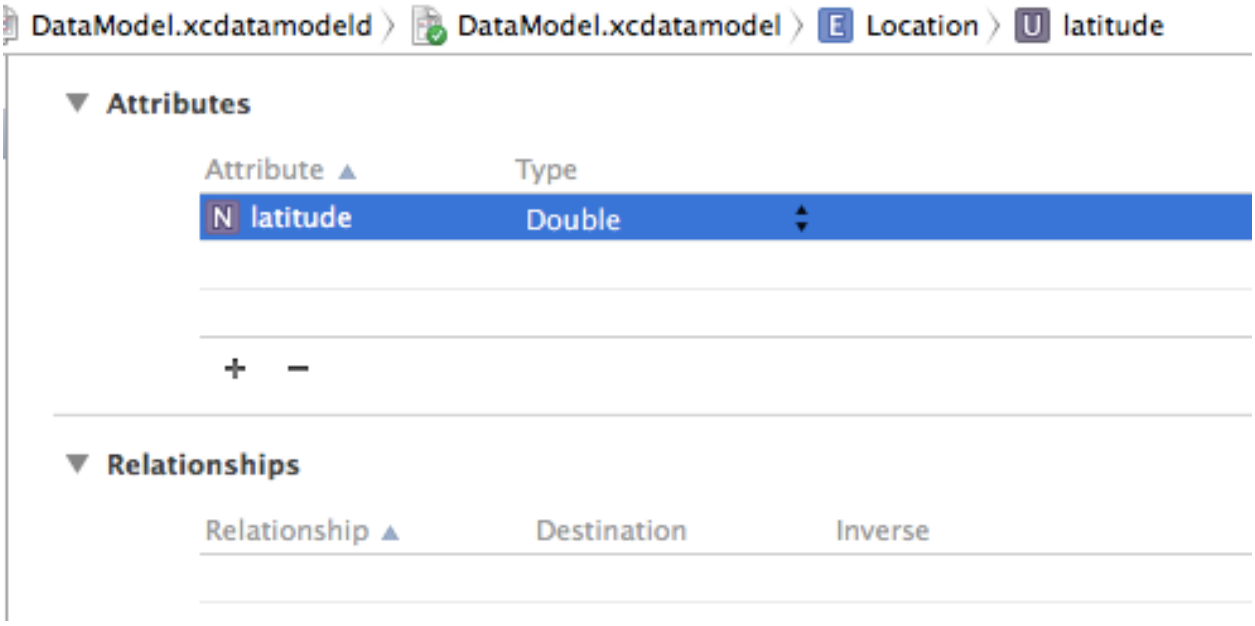
欢迎继续我们的学习。

在数据模型编辑器的底部点击Add Entity按钮。这样就会在ENTITIES这个heading标题的下面添加一个新的entity，将其命名为Location。（我们可以通过点击其名称或者从Data Model面板中更改entity的名称）。



在右侧可以看到有三个部分：Attributes,Relationships和Fetched Properties。Attributes部分其实就是entity的数据字段。这个应用只有一个entity，但通常情况下应用会有多个相互关联在一起的entity。通过Relationships和Fetched Properties，我们可以告诉Core Data对象之间的依赖关系。对当前这个应用，我们只需要使用Attributes 部分就可以了。

点击编辑器底部的Add Attribute按钮（或是Attributes部分下面的小加号），可以将新的attribute命名为latitude，将其Type设置为Double：



Attributes基本上和实例变量是相同的，因此他们有一个数据类型。之前我们看到latitude和longtitude的坐标也有数据类型double，因此这里选择的attributes也是如此。

好吧，这里用到了两个新名词，但是不要恐慌。Don't panic，你可以这样来看这两个术语：
entity= object(或者class)
attribute = variable

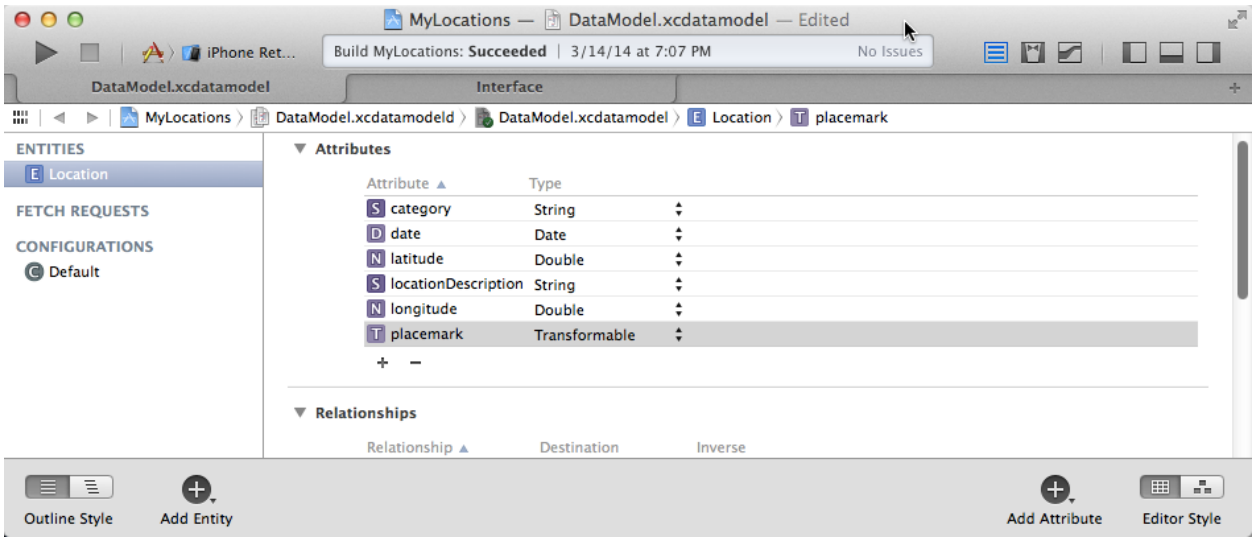
或许你会问，在Core Data中我们在哪里定义方法呢？答案是：我们不需要这么做。Core Data的唯一目的就是保存对象的数据部分。也就是一个entity所描述的内容，一个对象的数据，或者对象如何和其它对象关联在一起（使用Relationships和Fetched Properties）。

当然，我们后续仍然需要通过创建一个.h和.m文件来定义一个自己的Location类，就和之前所做的一样。因为它描述的是一个managed object，这个类将和数据模型中的Location entity关联在一起。不过即便如此它也仍然是一个常规的类，我们也可以在其中添加自己的方法。

在Location 这个entity中添加其它的attributes如下：

longitude,类型是Double
date，类型是Date
locationDescription，类型是String
category，类型是String
placemark,类型是Transformable

完成后的数据模型如下所示：



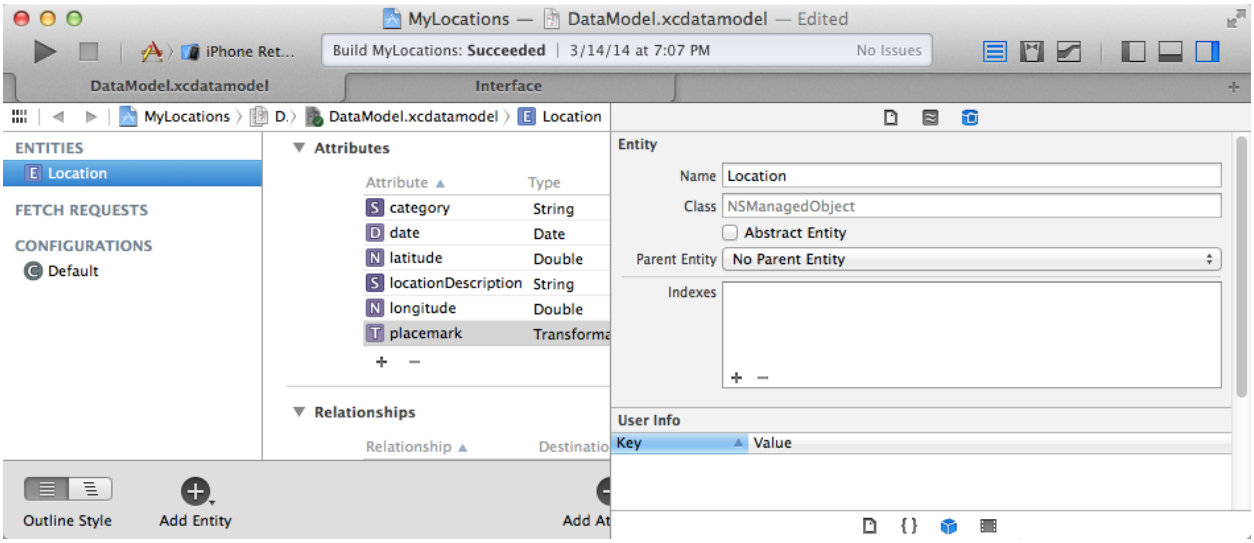
好吧，为什么要用locationDescription而不是简单的description呢？因为description是NSObject的一个方法名称。如果我们将某个attribute命名为description，就会导致该名称和方法名称的冲突。实际上Xcode会直接给你红色的error信息打消你的这一想法。

placemark这个attribute的类型是Transformable。Core Data只支持有限的几种数据类型，比如String, Integer 32和Date。 placemark是CLPlacemark类型的对象，因此不在Core Data所支持的数据类型之中。

幸运的是，Core Data提供了处理此类数据类型的解决方案。任何遵从NSCoding协议的类都可以保存在Transformable类型的attribute中，不需要进行任何额外的操作。恰好CLPlacemark就遵从NSCoding协议，因此我们可以直接将其保存在Core Data之中。

对于数据模型还有一个小小的事情要做，然后就大功告成了。

选中Location 这个entity，在inspector面板中切换到Data Model inspector。



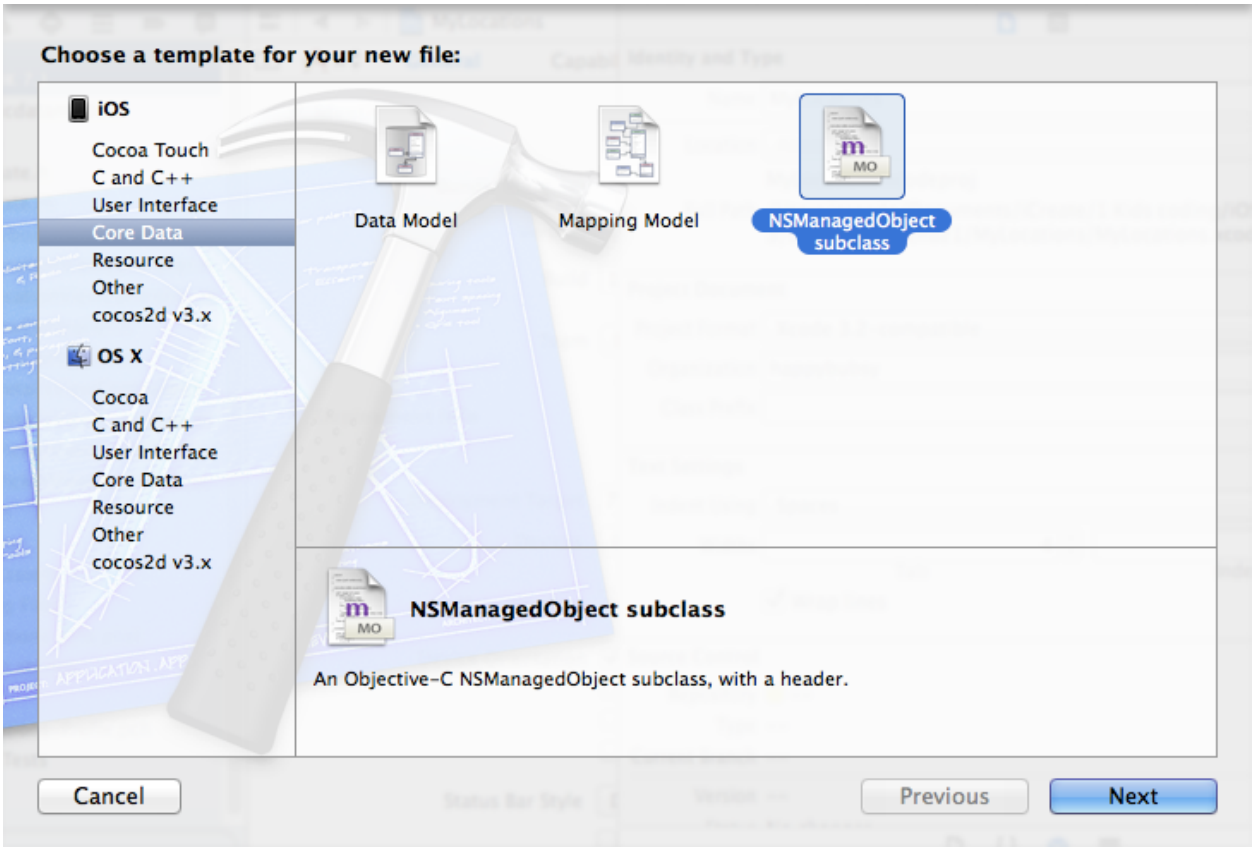
当前在Class字段的信息是NSManagedObject，这就意味着当我们从Core Data获取一个Location的entity时，会获得一个NSManagedObject类型的对象。NSManagedObject是所有由Core Data管理的对象的基类。通常来说iOS开发中对象继承自NSObject，但来自Core Data的类则继承自NSManagedObject。因为直接使用NSManagedObject有一些限制，我们考虑使用自定义的类。

将Class字段的值更改为Location，也就是我们即将创建的新类名称。

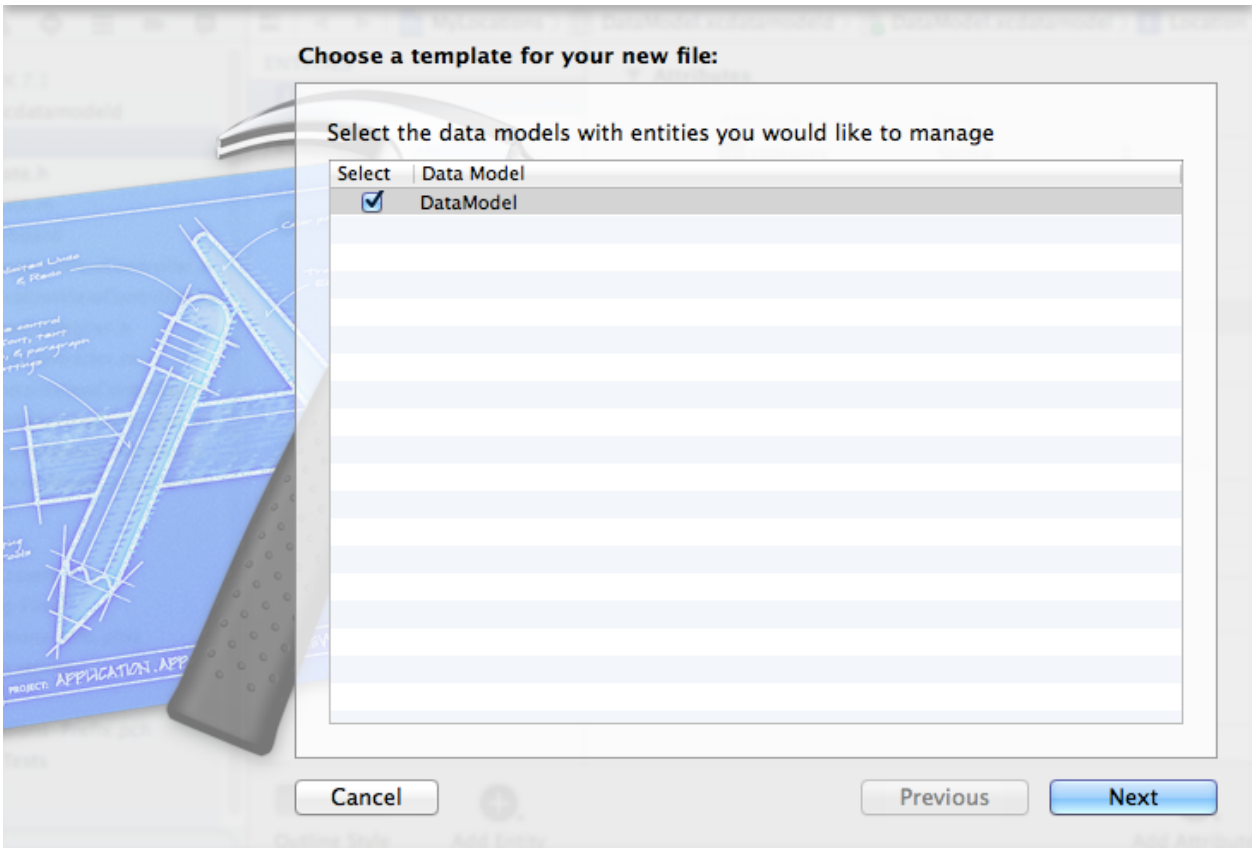
我们并非必须为entity创建自定义的类，但通过这种方式可以让Core Data更方便使用。当我们从data store中获取一个Location entity的时候，Core Data不会提供一个NSManagedObject，而是一个自定义类的实例。

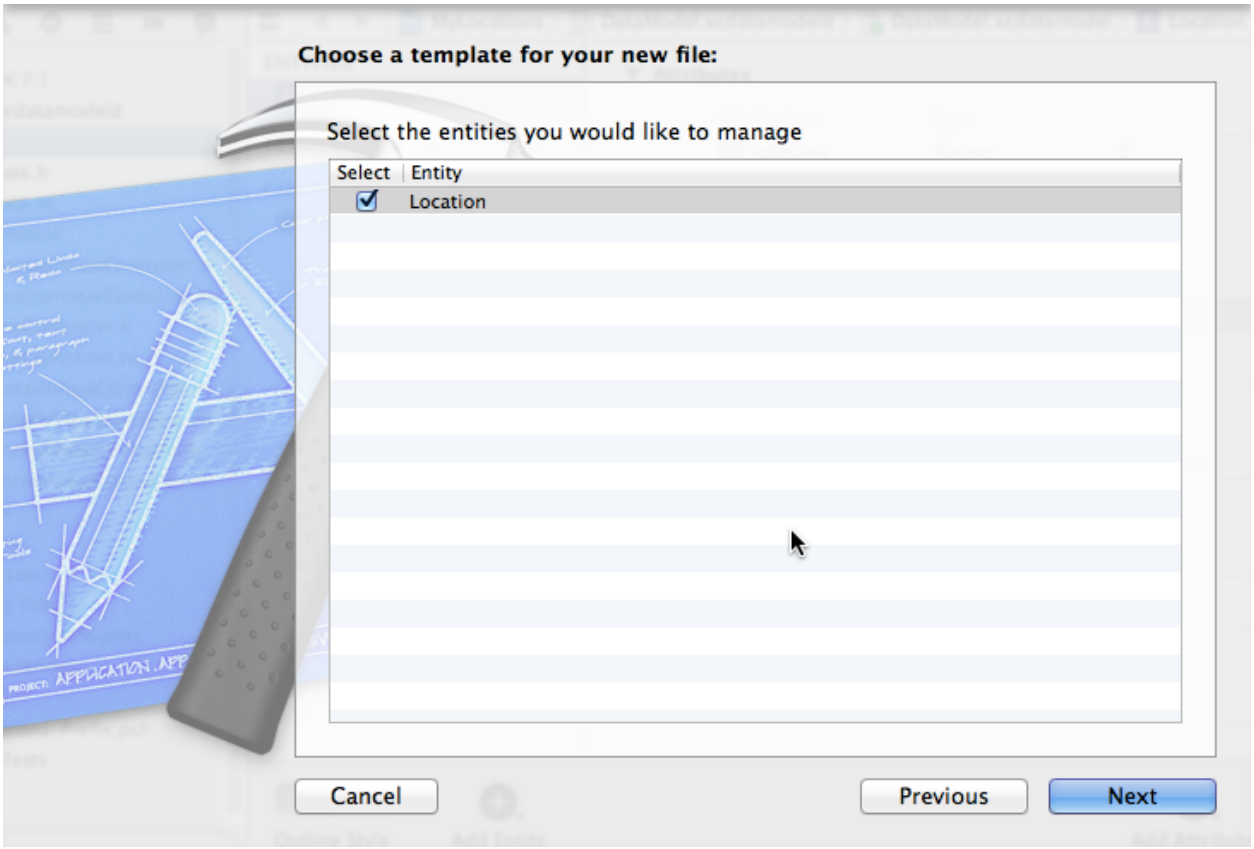
Xcode里的小技巧可以很方便的从数据模型中自动创建类。

接下来让我们向项目中添加一个新的文件。选择Core Data部分，然后选择NSManagedObject subclass模板：

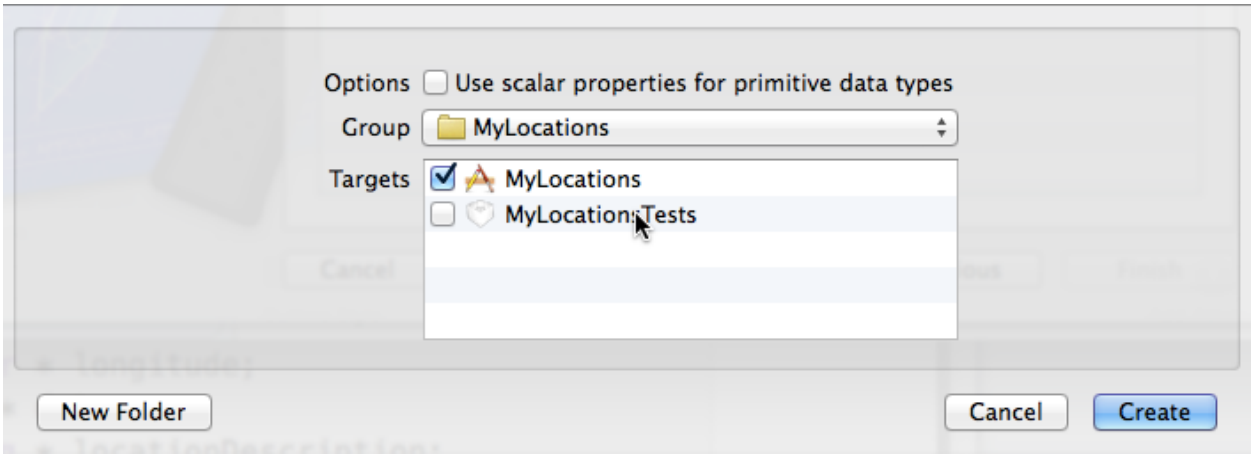


接下来我们需要选择对哪个数据模型和entity创建类：





确保选中Location这个Entity。接下来在提示选择哪里保存源文件的时候，要确保不要勾选Use scalar properties for primitive data types



此时项目中添加了两个新的文件，Location.h和Location.m。其中Location.h的头文件内容如下：

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

```
@interface Location : NSManagedObject

@property (nonatomic, retain) NSString * category;
@property (nonatomic, retain) NSDate * date;
@property (nonatomic, retain) NSNumber * latitude;
@property (nonatomic, retain) NSString * locationDescription;
@property (nonatomic, retain) NSNumber * longitude;
@property (nonatomic, retain) id placemark;

@end
```

在@interface这一行可以看到，Location类继承自NSManagedObject，而不是常规的NSObject。同时，Xcode还会Data Model编辑器中的attributes创建了对应的属性变量。

尽管我们为latitude何longtitude选择的数据类型是Double，但在这里显示的却是NSNumber对象。这是因为在Core Data中所有的东西都以对象的形式保存，而不是基本数据类型。任何常规基本数据类型，如int,float,double或BOOL都会在Core Data中以NSNumber的形式存在。

因为我们将placemark设置为Transformable类型的属性，Xcode不知道这种类型的对象究竟是神马东西，因此它选择了通用数据类型id。当然我们知道它实际上会是一个CLPlacemark对象，因此就知道该如何进行手动的调整了。

将placemark属性声明的语句更改为：

```
@property (nonatomic, retain) CLPlacemark *placemark;
```

Location.m中的内容非常简单：

```
#import "Location.h"


@implementation Location

@dynamic category;
@dynamic date;
@dynamic latitude;
@dynamic locationDescription;
@dynamic longitude;
@dynamic placemark;

@end
```

通常情况下属性变量有一个对应的实例变量来保存它的数值。但因为这里是一个managed 对象，数据在data store中生存，因此Core Data将以另外一种方式来处理属性。@dynamic关键字告诉编译器这些属性在运行时将由Core Data进行处理。当我们将一个新的数值保存到属性中时，Core Data为了安全考虑会把数值保存到data store中，而不是保存在实例变量中。

好了，通过上面的操作，我们就为这款应用定义了数据模型。接下来我们需要把它关联到一个data store中。

Data Store（数据存储）

在iOS开发中，Core Data将所有的数据保存在一个SQLite数据库中。如果你头一次听说这个名字，Don’t panic.后续我们会大致了解下数据库的概念，不过在初次使用Core Data的时候，你无须了解太多关于数据存储的内部原理。当然，我们需要在应用启动的时候初始化数据存储。对于任何使用Core Data的应用来说，相关代码都是相似的，我们将把它放在应用的app delegate类中。

app delegate用来获取关于应用整体的相关消息。比如在这里iOS会通知应用它已经启动了。接下来我们将对项目的AppDelegate类进行一些调整。

在Xcode中打开AppDelegate.m，在#import和@implementation之间添加以下代码：

```
@interface AppDelegate()

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

@end
```

记住在.m文件中的@interface语句属于类扩展。
此前我们在LocationDetailsViewController中曾经为其outlet属性变量添加了类扩展，目的是保持属性变量的私有性。
而这里我们把常规的属性放到类扩展中，因为这些属性将只会在AppDelegate.m中使用。

在@end之前添加以下的代码内容：

```
#pragma mark -Core Data

-(NSManagedObjectModel *)managedObjectModel{

    if(_managedObjectModel ==nil){
        NSString *modelPath = [[NSBundle mainBundle]pathForResource:@"DataModel"
ofType:@"momd"];
        NSURL *modelURL = [NSURL fileURLWithPath:modelPath];

        _managedObjectModel = [[NSManagedObjectModel
alloc]initWithContentsOfURL:modelURL];

    }
    return _managedObjectModel;

}

-(NSString*)documentsDirectory{

    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);

    NSString *documentsDirectory = [paths lastObject];

    return documentsDirectory;

}
```

```
-(NSString*)dataStorePath{

    return [[self documentsDirectory]stringByAppendingPathComponent:@"DataStore.sqlite"];

}

-(NSPersistentStoreCoordinator*)persistentStoreCoordinator{

    if(_persistentStoreCoordinator ==nil){

        NSURL *storeURL = [NSURL fileURLWithPath:[self dataStorePath]];

        _persistentStoreCoordinator = [[NSPersistentStoreCoordinator
alloc]initWithManagedObjectModel:self.managedObjectModel];

        NSError *error;

        if(![_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:nil URL:storeURL options:nil error:&error]){
            NSLog(@"Error adding persistent store %@, %@",error,[error userInfo]);

            abort();

        }


    }
          return _persistentStoreCoordinator;

}


-(NSManagedObjectContext*)managedObjectContext{

    if(_managedObjectContext == nil){

        NSPersistentStoreCoordinator *coordinator = self.persistentStoreCoordinator;

        if(coordinator !=nil){
            _managedObjectContext = [[NSManagedObjectContext alloc]init];
            [_managedObjectContext setPersistentStoreCoordinator:coordinator];

        }

    }
    return  _managedObjectContext;

}
```

以上代码用于加载之前所定义的数据模型，并连接到一个SQLite数据存储中。实际上对于任何采用Core Data的应用，以上代码的内容都是相同的。

为了让大家把精力集中在如何运用Core Data上，我暂时不会花费太多时间来解释以上代码的具体工作原理。从现在开始，我们唯一要关注的是NSManagedObjectContext。

当然，作为一个典型的强迫症患者，你已经对Xcode的红色错误提示忍无可忍了。这是因为我们还没有在AppDelegate.m中导入Core Data相关的头文件。
让我们在MyLocations-Prefix.pch中中添加以下代码：

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

传递context

当用户触碰Tag Location界面中的Done按钮时，应用目前只是将界面关闭而已。这里我们会做一些调整，这样当用户触碰Done按钮时会将一个新的Location对象保存到Core Data数据存储中。

之前我们提到过NSManagedObjectContext对象，该对象将用于和Core Data进行沟通。通常我们将其描述为“scratchpad”-暂存器。我们首先对context对象进行修改，然后调用其save方法将相关的变化信息永久保存到数据存储中。这就意味着所有需要保存到Core Data中的对象都需要有一个到NSManagedObjectContext对象的引用。

在Xcode中切换到LocationDetailsViewController.h，在其中添加一个新的属性变量：

```
@property(nonatomic,strong) NSManagedObjectContext *managedObjectContext;
```

问题在于：

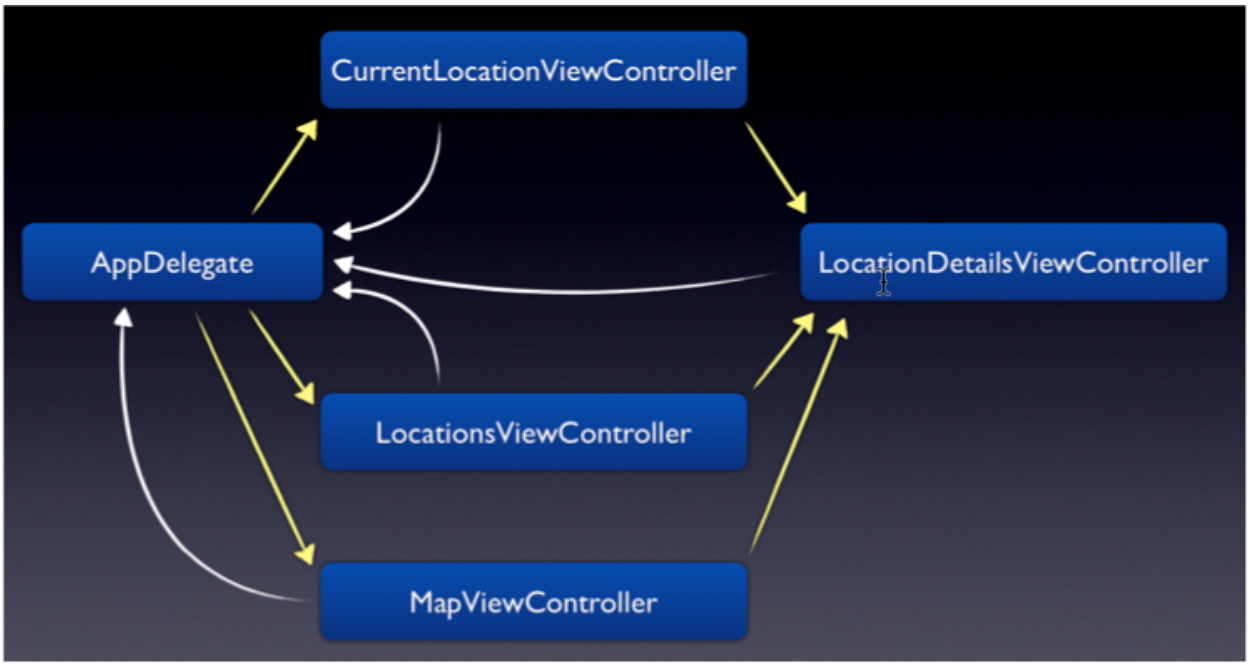
我们如何将NSManagedObjectContext保存到该属性中？context对象是由AppDelegate创建的（在刚才我们所添加的那一大堆代码中），但AppDelegate并没有到LocationDetailsViewController的引用。这一点并不奇怪，因为除非用户触碰了Tag Location中的按钮，否则这个视图控制器根本就不会存在。在初始化这个segue前，根本就没有LocationDetailsViewController对象。

答案是：
当CurrentLocationViewController打开Tag Location界面时，我们将在其prepareForSegue方法中设置managedObjectContext属性。因此现在我们首先需要找到一个方法将NSManagedObjectContext对象放到CurrentLocationViewController中。

我看到有不少类似下面这样的代码：

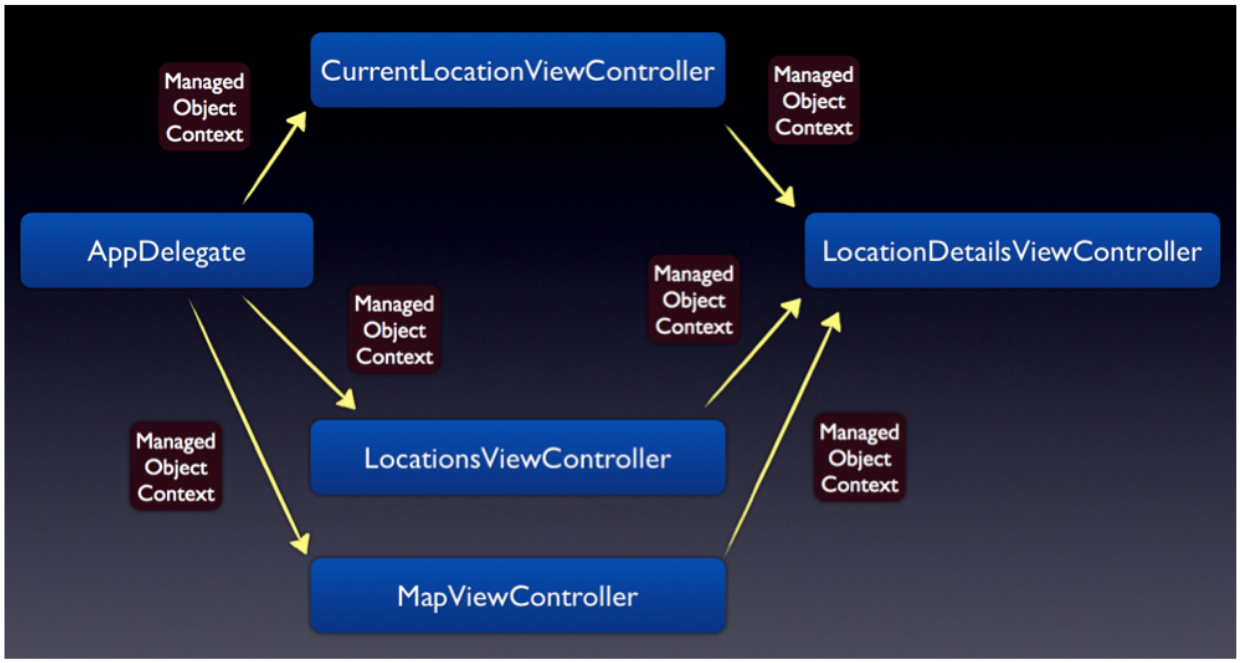
```
#import "AppDelegate.h"

...
AppDelegate *appDelegate = (AppDelegate *)[[NSApplication sharedApplication] delegate];

NSManagedObjectContext *context = appDelegate.managedObjectContext;

// do something with the context
```

如果采用这种方式，那么managedObjectContext必须是app delegate的一个属性（在.h文件中声明）。这样我们可以从项目中的任何一个地方获取到context的引用。看起来这个方法还是比较靠谱的，是吧？
实际上不是，因为这样会导致所有的对象都依赖于app delegate，这样会使得代码的依赖性变得迅速混乱。



通常来首，我们希望项目中的各个类尽量减少彼此的依赖性。项目中的各部分关联越少，则项目的结构就越清晰。如果很多类都需要和app delegate这样的对象关联起来，那么我们就需要重新思考代码的结构设计。

这里我们会采用一种更高的解决方案，就是将NSManagedObjectContext对象传递给每个需要它的对象。



通过采用以上架构，AppDelegate将会把managed object context传递给CurrentLocationViewController，而它又会在执行segue的时候继续将其传递给LocationDetailsViewControllller。

注意到我们将managedObjectContext属性放到AppDelegate的类扩展中，在AppDelegate.m中，我们可以自由使用该属性，但其它对象却无权向AppDelegate请求访问它。这样就可以避免其它类"滥用"app delegate。

在AppDelegate.m的顶部添加以下语句：

```
#import "CurrentLocationViewController.h"
```

然后更改didFinishLaunchingWithOptions方法的代码为：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UITabBarController *tabBarController = (UITabBarController*)self.window.rootViewController;

    CurrentLocationViewController *currentLocationViewController = (CurrentLocationViewController*) tabBarController.viewControllers[0];

    currentLocationViewController.managedObjectContext = self.managedObjectContext;

    return YES;
}
```

不幸的是，Interface Builder不允许我们为App Delegate中的视图控制器创建outlet属性。因此我们必须通过storyboard来查看这些视图控制器。为了获取到CurrentLocationViewController的引用，我们首先必须找到UITabBarController，并查看其viewControllers数组。

一旦我们获取了CurrentLocationViewController对象，就可以：

```
    currentLocationViewController.managedObjectContext = self.managedObjectContext;
```

这里我们使用self.managedObjectContext来获取到App Delegate的NSManagedObjectContext对象的指针，即便我们根本都没有创建这个对象。之所以可以这样做，是因为我们已经在类扩展中声明了managedObjectContext属性，同时我们也添加到该属性的对应getter方法。

注意：

当我们通过self.propertyName的方式来访问一个属性变量的时候，在幕后发生的事情是调用了该属性的getter方法。对一般的属性变量来说，getter方法只会返回实例变量的数值，但是当我们提供了自己的getter方法时，还能够同时处理一些其他的事情。

下面让我们来看看相关的getter方法：

```
-(NSManagedObjectContext*)managedObjectContext{

    if(_managedObjectContext == nil){

        NSPersistentStoreCoordinator *coordinator = self.persistentStoreCoordinator;
```

```
        if(coordinator !=nil){
            _managedObjectContext = [[NSManagedObjectContext alloc]init];
            [_managedObjectContext setPersistentStoreCoordinator:coordinator];

        }

    }
    return  _managedObjectContext;

}
```

代码的细节并不重要，重要的是我在这里所展示的设计原则。每当我们通过self.managedObjectContext来获取属性变量时，实际上就调用了该方法。当第一次这样操作的时候，实例变量_managedObjectContext是nil（因为所有实例变量的默认值是nil）。在这种情况下我们会创建一个新的NSManagedObjectContext对象，然后将其放到变量中。之后_managedObjectContext不再是nil,随后对该方法的调用只会返回已有的context对象。

以上方法就是所谓的lazy loading（懒加载）。除非我们需要，就不会创建context对象。注意这个方法还访问了self.persistentStoreCoordinator属性，这样就会懒加载persistent store coordinator，也就是处理SQLite 数据存储的对象。而接下来，persistenStoreCoordinator的getter方法会访问self.managedObjectModel属性，然后懒加载数据模型。当三个属性都被加载后，Core Data就可以开始使用了。

仅仅通过使用self.managedObjectContext属性，我们就设置了一个事件链，可以初始化整个Core Data 堆。这就是lazy loading的威力所在！

当然，CurrentLocationViewController仍然需要声明一个自己的NSManagedObject context属性。

在Xcode中切换到CurrentLocationViewController.h，添加以上属性变量的声明：

```
@property(nonatomic,strong) NSManagedObjectContext *managedObjectContext;
```

最后我们还需要将context传递到Tag Location界面，切换到CurrentLocationViewController.m，然后更改prepareForSegue方法的代码如下：

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{

    if([segue.identifier isEqualToString:@"TagLocation"]){
        UINavigationController *navigationController = segue.destinationViewController;
        LocationDetailsViewController *controller =
(LocationDetailsViewController*)navigationController.topViewController;

        controller.coordinate = _location.coordinate;
        controller.placemark = _placemark;
        controller.managedObjectContext = self.managedObjectContext;
    }

}
```

编译运行应用，一切应和往昔相似，但不同的是我们已经创建了一个新的数据库，同时也开始真正使用Core Data。

好了，今天的学习就到此结束，还是送上福利美女吧。

武大女神赏樱花，其实个人觉得也没啥