Welcome Code Pot!

Short introduction

Paweł Dudek

Maciej Oczko

Mobile Academy

Advanced iOS workshops

@mobacademy

Setup

WiFi

Login:
Password:

Repo

https://github.com/paweldudek/core-data-workshop-ios

Successfully working with

Core Data

Successfully

What does it mean?

Deep understanding of the framework

Knowledge about common pitfalls

What will you learn today?

The Basics

Multithreading

Performance

Let's get started!

What is Core Data?

It is *not* a database

The Core Data framework provides (…) object life-cycle and object graph management, including persistence.

Core Data is an object graph management framework

What is an object graph?

Groups of objects that form a network through their relationships with each other—either through a direct reference to another object or through a chain of intermediate references.

Core Data also providers persistence

You are not required to persist your objects

In fact you might *not* use a database as underlying storage at all

Core Data is storage-agnostic

Storage type is completely abstracted from top layer

Base classes

`NSManagedObject`

Generic class implementing basic Core Data model object behavior

How does it actually work?

Oversimplifying `NSManagedObject` is a dictionary

It acts as a key-value storage

Obtaining raw dictionary data

```
// NSManagedObject
- (NSDictionary *)committedValuesForKeys:(NSArray *)keys;
```

This will return only last saved/ retrieved data

Moreover this would be really painful to use, wouldn't it?

Obtaining and setting raw property data

```
// NSManagedObject
- (id)valueForKey:(NSString *)key;
- (void)setValue:(id)value forKey:(NSString *)key;
```

Having to call `valueForKey:` would make your code really hard to read

However you can subclass `NSManagedObject`

Core Data generates getters and setters based on model definition of your object

All you have to do is create a property and mark it as

`@dynamic`

```objc
// Plain NSManagedObject
NSManagedObject *employee = ...;
NSString *firstName = [employee valueForKey:@"firstName"];

// Subclass that defines a dynamic property firstName
Employee *employee = ...
NSString *firstName = [employee firstName];
```

Generated accessors are KVC compliant.

KVO is not officially supported out of the box.

Calling a setter will mark that property dirty

This will result in the object being saved upon next save operation

You can access `NSManagedObject` model description for introspection

```
@property(nonatomic, readonly, strong) NSEntityDescription *entity;
```

`NSManagedObject` **will be abbreviated as** `MO` **during this presentation**

`NSManagedObjectContext`

Single object space for managed objects

`NSManagedObjectContext` **is responsible for managing a collection of managed objects**

Managing means

- Retrieving data from underlying storage
- Modifying data in underlying storage
- Discarding unsaved changes
- Handling MO lifecycle, including faulting and unfaulting
- Handling MO validation
- Handling inverse relationships

A single managed object instance exists in one and only one `NSManagedObjectContext`

However you can have multiple `NSManagedObjectContext`s and each will have its own instance of given object

You *cannot* use objects from different contexts to establish relationships

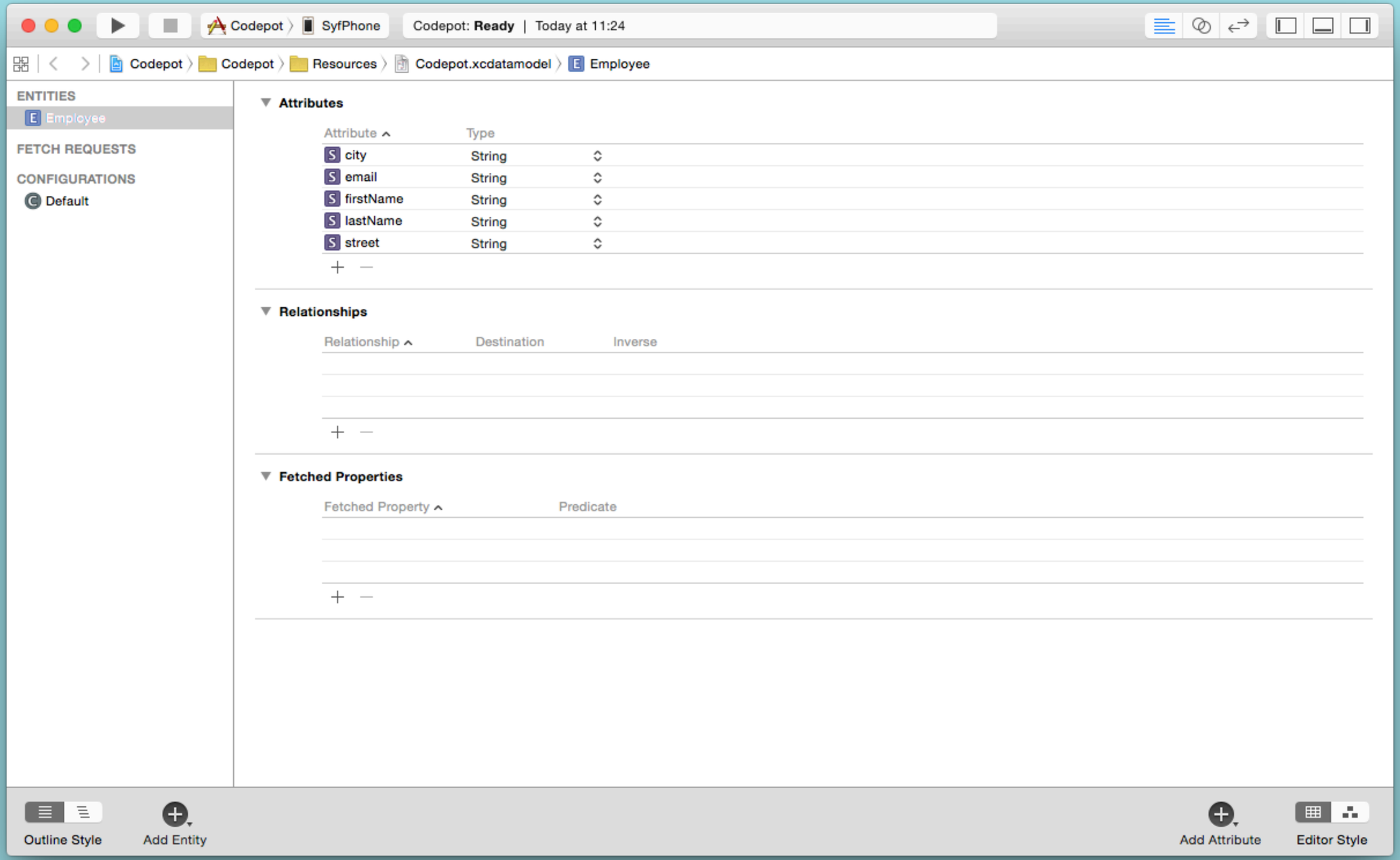
Creation

```
- (instancetype)initWithConcurrencyType:(NSManagedObjectContextConcurrencyType)type;
```

`NSManagedObjectContext` **will be abbreviated as** `MOC` **during this presentation**

`NSManagedObjectModel`

Defines the schema of your object graph


Codepot
SyfPhone
Codepot: Ready | Today at 11:24
Codepot
Codepot
Resources
Codepot.xcdatamodel
Employee
ENTITIES
Employee
FETCH REQUESTS
CONFIGURATIONS
Default
Attributes
Attribute
Type
city
String
email
String
firstName
String
lastName
String
street
String
Relationships
Relationship
Destination
Inverse
Fetched Properties
Fetched Property
Predicate
Outline Style
Add Entity
Add Attribute
Editor Style

Entity

Entity == `NSEntityDescription`

Entity

- Defines a set of attributes (properties) given object has
- Allows to define additional rules for each property (optional, validation, indexing etc)
- Defines relationships
- Defines fetched properties

Schema also defines

Fetch Requests and Configurations

Creating NSManagedObjectModel

```
- (instancetype)initWithContentsOfURL:(NSURL *)url;

// You can nil to use main bundle
+ (NSManagedObjectModel *)mergedModelFromBundles:(NSArray *)bundles;
```

NSPersistentStoreCoordinator

Coordinates persistent stores based on
NSManagedObjectModel

`NSPersistentStoreCoordinator`

Provides façade for for `NSManagedObjectContext`

MOC **uses persistent store coordinator to save, update, delete or retrieve objects in its graph.**

`MOC` cannot fully work without a persistent store coordinator

Creating `NSPersistentStoreCoordinator`

```
- (instancetype)initWithManagedObjectModel:(NSManagedObjectModel *)model;
```

`NSPersistentStoreCoordinator` **will be abbreviated as** `PSC` **during this presentation**

`NSPersistentStore`

Provides an interface for accessing actual storage. Translates requests from `PSC` into actual fetches to underlying storage.

Store Types

- NSSQLiteStoreType
- NSBinaryStoreType
- NSInMemoryStoreType
- NSXMLStoreType (not available for iOS)

You can build your own store by subclassing `NSAtomicStore` or `NSIncrementalStore`

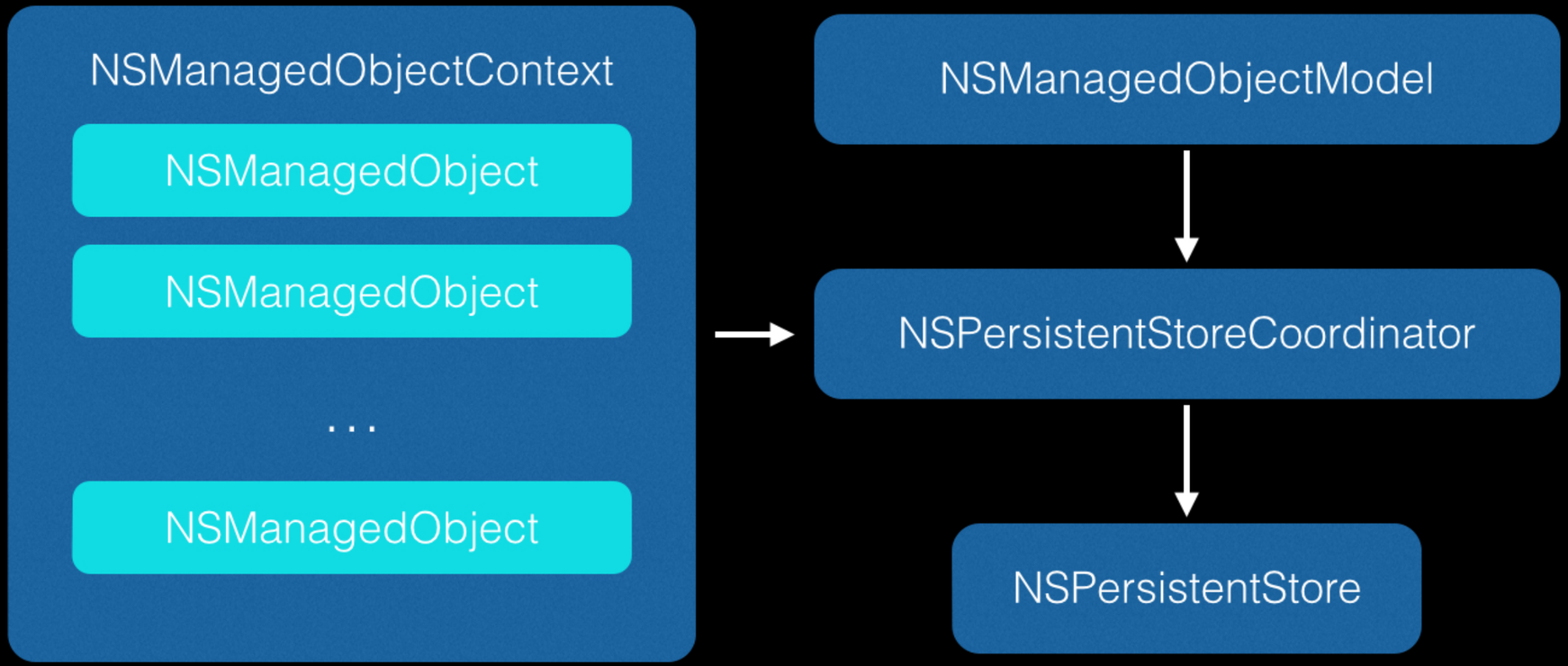
Creating stores

```
// NSPersistentStore
- (instancetype)initWithPersistentStoreCoordinator:(NSPersistentStoreCoordinator *)coordinator
                                   configurationName:(NSString *)configurationName
                                                 URL:(NSURL *)url
                                             options:(NSDictionary *)options;
```

Creating stores

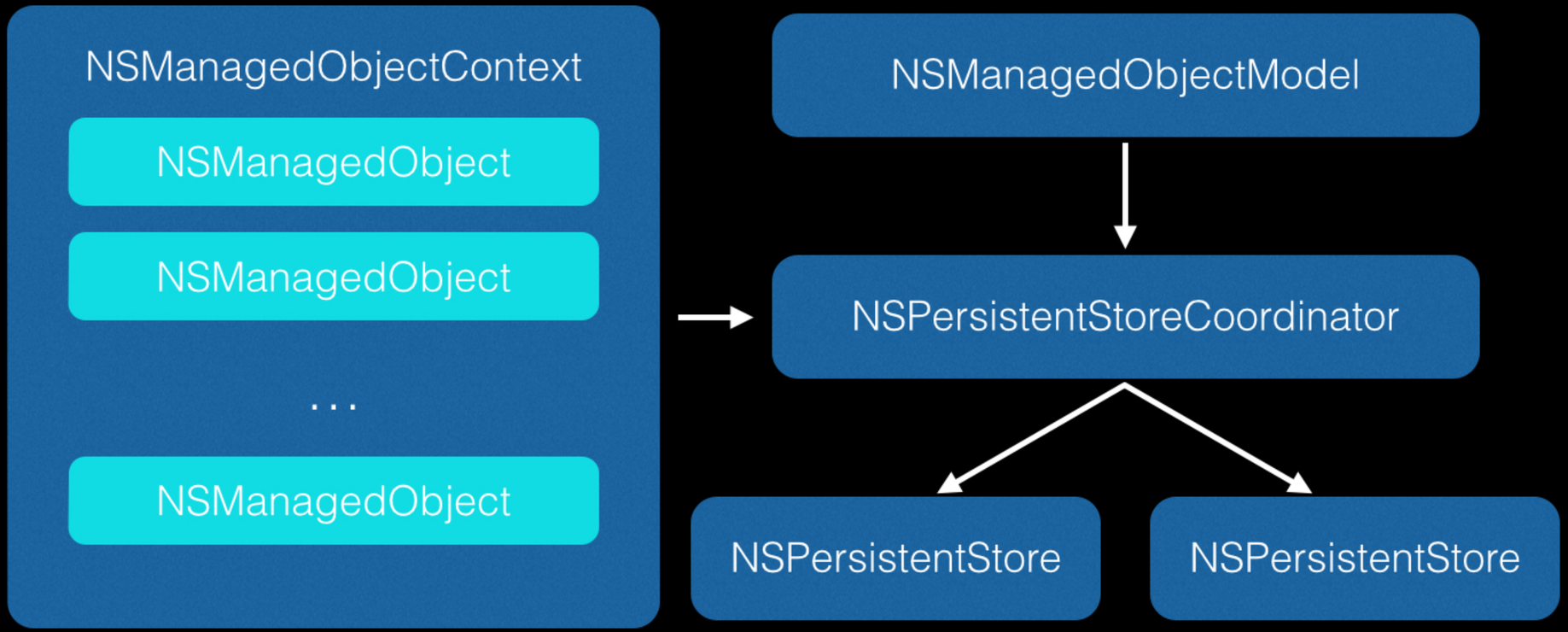
```
//NSPersistentStoreCoordinator
- (NSPersistentStore *)addPersistentStoreWithType:(NSString *)storeType
                                    configuration:(NSString *)configuration
                                              URL:(NSURL *)storeURL
                                          options:(NSDictionary *)options
                                            error:(NSError **)error
```

Single
NSPersistentStore


NSManagedObjectContext
NSManagedObject
NSManagedObject
...
NSManagedObject
NSManagedObjectModel
NSPersistentStoreCoordinator
NSPersistentStore

Multiple `NSPersistentStore`

Makes things quite complicated


NSManagedObjectContext
NSManagedObject
NSManagedObject
...
NSManagedObject
NSManagedObjectModel
NSPersistentStoreCoordinator
NSPersistentStore
NSPersistentStore

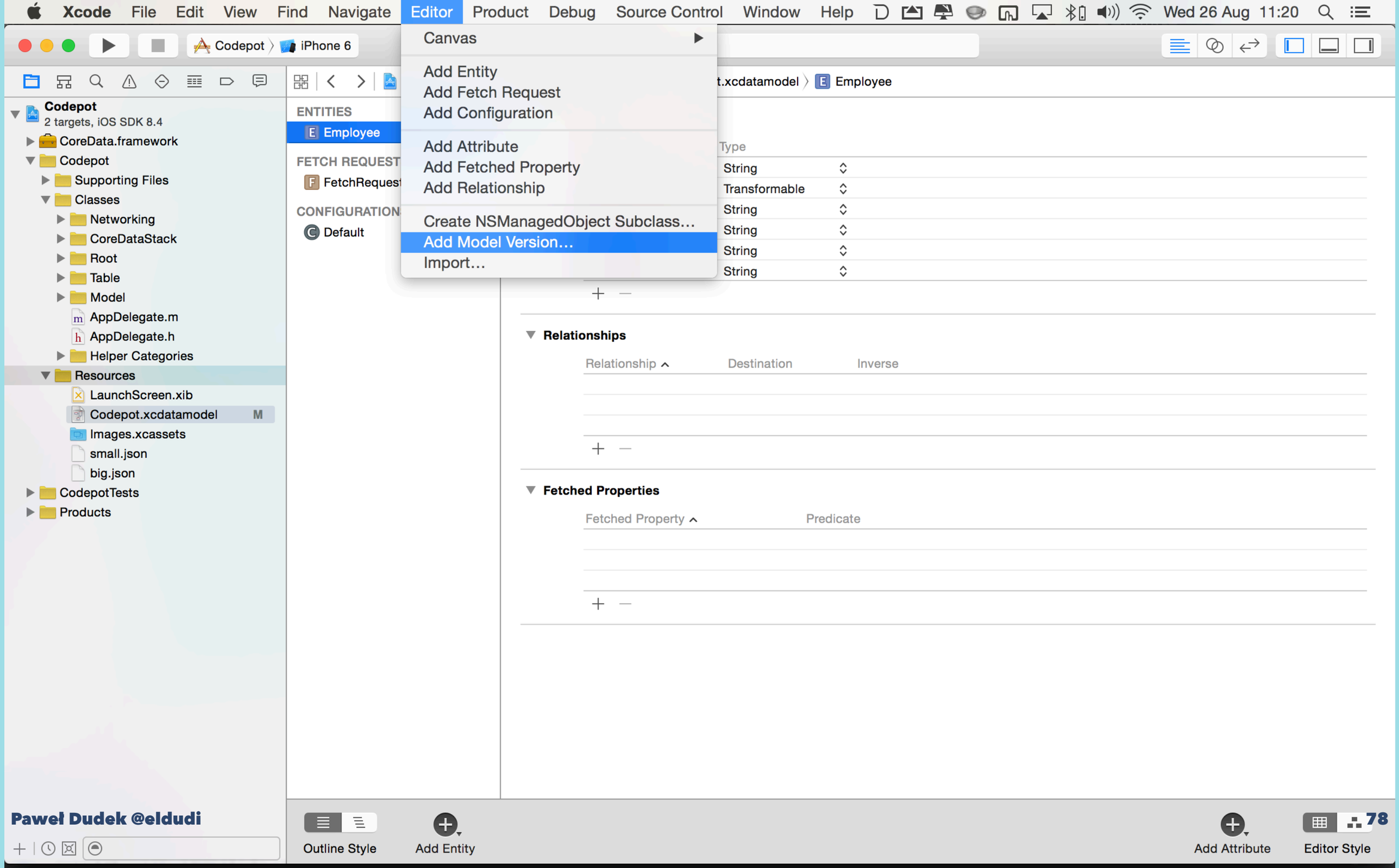
Core Data Versioning

The short explanation

`NSManagedObjectModel` **contains information about how to read/write to underlying storage**

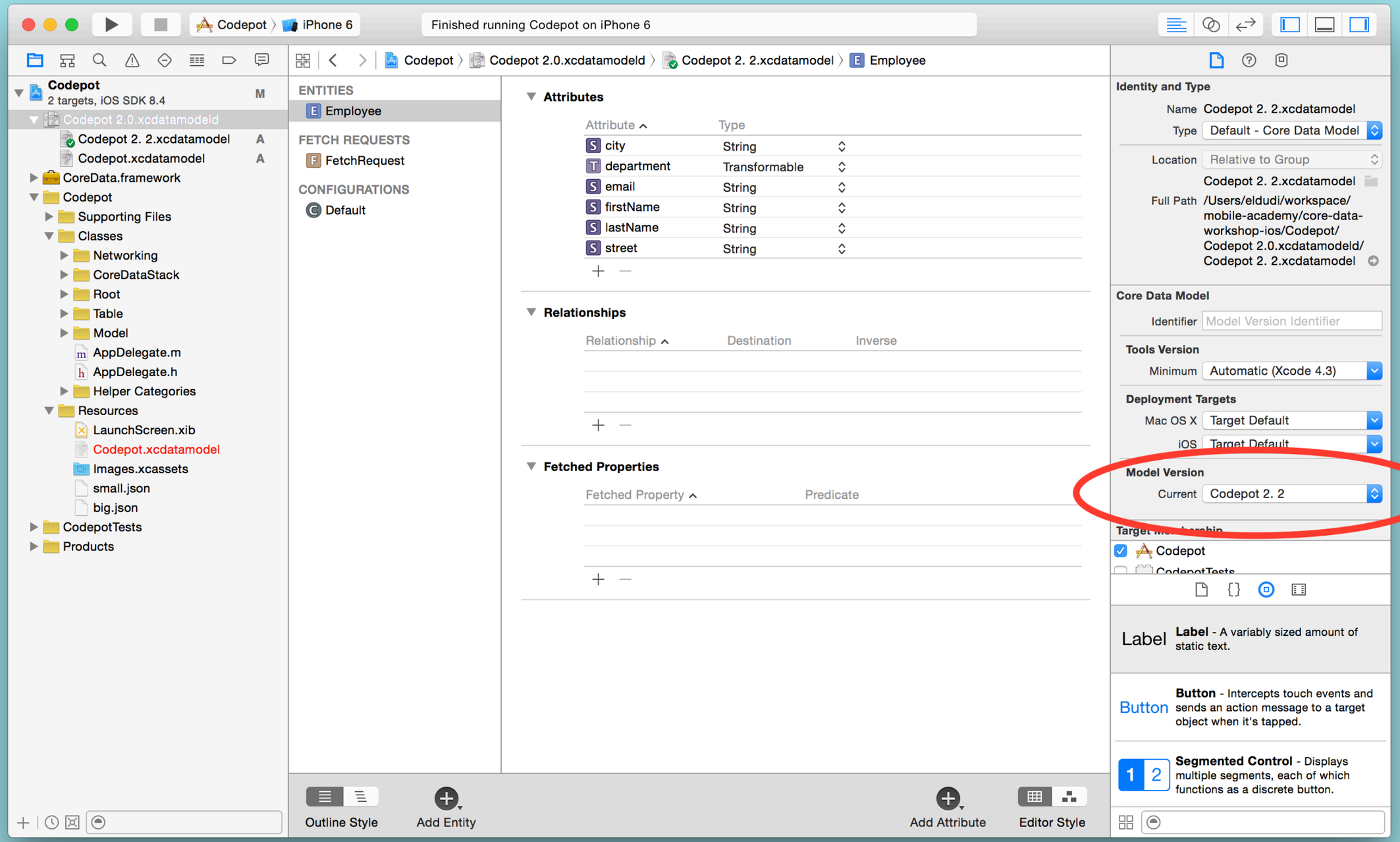
Changing anything in your model will mean that Core Data will not be able to read/write to given store using that model

That's why you should use model versions


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help Wed 26 Aug 11:20
Codepot iPhone 6
Canvas
Add Entity
Add Fetch Request
Add Configuration
Add Attribute
Add Fetched Property
Add Relationship
Create NSManagedObject Subclass…
Add Model Version…
Import…
Codepot
2 targets, iOS SDK 8.4
CoreData.framework
Codepot
Supporting Files
Classes
Networking
CoreDataStack
Root
Table
Model
AppDelegate.m
AppDelegate.h
Helper Categories
Resources
LaunchScreen.xib
Codepot.xcdatamodel M
Images.xcassets
small.json
big.json
CodepotTests
Products
ENTITIES
Employee
FETCH REQUEST
FetchRequest
CONFIGURATION
Default
t.xcdatamodel
Employee
Type
String
Transformable
String
String
String
String
Relationships
Relationship
Destination
Inverse
Fetched Properties
Fetched Property
Predicate
Pawel Dudek @eldudi
Outline Style
Add Entity
Add Attribute
Editor Style
78

By using model versions you ensure Core Data has access to all previous versions and is capable of performing a migration

After adding new model version you just have to tell Core Data what is the latest version


Codepot
iPhone 6
Finished running Codepot on iPhone 6
Codepot
2 targets, iOS SDK 8.4
Codepot 2.0.xcdatamodeld
Codepot 2. 2.xcdatamodel
Codepot.xcdatamodel
CoreData.framework
Codepot
Supporting Files
Classes
Networking
CoreDataStack
Root
Table
Model
AppDelegate.m
AppDelegate.h
Helper Categories
Resources
LaunchScreen.xib
Codepot.xcdatamodel
Images.xcassets
small.json
big.json
CodepotTests
Products
Codepot 2.0.xcdatamodeld
Codepot 2. 2.xcdatamodel
Employee
ENTITIES
Employee
FETCH REQUESTS
FetchRequest
CONFIGURATIONS
Default
Attributes
Attribute
Type
city String
department Transformable
email String
firstName String
lastName String
street String
Relationships
Relationship
Destination
Inverse
Fetched Properties
Fetched Property
Predicate
Outline Style
Add Entity
Add Attribute
Editor Style
Identity and Type
Name Codepot 2. 2.xcdatamodel
Type Default - Core Data Model
Location Relative to Group
Codepot 2. 2.xcdatamodel
Full Path /Users/eldudi/workspace/mobile-academy/core-data-workshop-ios/Codepot/Codepot 2.0.xcdatamodeld/Codepot 2. 2.xcdatamodel
Core Data Model
Identifier Model Version Identifier
Tools Version
Minimum Automatic (Xcode 4.3)
Deployment Targets
Mac OS X Target Default
iOS Target Default
Model Version
Current Codepot 2. 2
Codepot
Label
Label - A variably sized amount of static text.
Button
Button - Intercepts touch events and sends an action message to a target object when it's tapped.
1 2
Segmented Control - Displays multiple segments, each of which functions as a discrete button.

You can define your own migration if model changed so much that Core Data cannot infer mapping model

NSManagedObject

Detailed look

Creating

```
// NSManagedObject
- (instancetype)initWithEntity:(NSEntityDescription *)entity
insertIntoManagedObjectContext:(NSManagedObjectContext *)context;

// If you passed nil as MOC you will also have to insert your MO into a MOC:

// NSManagedObjectContext
- (void)insertObject:(NSManagedObject *)object;
```

Creating

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

NSManagedObject
Lifecycle

`NSManagedObject` lifecycle differs from standard objects

```
@property (nonatomic, getter=isInserted, readonly) BOOL inserted;
@property (nonatomic, getter=isDeleted, readonly) BOOL deleted;
@property (nonatomic, getter=isFault, readonly) BOOL fault;
```

Inserted

Denotes whether object has been inserted into a `NSManagedObjectContext`

```
// NSManagedObject
- (instancetype)initWithEntity:(NSEntityDescription *)entity
insertIntoManagedObjectContext:(NSManagedObjectContext *)context;

// This will automatically call insertObject: if a MOC was provided.
// If you passed nil as MOC you will also have to insert your MO into a MOC:

// NSManagedObjectContext
- (void)insertObject:(NSManagedObject *)object;
```

```
// NSEntityDescription will automatically call insertObject: on the MOC
+ (id)insertNewObjectForEntityForName:(NSString *)entityName
               inManagedObjectContext:(NSManagedObjectContext *)context;
```

`awakeFromInsert`

Called after your `MO` has been inserted

Deleted

Denotes whether object has been scheduled for deletion

`prepareForDeletion`

Called automatically when the receiver is about to be deleted.

Deleting objects

```
NSManagedObjectContext *context = ...;

Employee *employee = ...;

[context deleteObject:employee]; // prepareForDeletion will *not* be called here

[context save:nil]; // prepareForDeletion will be called here
```

Let's talk about

Faults

An object is faulted if its data has not actually been retrieved from data storage

Accessing any data-related property will cause the fault to fire and will result in a fetch to the database

Database access is usually resource-consuming

Remember to avoid firing faults if accessing given object is not necessary at given moment

Database access is usually resource-consuming

Avoid many round trips to database by requesting already unfaulted objects

`willTurnIntoFault`

`didTurnIntoFault`

Called when `MOC` discards cached data

`awakeFromFetch`

Called when object has been fetched from database

Saving

`willSave/didSave`

Called when object is saved to a persistent store

Remember that Core Data automatic change detection is enabled for this method

Modifying properties, even when their values didn't change, will result in an infinite loop

Solution: if statement to check whether value actually changed

State

Final Thoughts

You should consider `awakeFromInsert` / `prepareForDeletion` **as** `init`/`dealloc` **methods for logic that should only be called when given** `MO` **is created/deleted.**

You should consider `awakeFromFetch` / `didTurnIntoFault` as `init`/`dealloc` methods for logic that should only be called when given `MO` is retrieved / data is discarded.

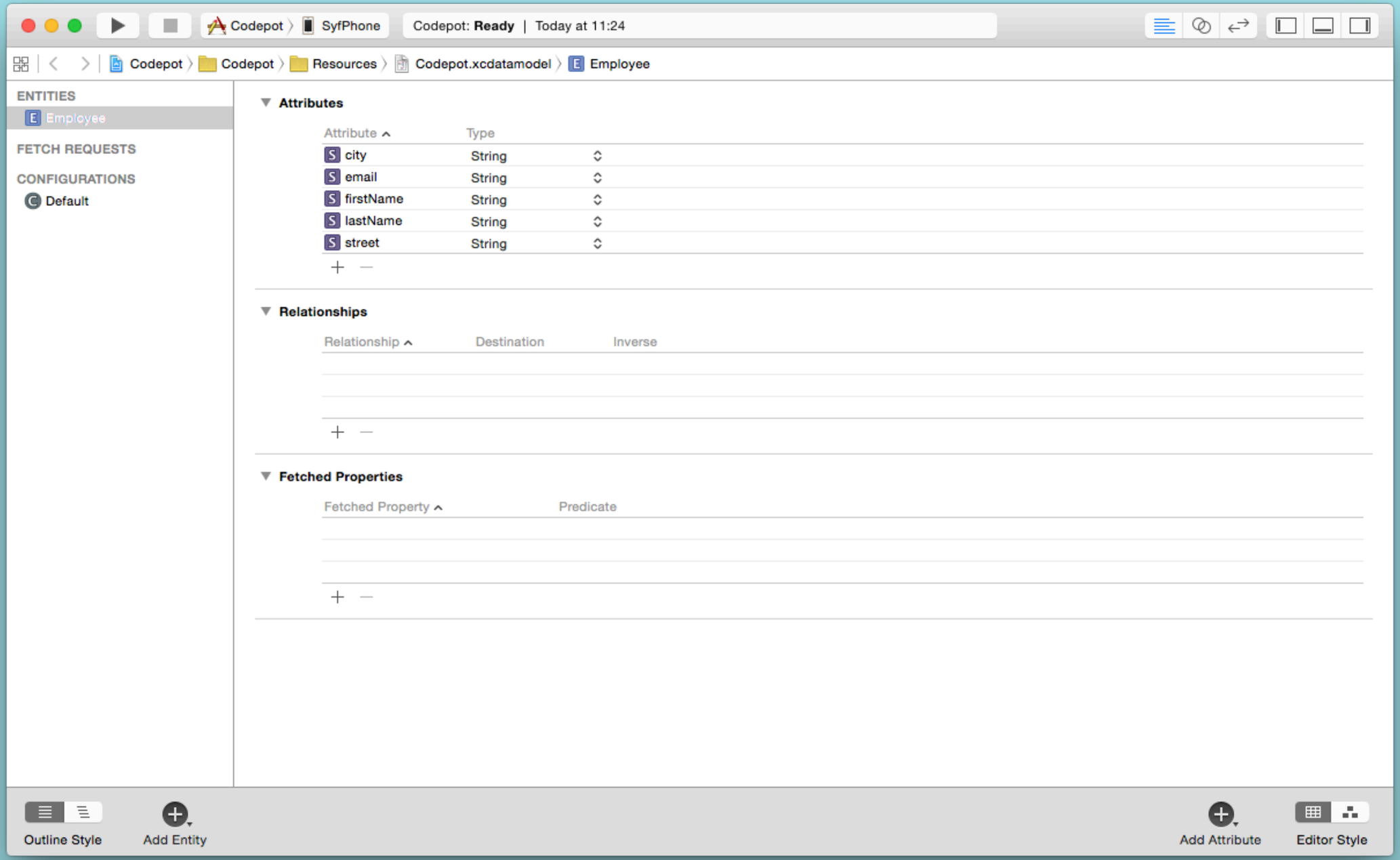
`NSManagedObject` **ownership**

Core Data reserves exclusive control over the life cycle of the managed object (that is, raw memory management)

Forcing refresh

```
// NSManagedObjectContext
- (void)refreshObject:(NSManagedObject *)object mergeChanges:(BOOL)flag;
```

Schema and property types


Codepot
SyfPhone
Codepot: Ready | Today at 11:24
Codepot
Codepot
Resources
Codepot.xcdatamodel
Employee
ENTITIES
Employee
FETCH REQUESTS
CONFIGURATIONS
Default
Attributes
Attribute
Type
city
String
email
String
firstName
String
lastName
String
street
String
Relationships
Relationship
Destination
Inverse
Fetched Properties
Fetched Property
Predicate
Outline Style
Add Entity
Add Attribute
Editor Style

Most values are obvious

- `date -> NSDate`
- `integer -> NSInteger`
- `decimal -> NSNumber`

etc

But there are some interesting ones...

Transformable Properties

Probably one of the most underlooked gems in Core Data

Sometimes you just don't need to create a managed object for given object type

Perhaps you have a non-standard object type that you want to store as a separate object.

You don't want to handle relationships and object creation deletion. And you don't really need to search.

This is when `transformable` property really shines

`transformable` **'transforms' your object using** `NSCoding` **and saves it in the database**

Thanks to `NSCoding`
`NSDictionary/NSArray`
work out of the box

Remember that you **_cannot_** use these properties for predicates when fetching data!

Relationships

Defines a relation between objects in your object graph

Most relationships should be bi-directional

They should have an inverse relationship (or two ends)

In fact if they don't Core Data will produce a warning when compiling your schema

Relationships ends can be to-many or to-one

Types

- One-to-One
- One-to-Many
- Many-to-Many

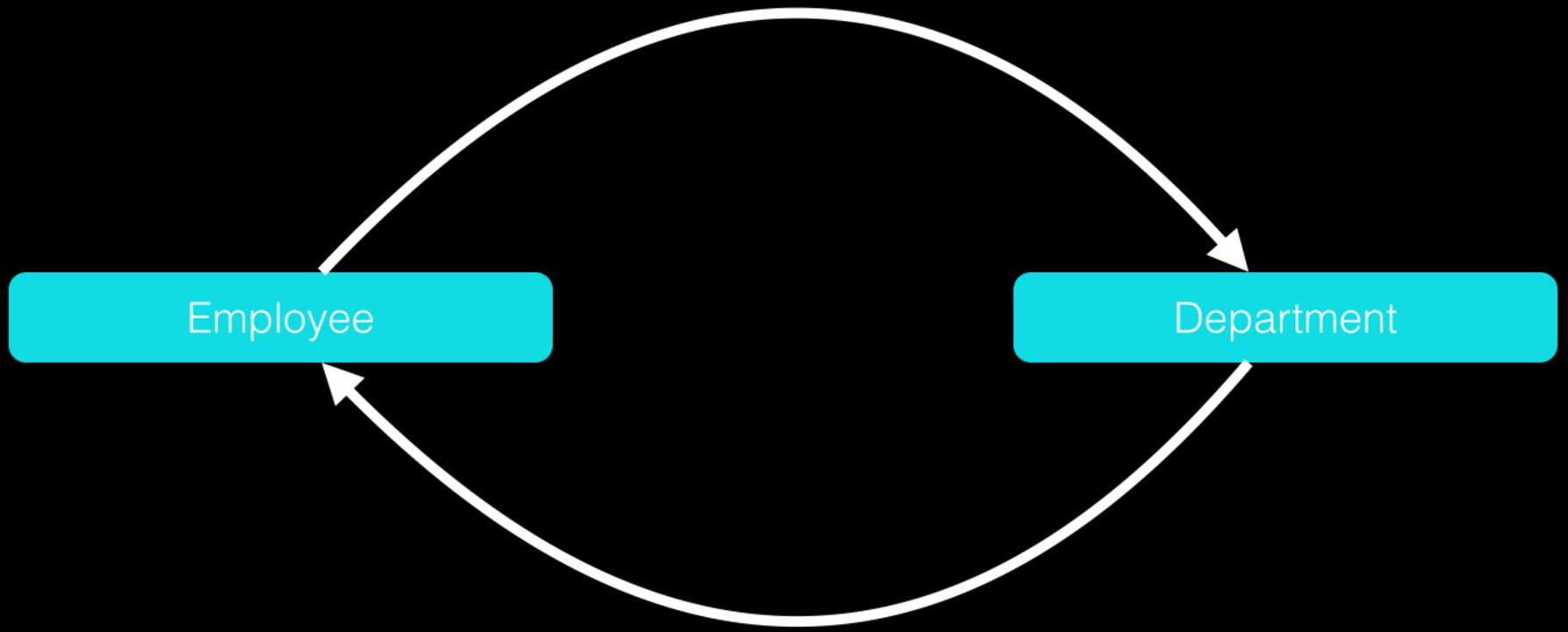
Relationships can be *reflexive*

Employee can have a relationship `manager` that points to another employee

Or not

Employee can have a relationship `department` **that points to a** `Department` **class**

department

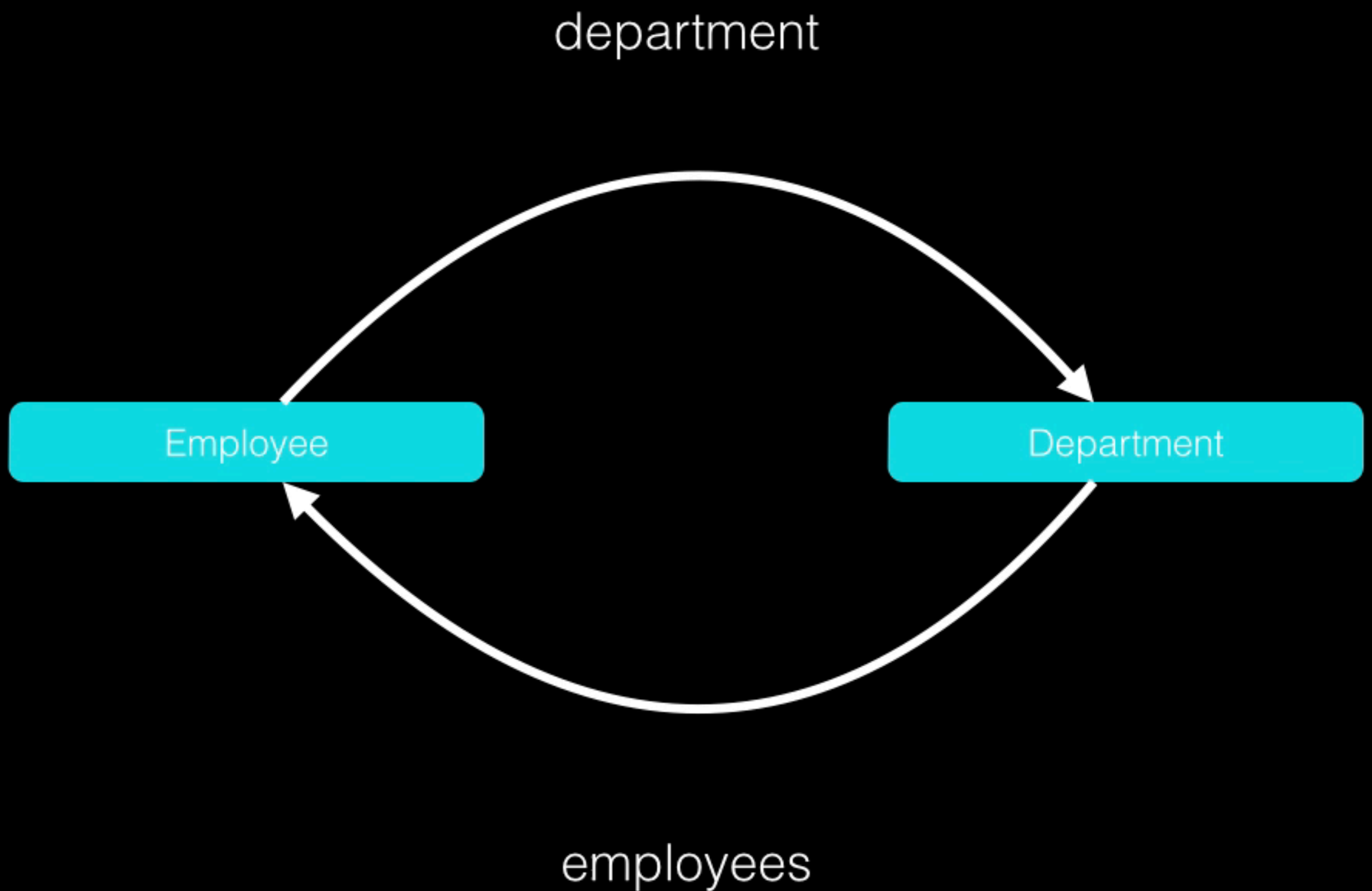


employees

To-many relationship ends are represented by `NSSet`

Delete rules

Defines how relationships behave when an object is deleted


department
Employee
Department
employees

Deny

Object cannot be deleted if it has objects in given relationship

Nullify

Set the inverse relationship to null

Cascade

Delete objects in relationship when given object in deleted

No Action

Do nothing

You are responsible for making sure your object graph is consistent when using No Action delete rule

Fetching Data

NSFetchRequest

NSManagedObjectContext

```
- (NSArray *)executeFetchRequest:(NSFetchRequest *)request error:(NSError **)error;
```

NSFetchRequest

```
@property (nonatomic, strong) NSEntityDescription *entity;
@property (nonatomic, strong) NSPredicate *predicate;
@property (nonatomic, strong) NSArray *sortDescriptors;
```

Entity

Defines what entity should be used for fetching data

Helper methods

```
+ (instancetype)fetchRequestWithEntityName:(NSString*)entityName NS_AVAILABLE(10_7, 4_0);
- (instancetype)initWithEntityName:(NSString*)entityName NS_AVAILABLE(10_7, 4_0);
```

Predicate

Defines requirements for objects that should be fetched. Can be nil.

Example predicate

```
[NSPredicate predicateWithFormat:@"firstName == %@", @"Paweł"]
```

```
BETWEEN, BEGINSWITH, ENDSWITH, CONTAINS, LIKE, MATCHES
ANY, SOME, ALL, NONE, IN
```

Sort Descriptors

Array of `NSSortDescriptor` objects

`NSSortDescriptor` **defines property to sort by and direction (ascending/descending) of the sort**

Saving Data

```
// NSManagedObjectContext
- (BOOL)save:(NSError **)error
```

Remember that there is a variety of reasons why save might fail!

Most popular Save Failure reasons

- Invalid database schema (wrong `NSManagedObjectModel`)
- Validation
- Database is not accessible
- Relationships deny the save (Deny delete rule)

Remember to keep a balance in how often you save

More saves

Less memory overhead, but more expensive trips to the database

Less saves

Higher memory overhead, but less expensive trips to the database

That's it!

Things you should check out as well

- Migrations
- `NSManagedObjectContext` save notifications
- Fetched and Transient Properties
- `NSFetchedResultsController`
- Undo/redo and Core Data

Enough talking!

Let's get our hands dirty

The App

Assignment I

Getting Employees on the screen

- Wire the `RootViewController` table view controller logic
- Implement `Employee` properties accordingly to data model
- Implement data parsing/saving in `ModelController`
- Implement data fetching in `EmployeesItemsProvider`
- Implement `Printable` category of `Employee` (first + last name as title, department name as subtitle)
- Department should be a transformable property

Tip: Use `TODO`s

Assignment II

Duplicates

Why is this happening?

Core Data does not know our objects should be unique

Find or create pattern

- Check whether object exists before creating it
- If it does - fetch it and update it
- If not - create and update it

Use `email` as unique key

Thanks for listening!

@eldudi
hello@dudek.mobi
github.com/paweldudek