



apple WATCH APPLICATION DEVELOPMENT

MPCS 51032 • SPRING 2020 • SESSION 5

CORE DATA

CORE DATA

Technique	Pros	Cons
Archiving	Allows ordered relationships (arrays, not sets). Easy to deal with versioning.	Reads all the objects in (no faulting). No incremental updates.
Web Service	Makes it easy to share data with other devices and applications.	Requires a server and a connection to the internet.
Core Data	Lazy fetches by default. Incremental updates.	Versioning is awkward (but can certainly be done using an NSModelMapping). No real ordered relationships (at the time this is being written).

- Options for data storage and persistence

CORE DATA

WHAT IS CORE DATA

- A framework that provides object-relational mapping
 - Converts objects into data stored on disk
- Creates a visual mapping between database and objects



CORE DATA

WHAT IS CORE DATA?

- An object-oriented API that conforms to MVC paradigm
- API for constructing detailed queries
- Access the data using **properties** on the objects



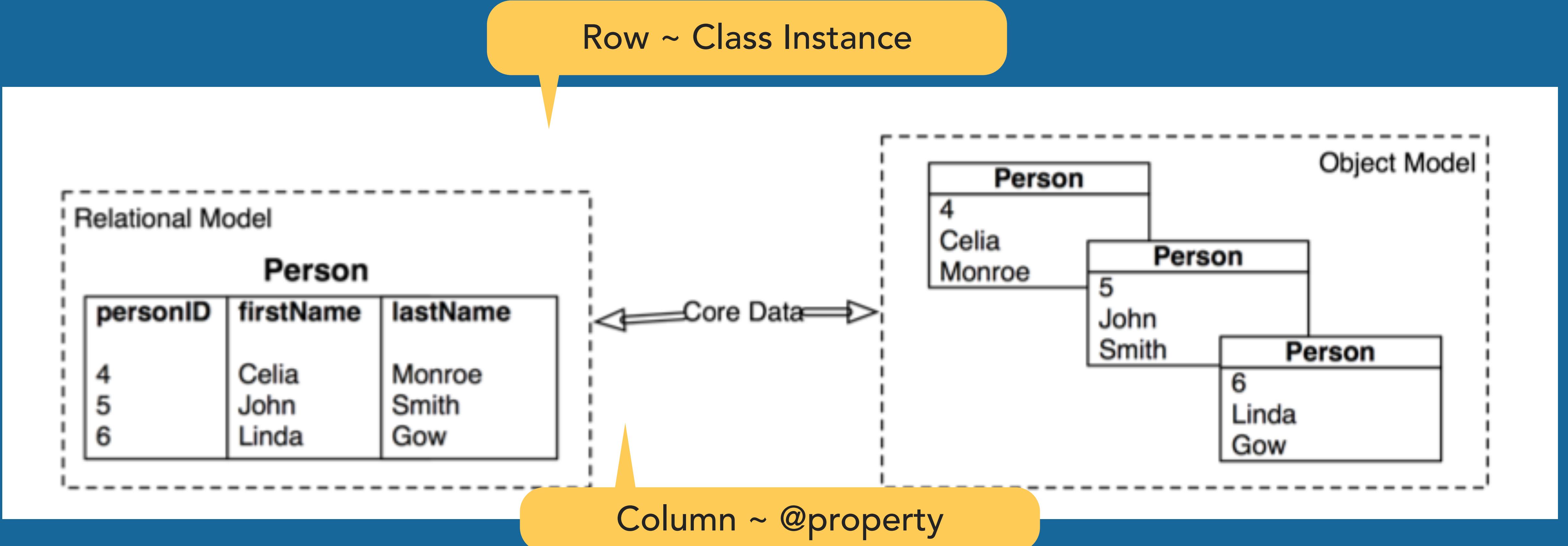
CORE DATA

WHAT IS CORE DATA?

- Core Data is an object graph management and persistence framework
- **Core Data is not a database!**



CORE DATA



- RDB table =~ class (entity)
 - Column =~ class instance variables (attributes)
 - Row =~ an instance of the class
- Core Data model file is the description of every entity and their attributes in your application

CORE DATA

WHAT YOU SHOULD KNOW ABOUT CORE DATA

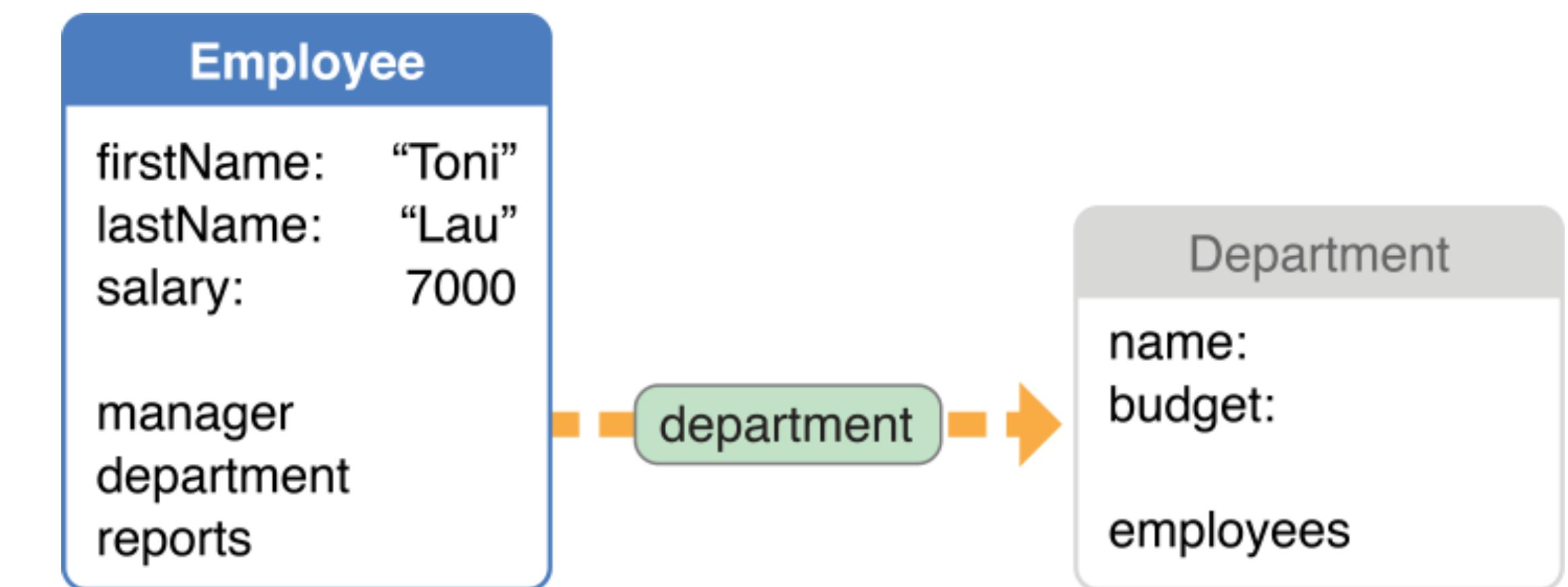
- Object graph store
 - Model layer represent state of your app
- Persistence
 - Read/write state of model objects to disk
- Querying
 - SQL-like parameter based queries
- Versioning
 - Lightweight migration



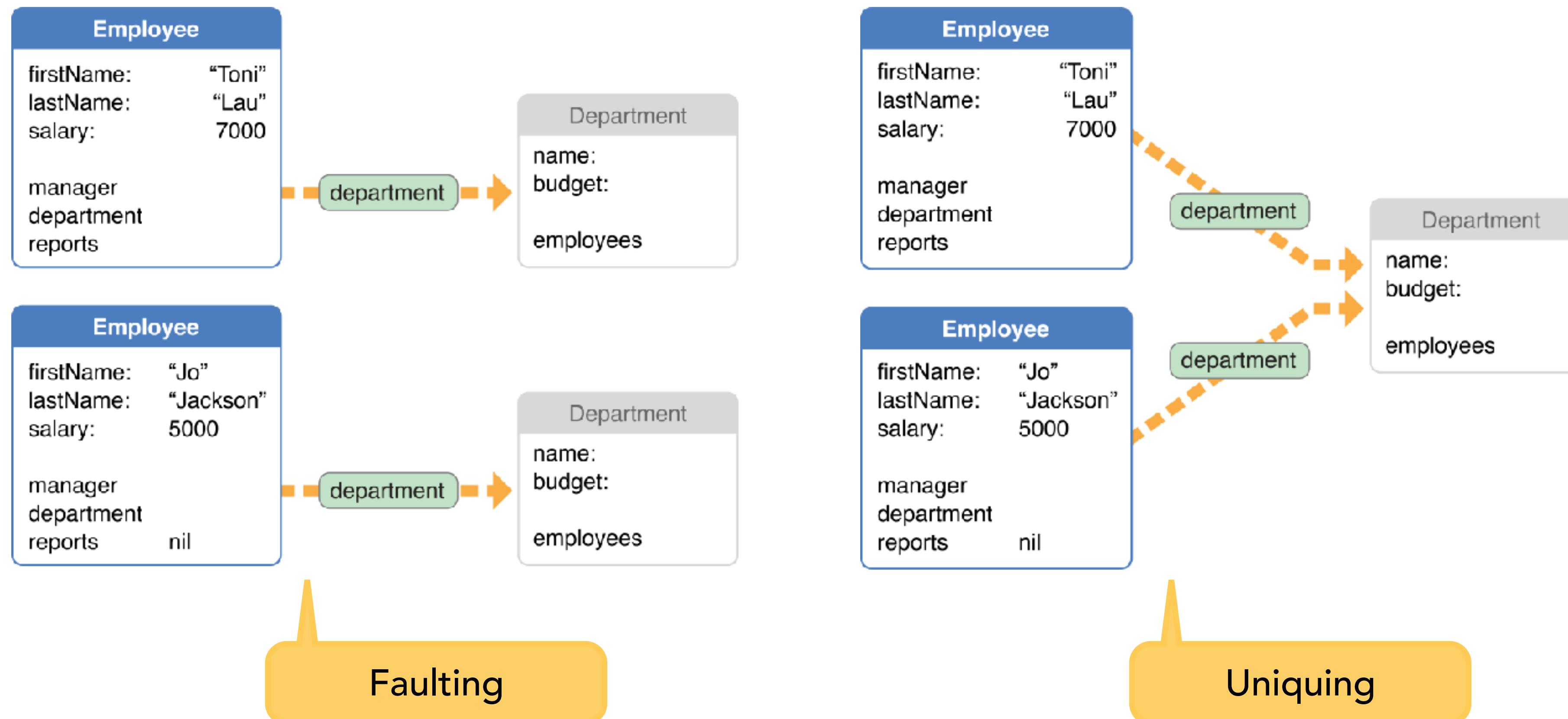
CORE DATA

WHAT YOU PROBABLY DO NOT KNOW

- Change tracking
- Optimize data work flows while in memory
- Faulting
 - Reduce application's memory usage
- Uniquing
 - Ensures that, in a given managed object context, you never have more than one managed object to represent a given record

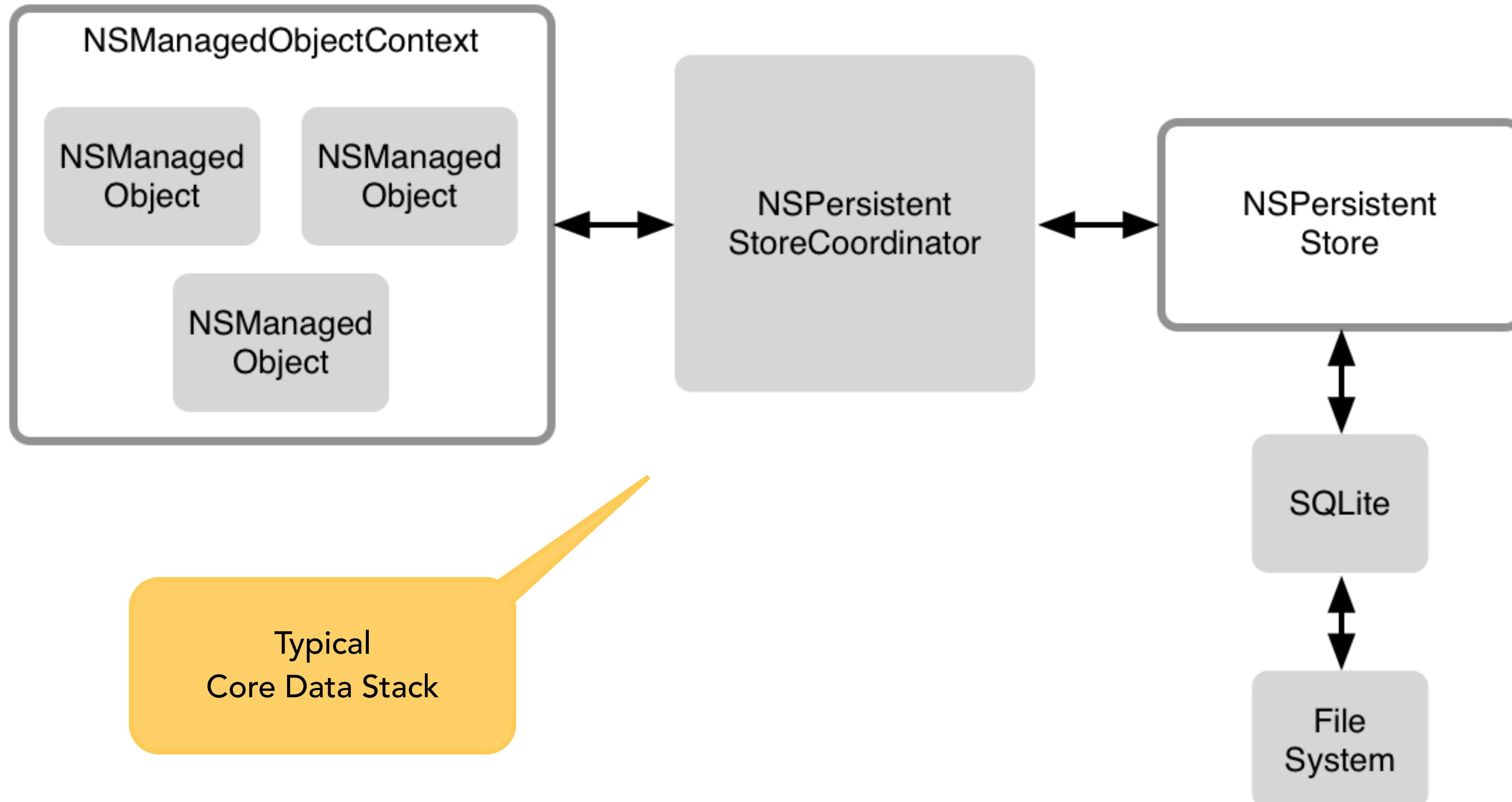


CORE DATA

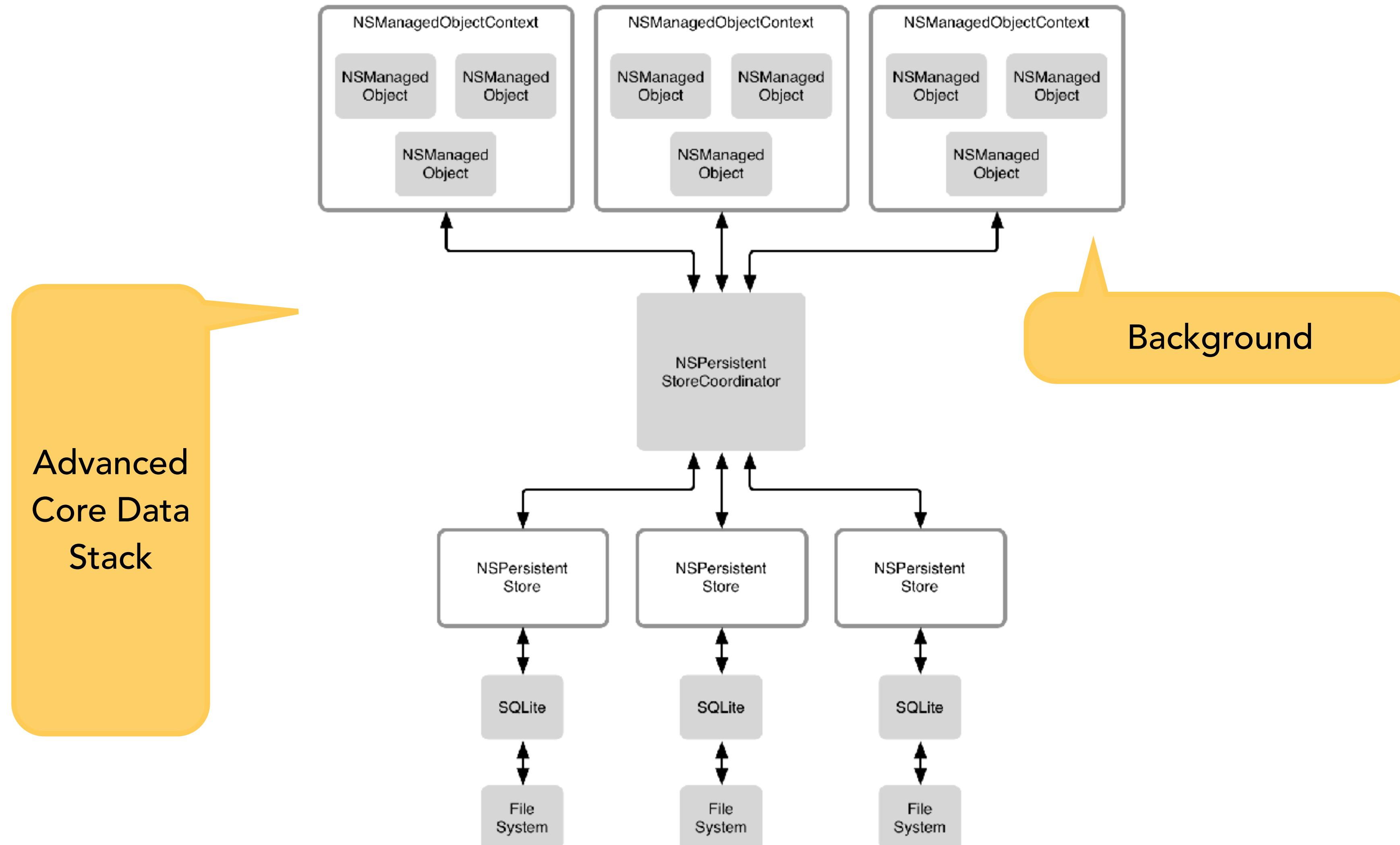


Faulting is a mechanism Core Data employs to reduce your application's memory usage
Uniquing ensures that you never have more than one managed object to represent a given record

CORE DATA



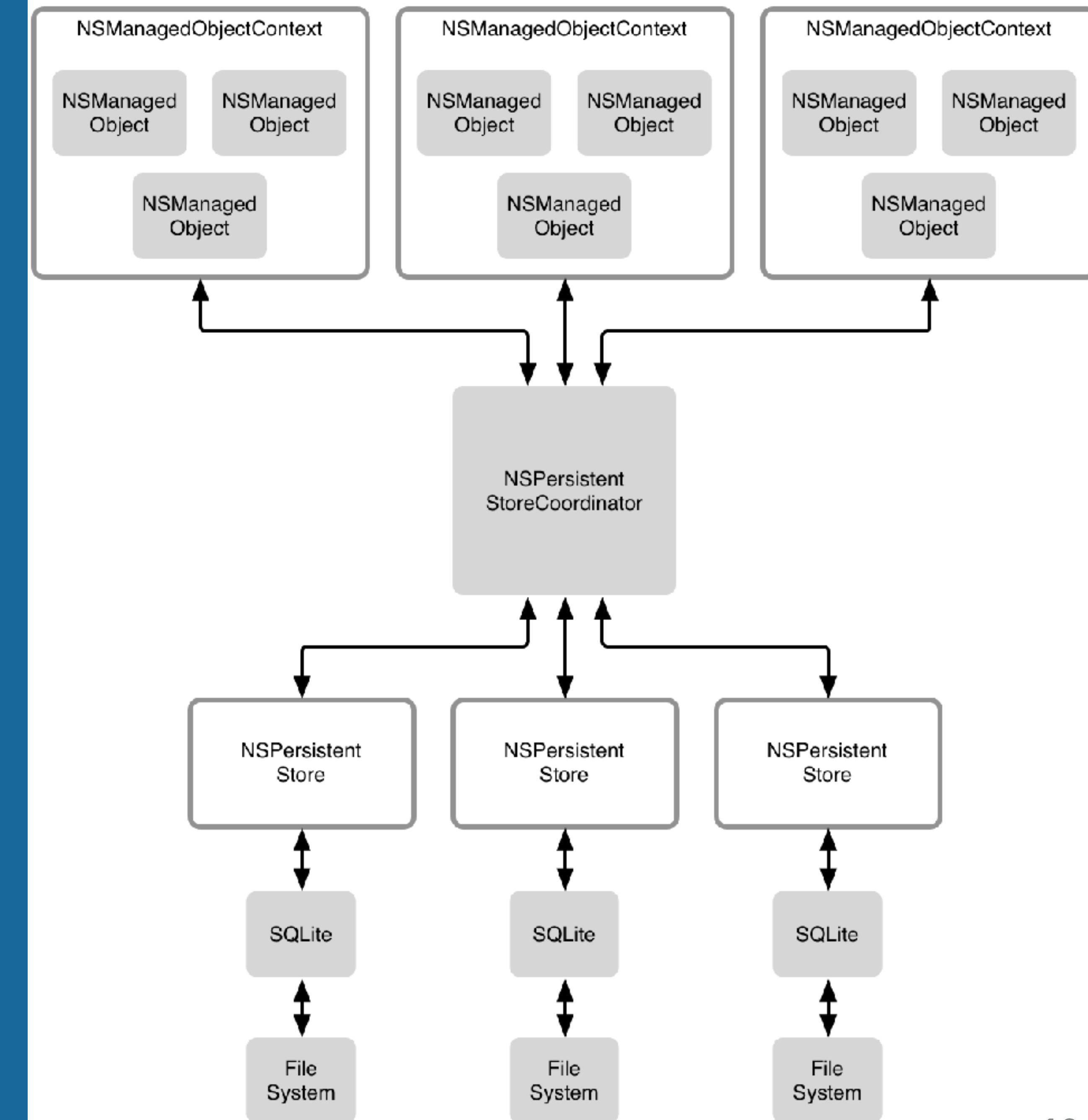
CORE DATA



CORE DATA

SUBTITLE

- Multiple Managed Object Context (MOC) setups
 - MOC only know about their objects
 - Objects only know about their MOC
- Persistent store doesn't care about what its storing



CORE DATA

APPLICATION FLOW

- Setup Core Data Stack

- Object model
- Persistent store
- MOC

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
        The persistent container for the application. This implementation
        creates and returns a container, having loaded the store for the
        application to it. This property is optional since there are legitimate
        error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "_017_CoreDataBasics")
    container.loadPersistentStores(completionHandler: { (storeDescription, error) in
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate. You should not use this function
            // in a shipping application, although it may be useful during development.

            /*
                Typical reasons for an error here include:
                * The parent directory does not exist, cannot be created, or disallows writing.
                * The persistent store is not accessible, due to permissions or data protection when the device is locked.
                * The device is out of space.
                * The store could not be migrated to the current model version.
                Check the error message to determine what the actual problem was.
            */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate. You should not use this function
            // in a shipping application, although it may be useful during development.
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

CORE DATA

APPLICATION FLOW

The screenshot shows the Xcode interface with the Core Data editor open. The project navigation sidebar on the left lists files like AppDelegate.swift, ViewController.swift, Main.storyboard, SearchTableController.swift, TableViewController.swift, Assets.xcassets, LaunchScreen.storyboard, Info.plist, and _017_CoreDataBasics.xcdatamodeld. The main editor area displays the _017_CoreDataBasics.xcdatamodel file. The Person entity is selected in the Entities list. The Attributes section shows three attributes: age (Integer 16), name (String), and timestamp (Date). The Relationships section is currently empty. The Fetched Properties section is also empty.

SearchTableViewController.swift

Running PodcastAPI

_017_CoreDataBasics.xcdatamodel

2017-CoreDataBasics

2017-CoreDataBasics

AppDelegate.swift

ViewController.swift

Main.storyboard

SearchTable...Controller.swift

TableViewController.swift

Assets.xcassets

LaunchScreen.storyboard

Info.plist

_017_CoreDataBasics.xcdatamodeld

PodcastAPI.playground

Products

ENTITIES

E Person

FETCH REQUESTS

CONFIGURATIONS

C Default

Attributes

Attribute	Type
N age	Integer 16
S name	String
D timestamp	Date

Relationships

Relationship	Destination	Inverse

Fetched Properties

Fetched Property	Predicate

CORE DATA

APPLICATION FLOW

- Create an instance of your entity and set the property values

```
var person = Person(context: context!)
person.name = "Homer"
person.age = 42
person.timestamp = Date() as NSDate
```

```
person = Person(context: context!)
person.name = "Marge2"
person.age = 41
person.timestamp = Date() as NSDate
```

CORE DATA

APPLICATION FLOW

- Changes are kept in memory until you explicitly call 'save'
- What happens?
 - MOC figures out what has changed
 - The managed object context then passes these changes on to the persistent store coordinator
 - The persistent store coordinator coordinates with the store (SQLite3 db) to write into the SQL database on disk

```
func saveContext () {  
    if managedObjectContext.hasChanges {  
        do {  
            try managedObjectContext.save()  
        } catch {  
            // Replace this implementation with code to handle the error  
            // appropriately.  
            // abort() causes the application to generate a crash log and  
            // terminate. You should not use this function in a shipping  
            // application, although it may be useful during development.  
            let nserror = error as NSError  
            NSLog("Unresolved error \(nserror), \(nserror.userInfo)")  
            abort()  
        }  
    }  
}
```

CORE DATA

APPLICATION FLOW

- Make a fetch request
- Core Data brings back entity objects
- Fine grain control on the performance

```
// Fetch our person objects
let personFetchRequest: NSFetchedRequest<Person> = Person.fetchRequest()

do {
    let fetchedEntities = try context?.fetch(personFetchRequest)

    for user in fetchedEntities! {
        print("> \(user.name!) #####")
    }
} catch {
    // Do something in response to error condition
}
```

CORE DATA

Why Should You Use Core Data?

There are a number of reasons why it may be appropriate for you to use Core Data. One of the simplest metrics is that, with Core Data, the amount of code you write to support the model layer of your application is typically 50% to 70% smaller as measured by lines of code. This is primarily due to the features listed above—the features Core Data provides are features you don't have to implement yourself. Moreover they're features you don't have to test yourself, and in particular you don't have to optimize yourself.

Core Data has a mature code base whose quality is maintained through unit tests, and is used daily by millions of customers in a wide variety of applications. The framework has been highly optimized over several releases. It takes advantage of information provided in the model and runtime features not typically employed in application-level code. Moreover, in addition to providing excellent security and error-handling, it offers best memory scalability of any competing solution. Put another way: you could spend a long time carefully crafting your own solution optimized for a particular problem domain, and not gain any performance advantage over what Core Data offers for free for any application.

In addition to the benefits of the framework itself, Core Data integrates well with the OS X tool chain. The model design tools allow you to create your schema graphically, quickly and easily. You can use templates in the Instruments application to measure Core Data's performance, and to debug various problems. On OS X desktop, Core Data also integrates with Xcode Interface Builder to allow you to create user interfaces from your model. These aspects help to further shorten integration application design, implementation, and debugging cycles.

Already done
for you

It's a mature
technology

CORE DATA

What Core Data Is Not

NOT A RDB

Having given an overview of what Core Data is and does, and why it may be useful, it is also useful to correct some common misperceptions and state what it is not.

- Core Data is not a relational database or a relational database management system (RDBMS).

Core Data provides an infrastructure for change management and for saving objects to and retrieving them from storage. It can use SQLite as one of its persistent store types. It is not, though, in and of itself a database. (To emphasize this point: you could for example use just an in-memory store in your application. You could use Core Data for change tracking and management, but never actually save any data in a file.)

- Core Data is not a silver bullet.

Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

- Core Data does not rely on Cocoa bindings.

Core Data integrates well with Cocoa bindings and leverages the same technologies—and used together they can significantly reduce the amount of code you have to write—but it is possible to use Core Data without bindings. You can readily create a Core Data application without a user interface (see *Core Data Utility Tutorial*).

"real-world"?

CORE DATA

```
NSManagedObjectModel *model = <#Get a model#>;
NSFetchRequest *requestTemplate = [[NSFetchRequest alloc] init];
NSEntityDescription *publicationEntity =
    [[model entitiesByName] objectForKey:@"Publication"];
[requestTemplate setEntity:publicationEntity];

NSPredicate *predicateTemplate = [NSPredicate predicateWithFormat:
    @"(mainAuthor.firstName like[cd] $FIRST_NAME) AND \
    (mainAuthor.lastName like[cd] $LAST_NAME) AND \
    (publicationDate > $DATE)"];
[requestTemplate setPredicate:predicateTemplate];
[model setFetchRequestTemplate:requestTemplate
    forName:@"PublicationsForAuthorSinceDate"];
```

- Verbose (simple fetch)
- Versioning
- Historical misunderstanding

**CORE DATA STACK PRE-
IOS10**



HISTORICAL PURPOSES ONLY

CORE DATA

APPLICATION FLOW

- This was tremendously simplified in iOS10
- Most criticism of CoreData it probably is outdated

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
        The persistent container for the application. This implementation
        creates and returns a container, having loaded the store for the
        application to it. This property is optional since there are legitimate
        error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "_017_CoreDataBasics")
    container.loadPersistentStores(completionHandler: { (storeDescription, e
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error app
            // fatalError() causes the application to generate a crash log a
            // in a shipping application, although it may be useful during
            /*
                Typical reasons for an error here include:
                * The parent directory does not exist, cannot be created, or di
                * The persistent store is not accessible, due to permissions or
                * The device is out of space.
                * The store could not be migrated to the current model version.
                Check the error message to determine what the actual problem wa
            */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            // Replace this implementation with code to handle the error app
            // fatalError() causes the application to generate a crash log a
            // in a shipping application, although it may be useful during
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

CORE DATA

APPLICATION FLOW

- Setup Core Data Stack

- Object model

- Persistent store

- MOC

Old way had to set up all parts of the stack manually

```
// MARK: - Core Data stack

lazy var applicationDocumentsDirectory: NSURL = {
    // The directory the application uses to store the Core Data store file. This code uses a directory named
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not optional. It is a fatal error for
    let modelURL = NSBundle.mainBundle().URLForResource("_016_CoreDataStack", withExtension: "momd")!
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implementation creates and returns a coordinator.
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading the application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSErrorUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate. You should not use this function in
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }
}

return coordinator
}()

lazy var managedObjectContext: NSManagedObjectContext = {
    // Returns the managed object context for the application (which is already bound to the persistent store).
    let coordinator = self.persistentStoreCoordinator
    var managedObjectContext = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()
```

CORE DATA

```
_managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:[self modelURL]];
_persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                               initWithManagedObjectModel:self.managedObjectModel];

_managedObjectContext = [[NSManagedObjectContext alloc]
                        initWithConcurrencyType:NSMainQueueConcurrencyType];

self.managedObjectContext.persistentStoreCoordinator = self.persistentStoreCoordinator;
[self.managedObjectContext.persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
                      configuration:nil
                            URL:[self storeURL]
                          options:nil
                        error:&error];
```

- Application Flow: Setup Core Data Stack

CORE DATA

```
let entity = NSEntityDescription.entityForName("Standard", inManagedObjectContext: context)
self.init(entity: entity!, insertIntoManagedObjectContext: context)
self.identifier = identifier
self.category = data["category"] as? String
self.stateStandard = data["stateStandard"] as? String
self.subCategory = data["subCategory"] as? String
self.uuid = data["uuid"] as? String
```

Lots of type casting

- Create/Edit entities

CORE DATA

APPLICATION FLOW

```
guard self._fetchedResultsController == nil else {
    return self._fetchedResultsController!
}

let fetchRequest = NSFetchedResultsController(entityName: "Lesson")
let sectionSortDescriptor = NSSortDescriptor(key: "currentIndexSection", ascending: true)
let rowSortDescriptor = NSSortDescriptor(key: "currentIndexRow", ascending: true)
fetchRequest.sortDescriptors = [sectionSortDescriptor, rowSortDescriptor]
if let cp = currentPredicate {
    fetchRequest.predicate = cp
} else {
    fetchRequest.predicate = defaultPredicate
}

let frc = NSFetchedResultsController(fetchRequest: fetchRequest, managedObjectContext: self.context, sectionNameKeyPath: "currentIndexSection", cacheName: nil)
frc.delegate = self
self._fetchedResultsController = frc
return self._fetchedResultsController!
```

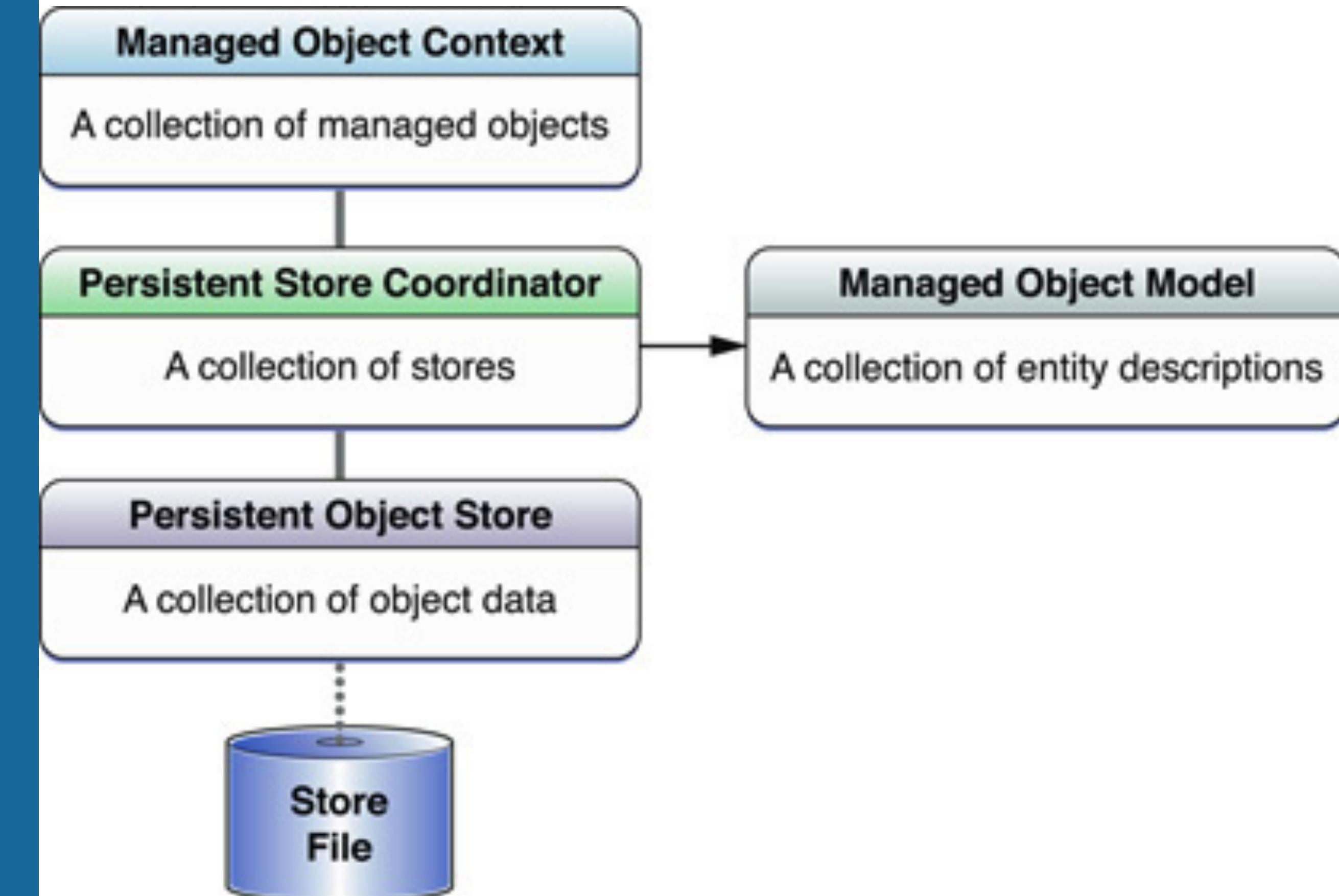
- Fetch

CORE DATA STACK

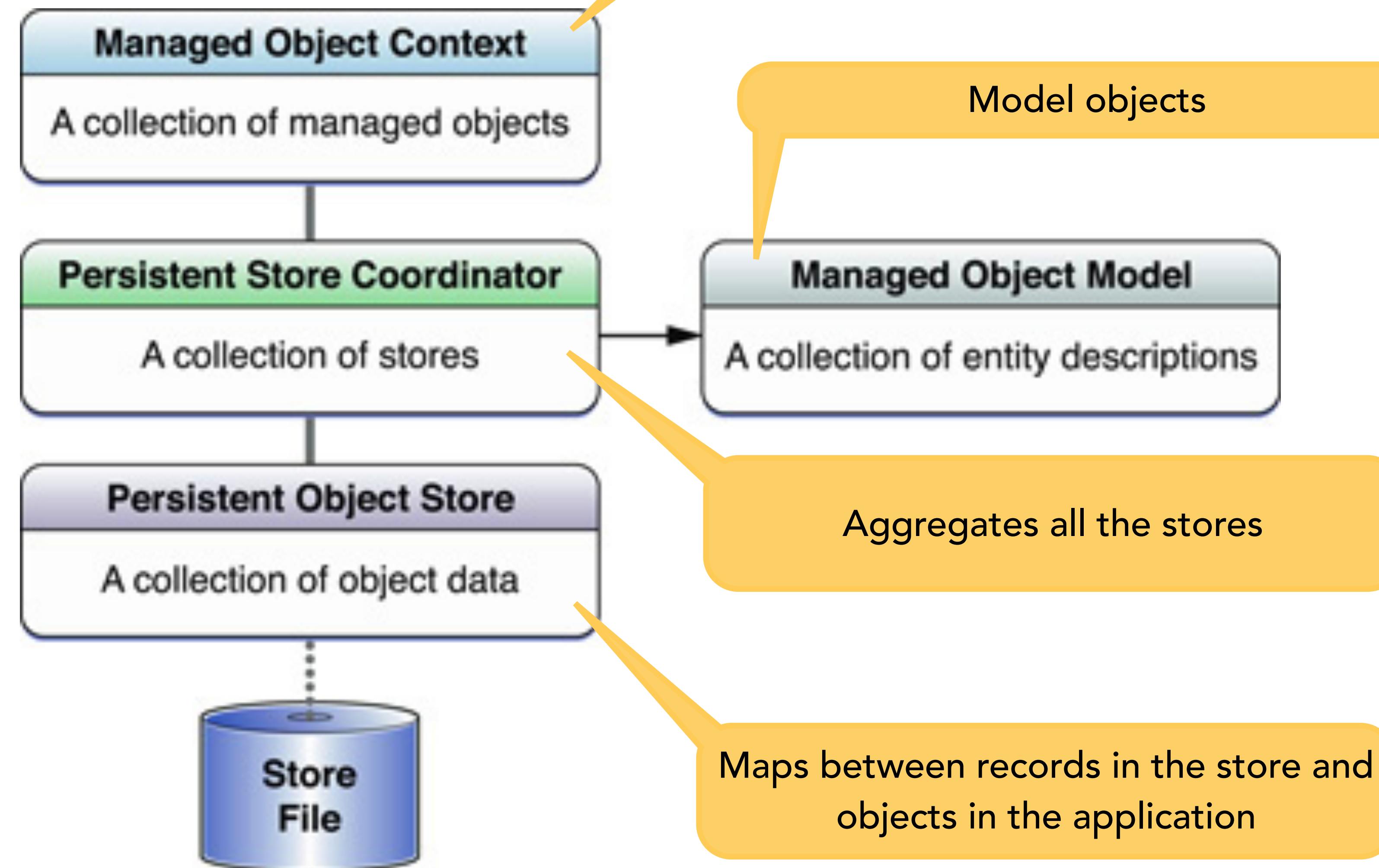
CORE DATA STACK

SUBTITLE

- Core Data stack
 - One or more managed object contexts
 - A single persistent store coordinator
 - Persistent store



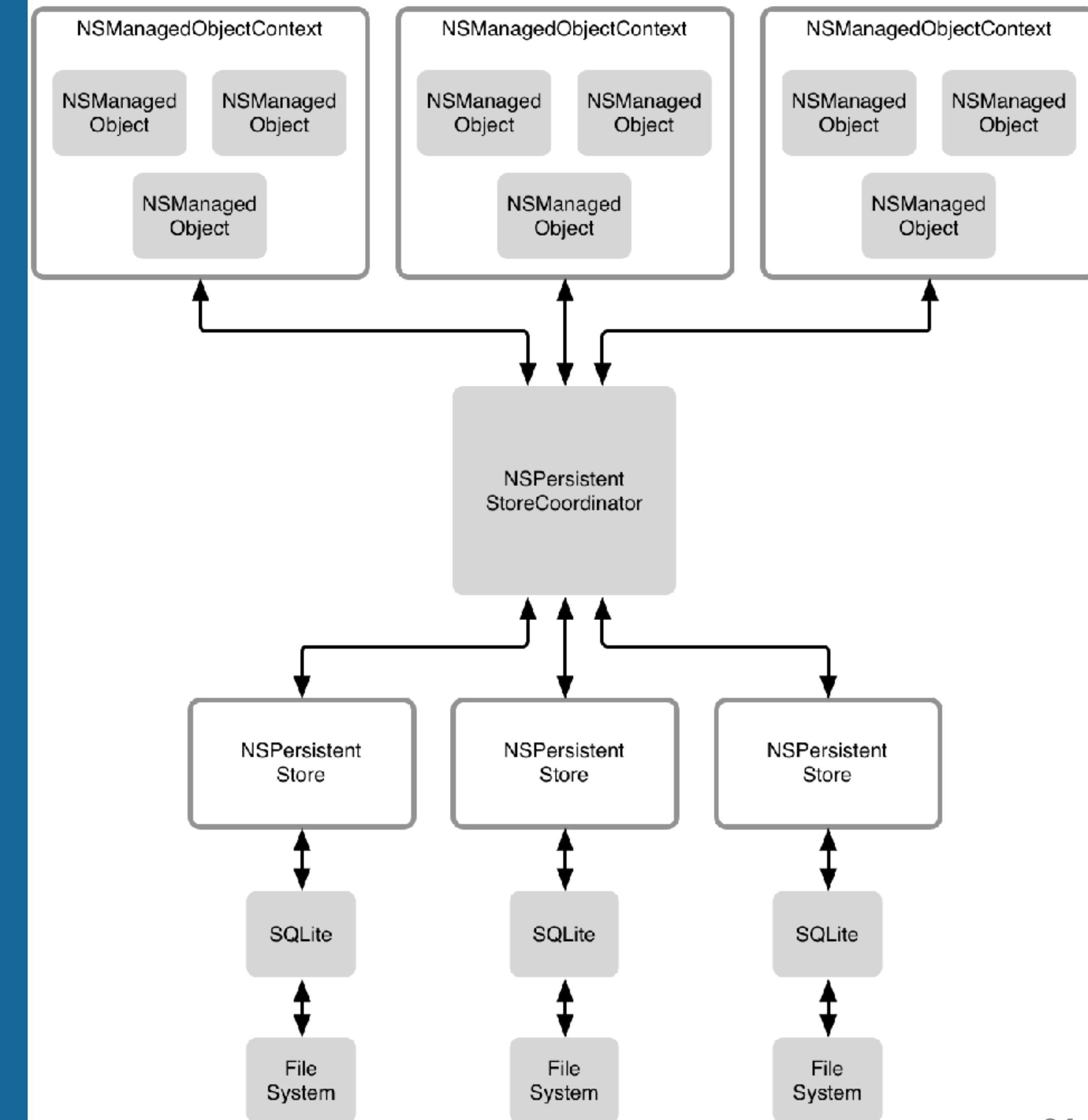
CORE DATA STACK



CORE DATA STACK

SUBTITLE

- Persistent Store Coordinator defines stack
- Can be multiple contexts, stores, etc.



CORE DATA STACK

XCODE TEMPLATE

- Xcode will create code stubs for working the persistent store
- AppDelegate with have a property called `persistentContainer`
- These are fairly complete and will not need to be dramatically altered

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
        The persistent container for the application. This implementation
        creates and returns a container, having loaded the store for the
        application to it. This property is optional since there are legitimate
        error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "_017_CoreDataBasics")
    container.loadPersistentStores(completionHandler: { (storeDescription, error)
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error appropriate
            // fatalError() causes the application to generate a crash log and terminate.
            // This message also appears in the console output.
            /*
                Typical reasons for an error here include:
                * The parent directory does not exist, cannot be created, or disallows
                  changes - check the path.
                * The persistent store is not accessible, due to permissions or data
                  protection settings. Check the error message to determine what the
                  actual problem was.
                */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()
```

CORE DATA STACK

XCODE TEMPLATE

- Provides a function to save the current store

```
// MARK: - Core Data Saving support

func saveContext () {
    print("Saving...")
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
            dump()

        } catch {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate.
            // This behavior may look different on iOS devices, where it will just
            // force quit.
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

CORE DATA STACK

WORKING WITH DATA

- Behind the scenes
 - SQL is generated
 - Results are fetched (lazily)
- Writes to database are only done in memory
 - Need to save them or you will lose data

```
func applicationWillTerminate(_ application: UIApplication) {
    // Called when the application is about to terminate. Save data if
    // appropriate. See also applicationDidEnterBackground:.
    // Saves changes in the application's managed object context before the
    // application terminates.
    self.saveContext()
}
```

CORE DATA STACK

WORKING WITH DATA

- Keys are String (eg. 'thumbnailData','date', 'name')
- The value is whatever is stored
 - Numbers and booleans are NSNumber type objects
 - “To-many” relationships are NSSet
 - Single relationships are NSManagedObjects
 - Binary data is NSData

OLD CORE DATA STACK

CORE DATA STACK

XCODE TEMPLATE
SUBTITLE

- Xcode will create codes stubs for working with `NSManagedObjectContext`
- AppDelegate with have a property called `managedObjectContext`
- These are fairly complete and will not need to be dramatically altered

The screenshot shows the Xcode interface with the title bar "iPhone 6s Plus" and "2016-CoreDataStack: Ready | Today at 12:32 AM". The file "AppDelegate.swift" is open, showing Swift code for initializing a Core Data stack. The code includes lazy var declarations for applicationDocumentsDirectory, managedObjectModel, and persistentStoreCoordinator, along with error handling logic for failed initialization.

```
// MARK: - Core Data stack

lazy var applicationDocumentsDirectory: NSURL = {
    // The directory the application uses to store the Core Data store file
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inContainerDirectory: nil)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not optional.
    let modelURL = NSBundle.mainBundle().URLForResource("016_CoreDataStack", withExtension: "momd")
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implements all interactions with the application's persistent stores.
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataStack.sqlite")
    var failureReason = "There was an error creating or loading the application's persistent store."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's persistent store."
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSErrorUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate.
        // You should not use this function in a shipping application, although it may be useful in a demo application so developers can see what happens if they break it.
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }
}

return coordinator
}()
```

CORE DATA STACK

XCODE TEMPLATE

MODEL FILE

```
lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not optional. It is a fatal error for the application to not have a managed object model.
    let modelURL = NSBundle.mainBundle().URLForResource("_016_CoreDataStack", withExtension: "momd")!
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()
```

STORE FILE

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implementation creates and returns a coordinator, having added a store to it.
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading the application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

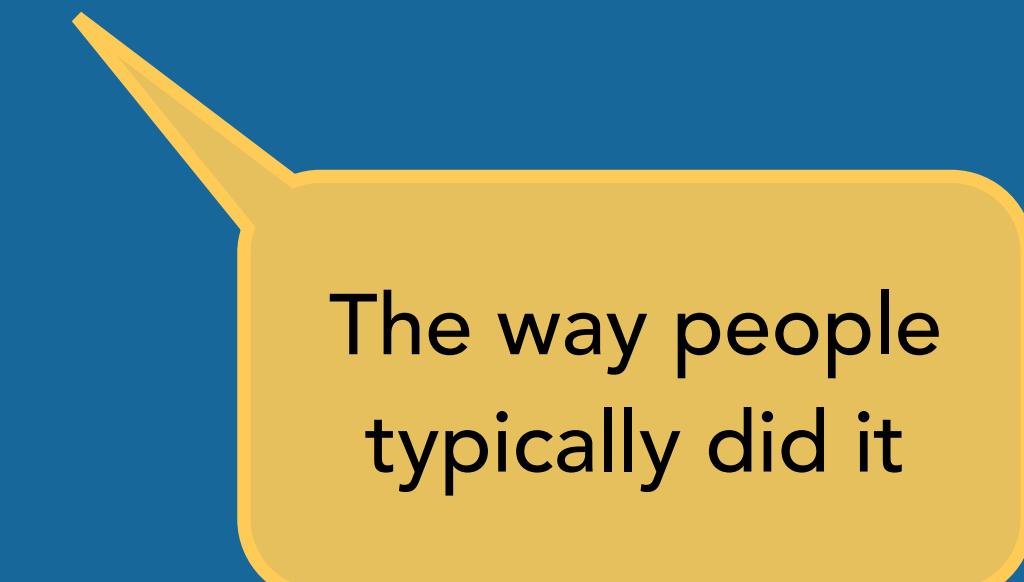
        dict[NSUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate. You should not use this function in a production application,
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }

    return coordinator
}
```

CORE DATA STACK

XCODE TEMPLATE
SUBTITLE

- Best practices (according to Apple)
 - Pass NSManagedObject around via API
 - Do not treat as a global in AppDelegate



```
// MARK: - Core Data stack

lazy var applicationDocumentsDirectory: NSURL = {
    // The directory the application uses to store the Core Data store file
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inContainerDirectory: nil)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not a weak pointer so it can't be deallocated while the application exists.
    let modelURL = NSBundle.mainBundle().URLForResource("_016_CoreDataStack", withExtension: "momd")
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implementation creates and returns a coordinator, thereby creating the store.
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataStack.sqlite")
    var failureReason = "There was an error creating or loading the application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data."
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

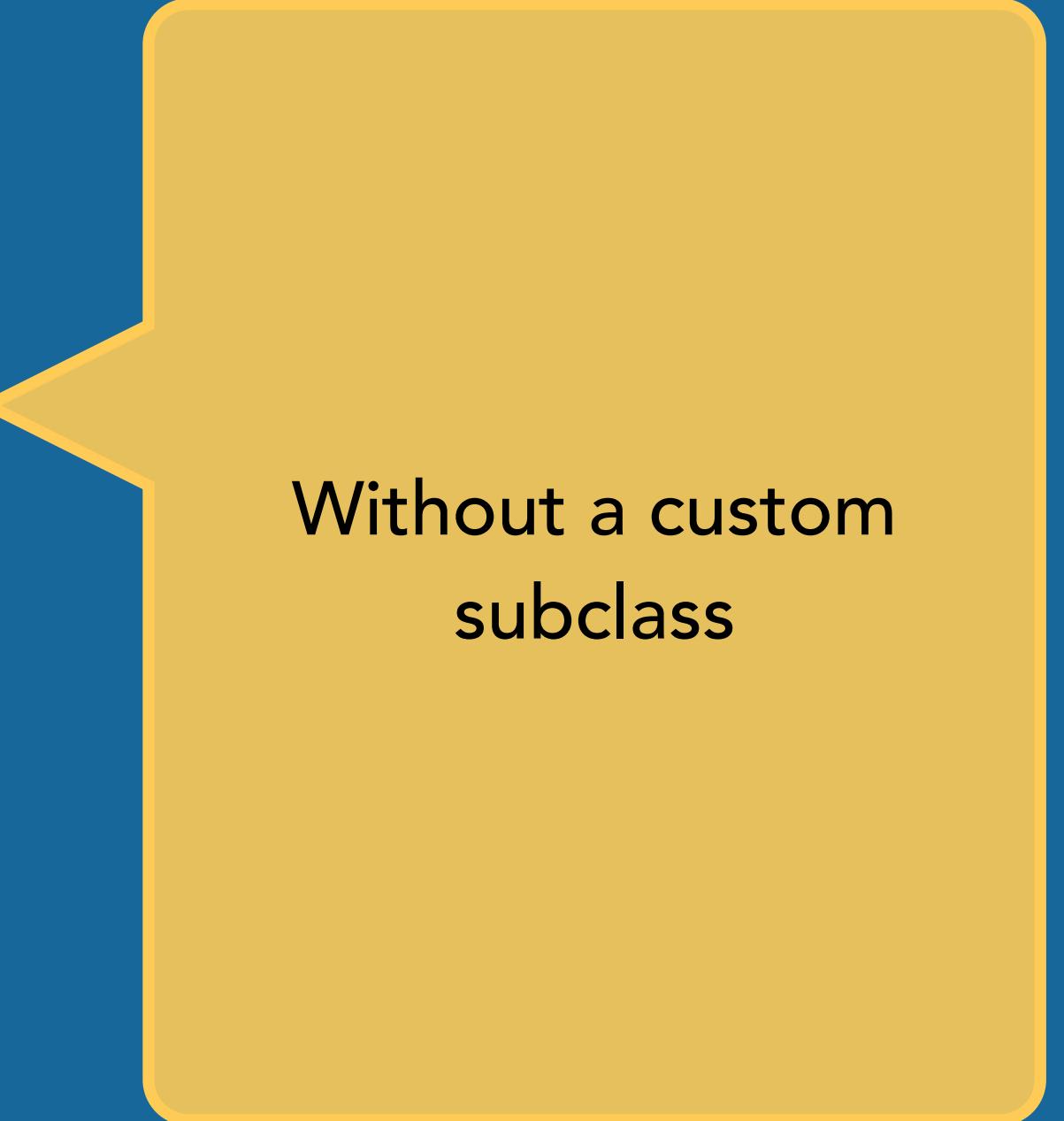
        dict[NSErrorUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate. You should not use this function in a shipping application, although it may be useful in a test environment.
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }
}

return coordinator
}()
```

CORE DATA STACK

WORKING WITH DATA

- Create objects in the database
 - `insertNewObjectForEntityName()`
- Access your data
 - `- (id)valueForKey:(NSString *)key;`
- Set your data
 - `- (void)setValue:(id)value forKey:(NSString *)key;`

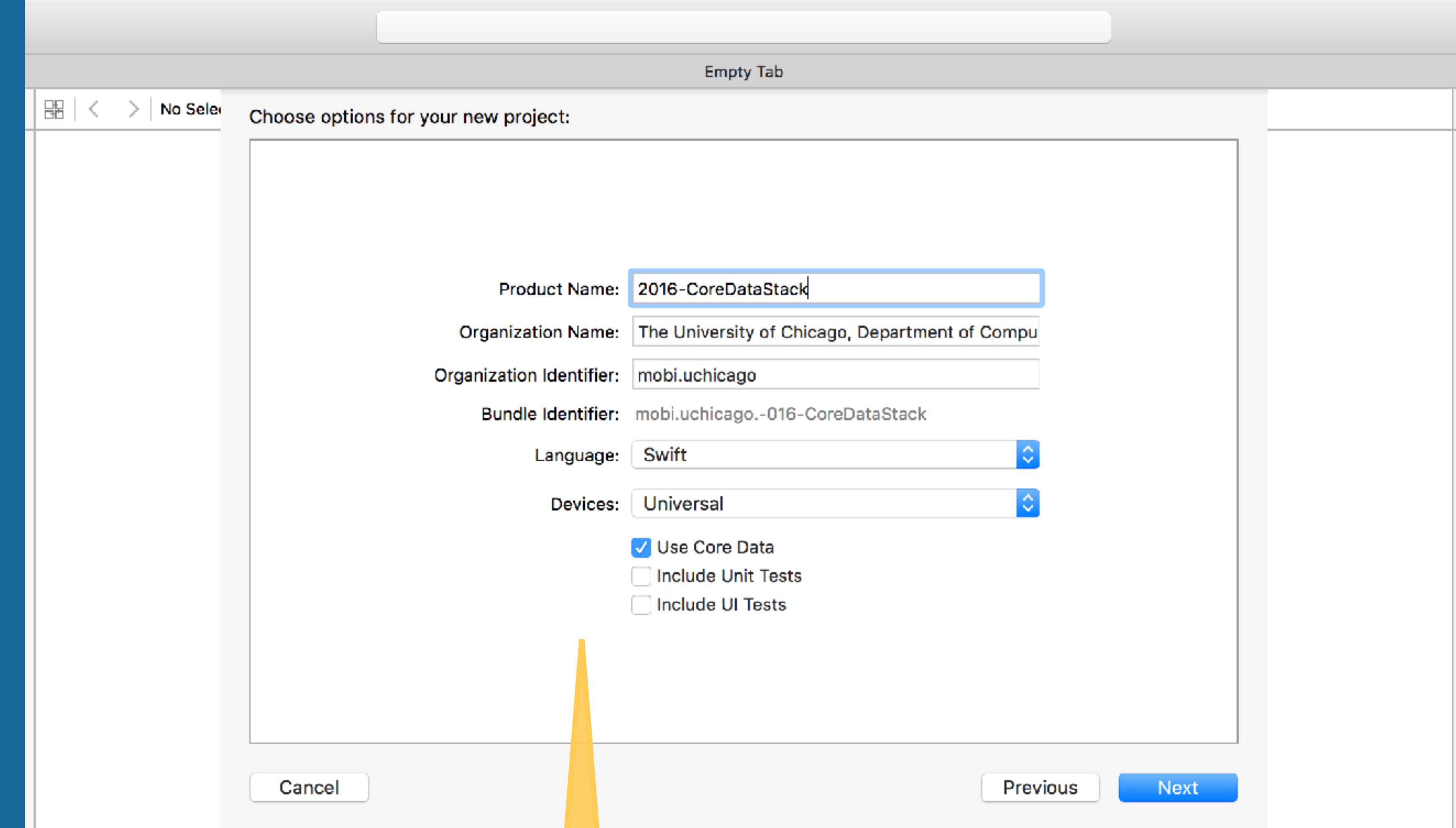


Without a custom
subclass

**CREATE A CORE DATA
MODEL**

CORE DATA MODEL

- Provide a simple data model (.xcdatamodel)
- Xcode will generate the boiler plate methods

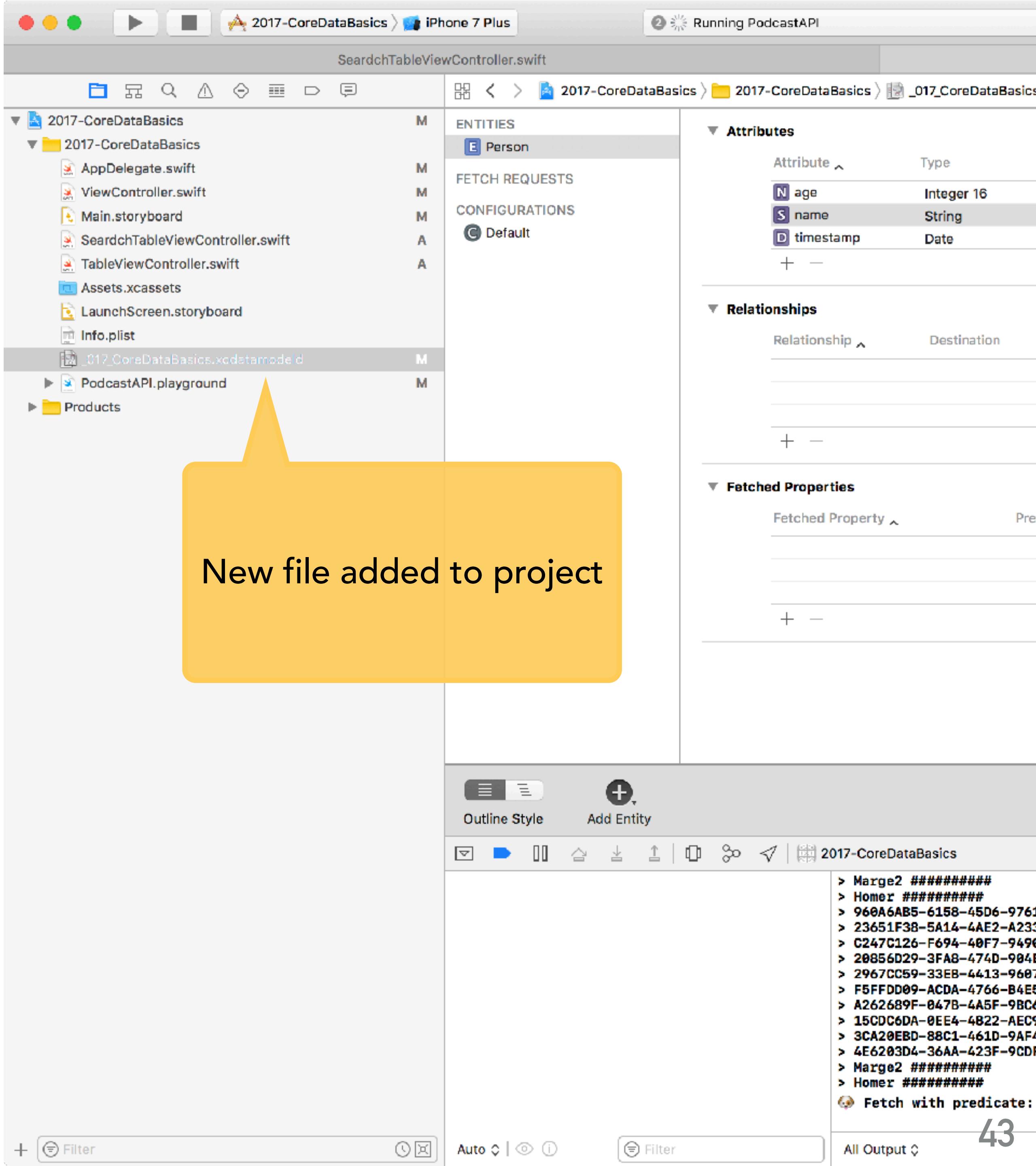


Use Core Date in the is application

CORE DATA

SUBTITLE

- .xcdatamodeld is a directory
 - Stores data model
 - Provides built-in versioning
- Changing your data model will break your app
 - Think about the changes you may make



CORE DATA

- Model is .xcdatamodel file
- Tasks
 - Add entities
 - Define attributes
- Attributes and relationships of an entity equivalent to @properties of a class

▼ **Attributes**

Attribute ^	Type
N age	Integer 16
S name	String
D timestamp	Date

+ -

▼ **Relationships**

Relationship ^	Destination	Inverse
----------------	-------------	---------

+ -

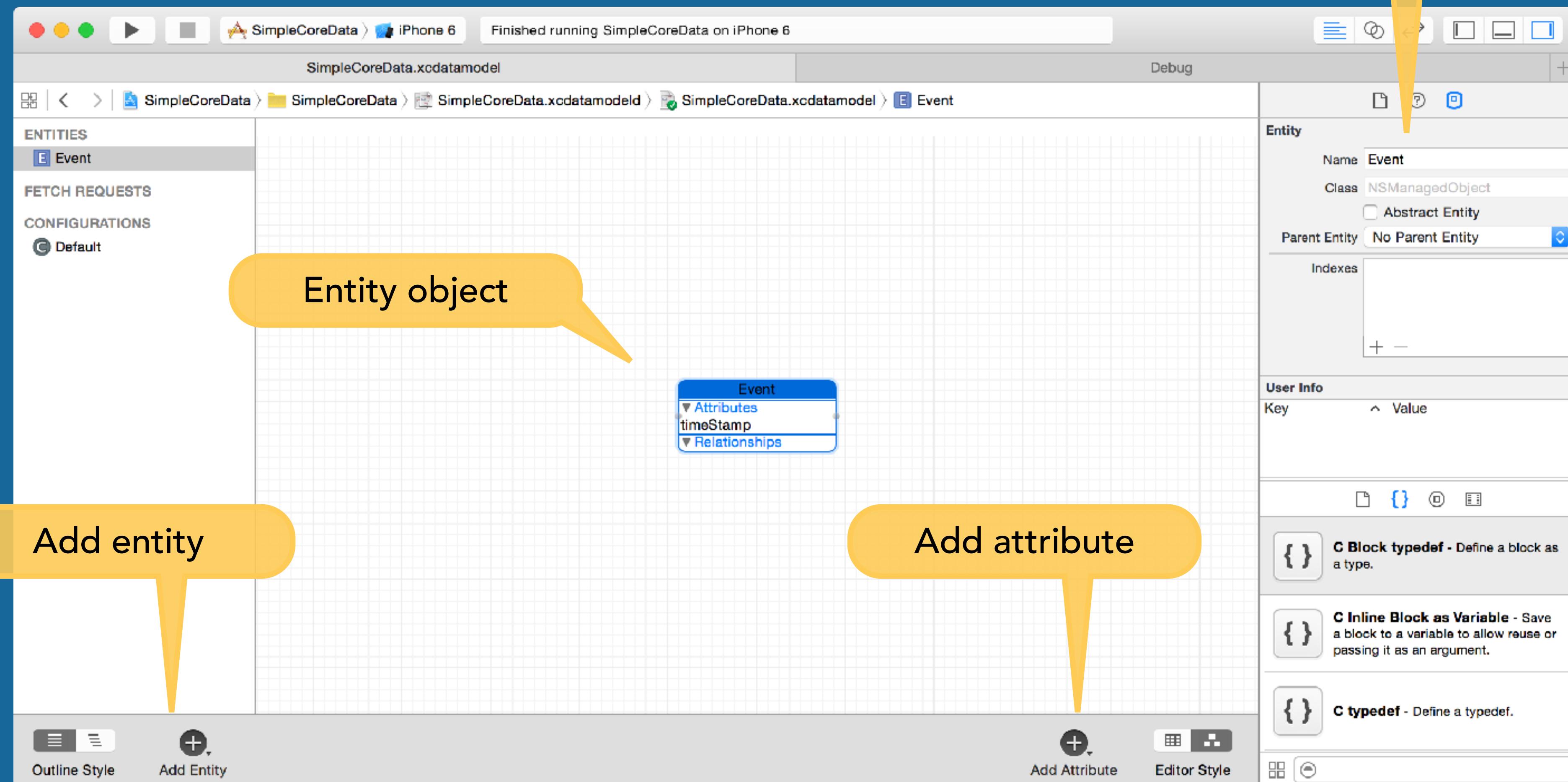
▼ **Fetched Properties**

Fetched Property ^	Predicate
--------------------	-----------

+ -

CORE DATA

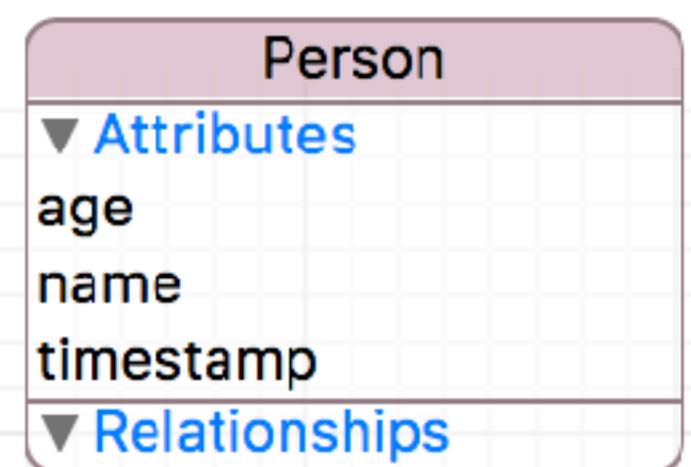
Data model inspector



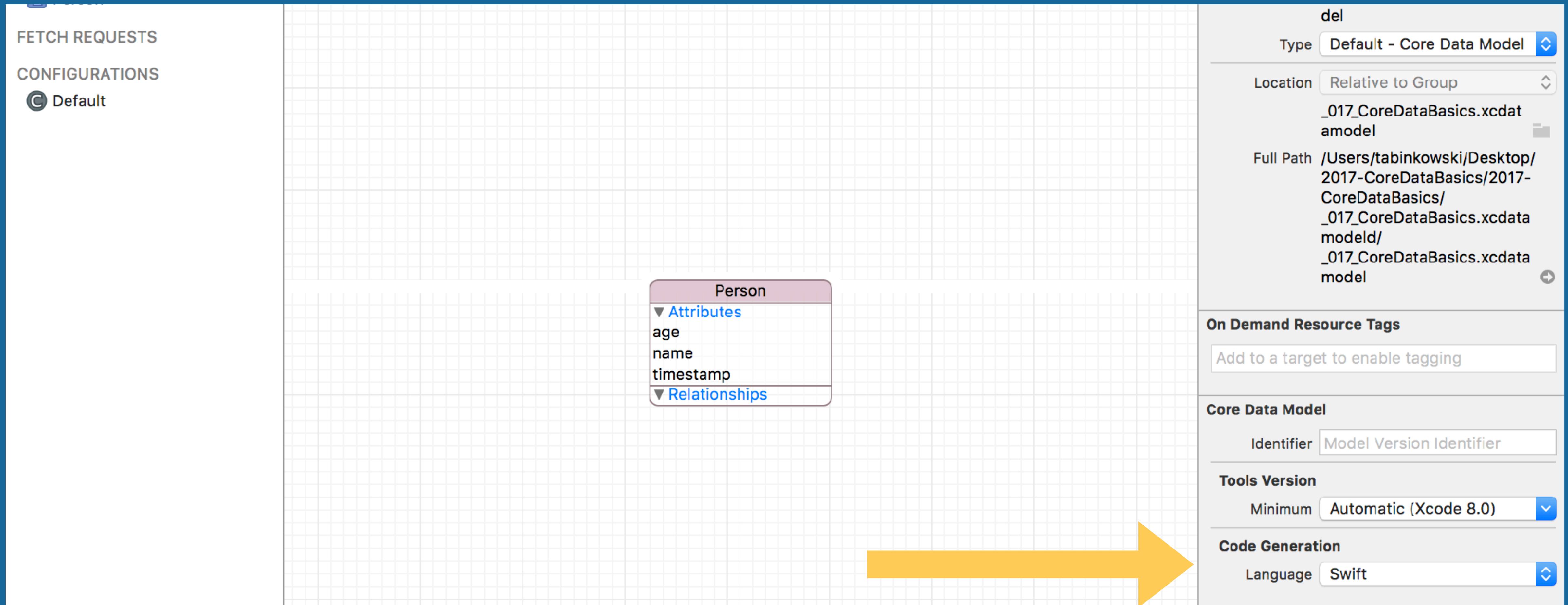
- Entity maps a database table to an object in our code

CORE DATA

- NSManagedObject
 - Subclass of NSObject
 - Access your Core Data model and data
 - The returned object of a Core Data query
 - Key-Value pairs represent the properties of an entity
 - Not thread safe (i.e. do work on main thread)
- Subclass NSManagedObject to access the properties
 - Can also get them using key-value coding with a subclass (this is uncommon)



CORE DATA



- Xcode will generate subclass files of NSManagedObject for you

CORE DATA

The screenshot shows the Xcode Core Data editor for an entity named "Person".

Attribute list:

- age** (Integer 16)
- name** (String)
- timestamp** (Date)

Relationships section is collapsed.

Fetched Properties section is collapsed.

Properties for the "age" attribute on the right side of the editor:

- Name: age
- Properties:
 - Transient:
 - Optional:
 - Indexed:
- Attribute Type: Integer 16
- Validation:
 - No Value
 - No Value
 - 0 (Default: checked)
- Advanced:
 - Index in Spotlight:
 - Store in External Record File:
- Use Scalar Type:

User Info section is collapsed.

Versioning section is collapsed.

- Attributes map to a database column
- Will be able to access via @property in code

CORE DATA

SUBTITLE

- Attributes
 - Type
 - Maps to Foundation object
 - Optional
 - Transient
 - Create a property that exist on object, but have no backing
 - Indexed
 - Create an index in the SQLite backend

Attribute

Name `age`

Properties Transient Optional
 Indexed

Attribute Type `Integer 16`

Validation `No Value` Minimum
`No Value` Maximum
`0` Default Use Scalar Type

Advanced Index in Spotlight
 Store in External Record File

User Info

CORE DATA

SUBTITLE

- Attribute types
 - Foundation types
 - NSData for everything else
 - Binary Data for holding images
 - Transformable for nonstandard object types

The screenshot shows the Xcode interface for managing Core Data entities. The title bar indicates the project is 'SimpleCoreData'. The main area displays the 'Session' entity with its attributes and relationships.

Attributes:

Attribute	Type
B active	Boolean
D endTime	Date
D startTime	Date
S uuid	String

Relationships:

Relationship	Destination
(empty)	(empty)

Fetched Properties:

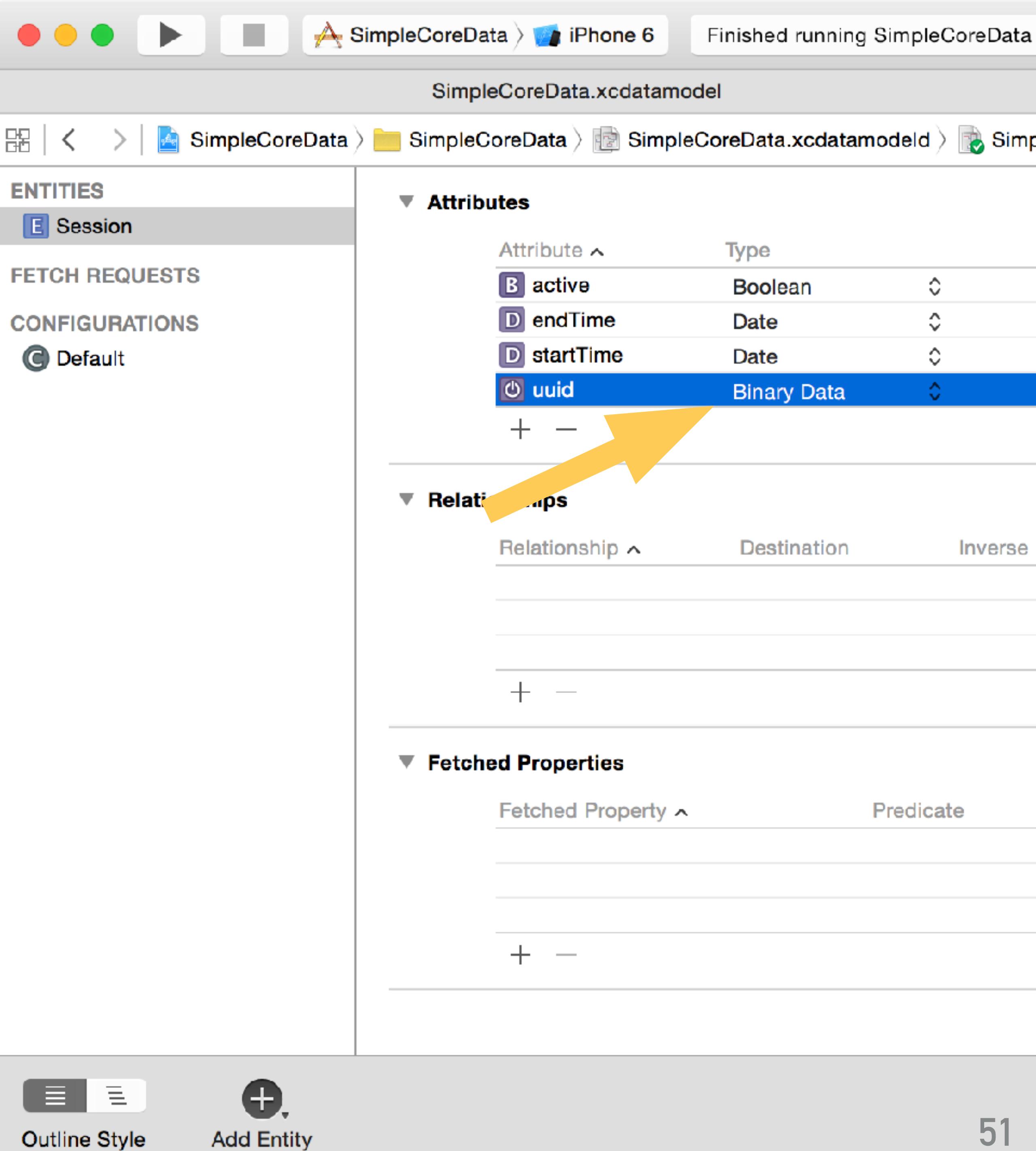
Fetched Property	Predicate
(empty)	(empty)

At the bottom, there are buttons for 'Outline Style' and 'Add Entity'.

CORE DATA

SUBTITLE

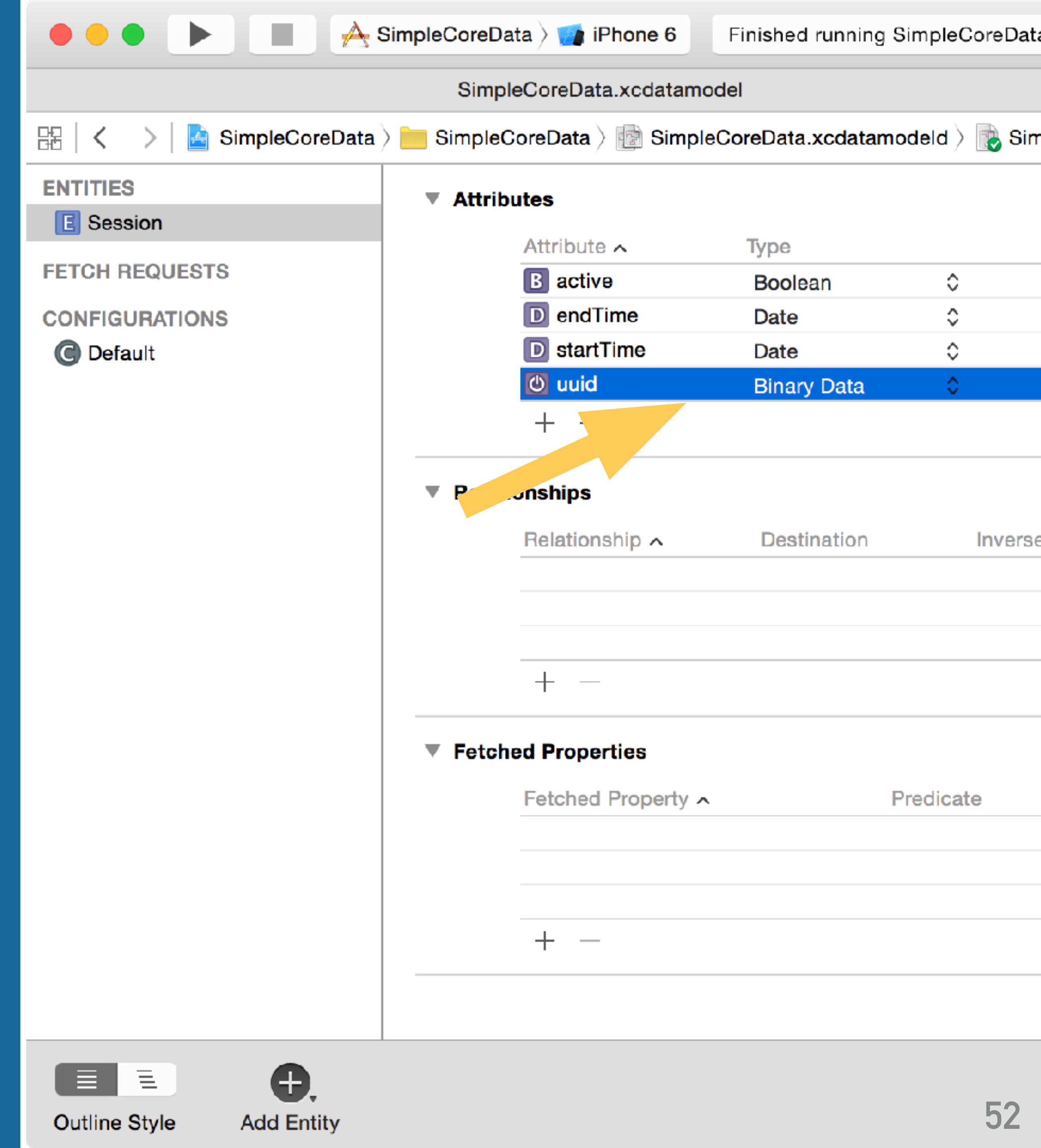
- External Binary Data
 - New in iOS5
- When enabled, Core Data heuristically decides on a per-value basis
 - Save the data directly in the database
 - Store a URL to a separate file



CORE DATA

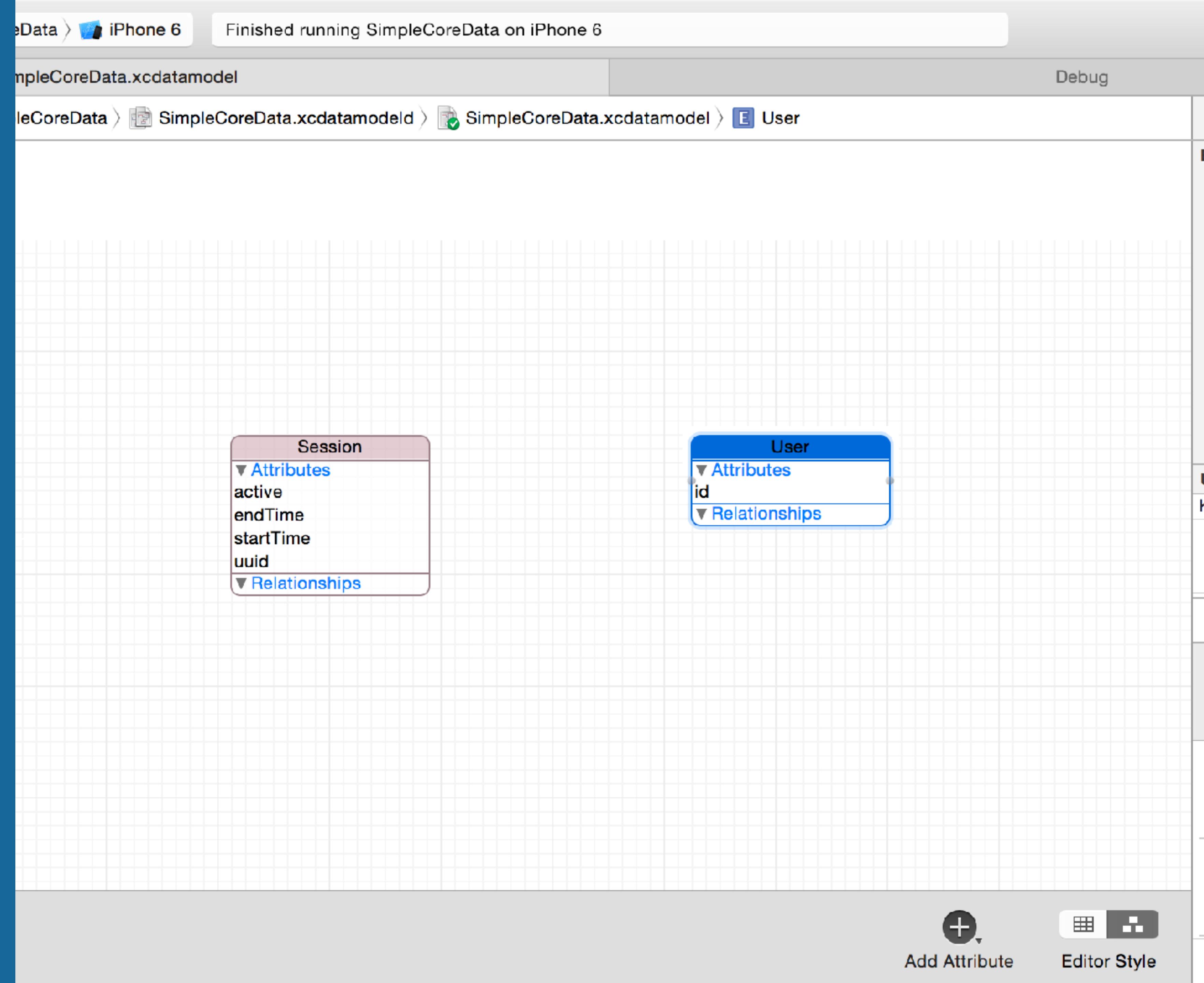
SUBTITLE

- Image storage guidelines
 - < 100 kb store in entity
 - < 1 mb store in a separate entity to avoid performance issues
 - > 1 mb store on disk and reference the path storing nonstandard object types within Core Data



CORE DATA

- Entities can have relationships
- Example: Users and Session
 - A user can have multiple sessions
 - A session has only one user



CORE DATA

The screenshot shows the Xcode Core Data editor with the User entity selected in the sidebar.

Attributes

- id** (String)

Relationships

- sessions** (To Many, Session, user)

Fetched Properties

User Info

Session Entity Configuration

- Name: sessions
- Properties:
 - Transient:
 - Optional:
- Destination: Session
- Inverse: user
- Delete Rule: Nullify
- Type: To Many (highlighted with a red box)
- Arrangement: Ordered
- Count:
 - Unbounded:
 - Minimum:
 - Unbounded:
 - Maximum:
- Advanced:
 - Index in Spotlight:
 - Store in External Record File:

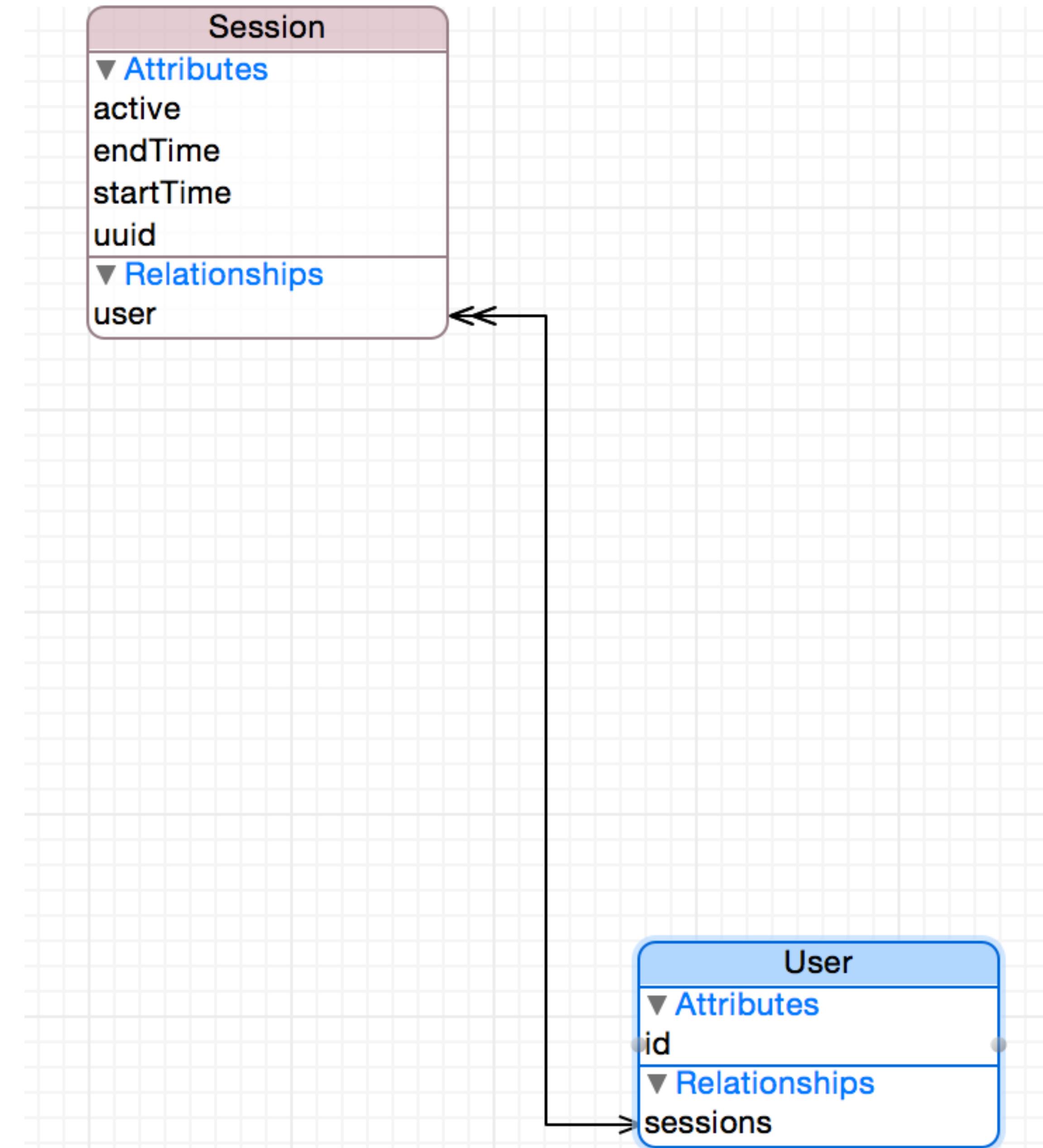
Blocks

- C Block typedef** - Define a block as a type.
- C Inline Block as Variable** - Save a block to a variable to allow reuse or passing it as an argument.

CORE DATA

SUBTITLE

- “To Many” relationship
 - Each user can have many different sessions
- Maps to an `NSSet` of `NSManagedObjects`
 - Remember that sets have no order guarantee



CORE DATA

- Inverse relationship from user to sessions
- Each session can have only 1 user

The screenshot shows the Xcode interface for managing a Core Data model. The title bar indicates "Finished running SimpleCoreData on iPhone 6". The main window is titled "SimpleCoreData.xcdatamodel" and displays the "Session" entity. The "Attributes" section contains four attributes: "active" (Boolean), "endTime" (Date), "startTime" (Date), and "uuid" (String). The "Relationships" section shows a relationship named "user" pointing to the "User" entity, with "sessions" as the inverse relationship. The "Fetched Properties" section is currently empty. The right side of the interface includes tabs for "Entity", "User Info", and "Editor Style". A sidebar on the right lists C-style blocks with their descriptions.

SimpleCoreData.xcdatamodel

SimpleCoreData.xcdatamodel

Session

Entity

Name Session

Class NSManagedObject

Abstract Entity

Parent Entity No Parent Entity

Indexes

User Info

Key Value

C Block typedef - Define a type.

C Inline Block as Variable

a block to a variable to passing it as an argument

C typedef - Define a type

Add Entity

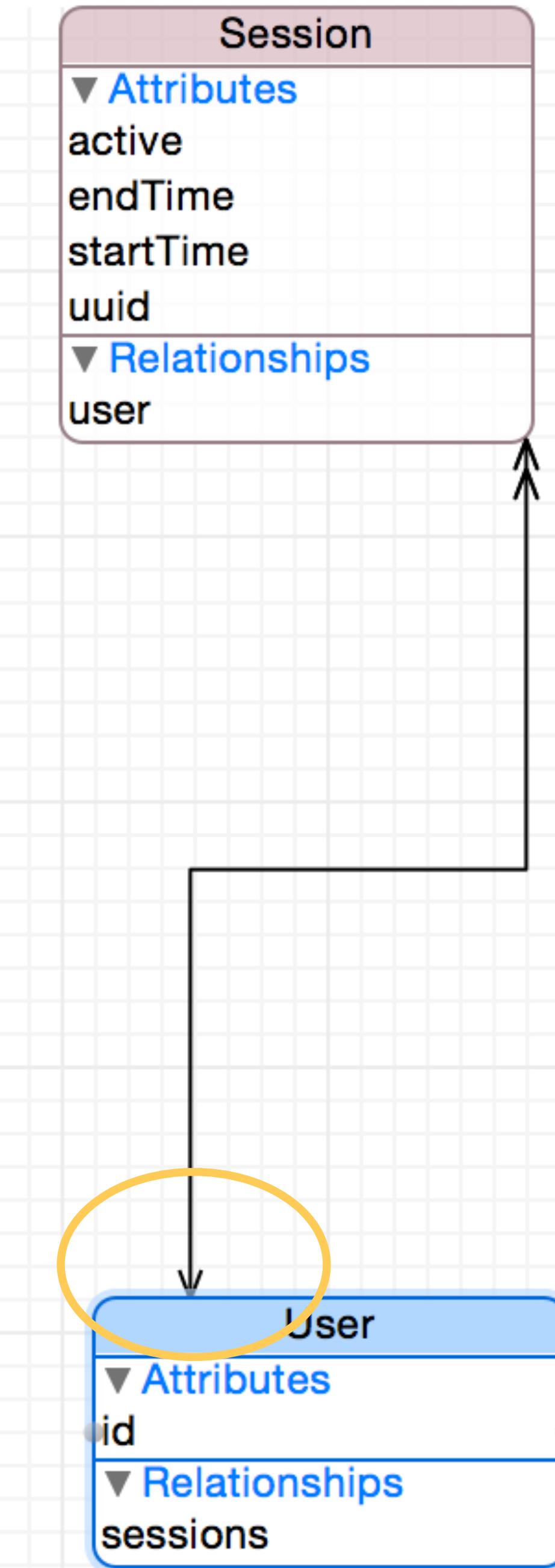
Add Attribute

Editor Style

CORE DATA

SUBTITLE

- Inverse goes only one way
- Maps to a single NSManagedObject



CORE DATA

E Session

E User

FETCH REQUESTS

CONFIGURATIONS

C Default

Attributes

Entity	Attribute ^	Type
Session	B active	Boolean
Session	D endTime	Date
User	S id	String
Session	D startTime	Date
Session	S uuid	String

+ -

Relationships

Entity	Relationship ^	Destination	Inverse
User	M sessions	Session	◊ user ◊
Session	O user	User	◊ sessions ◊

+ -

Fetched Properties

Entity	Fetched Property ^	Predicate
--------	--------------------	-----------

Name

Class

Abstract Entity

Parent Entity

Indexes

+ -

User Info

Key ^ Value

 **C Block typedef** - Define a block as a type.

 **C Inline Block as Variable** - Save a block to a variable to allow reuse or passing it as an argument.

**CREATE
NSMANAGEDOBJECT
SUBCLASS PRIOR TO
IOS10**

NSMANAGED OBJECT SUBCLASS

- Simplify access of object attributes
 - No type checking
- Create a subclass of NSManagedObject
 - @property for each attribute
- Xcode will auto-generate them for you

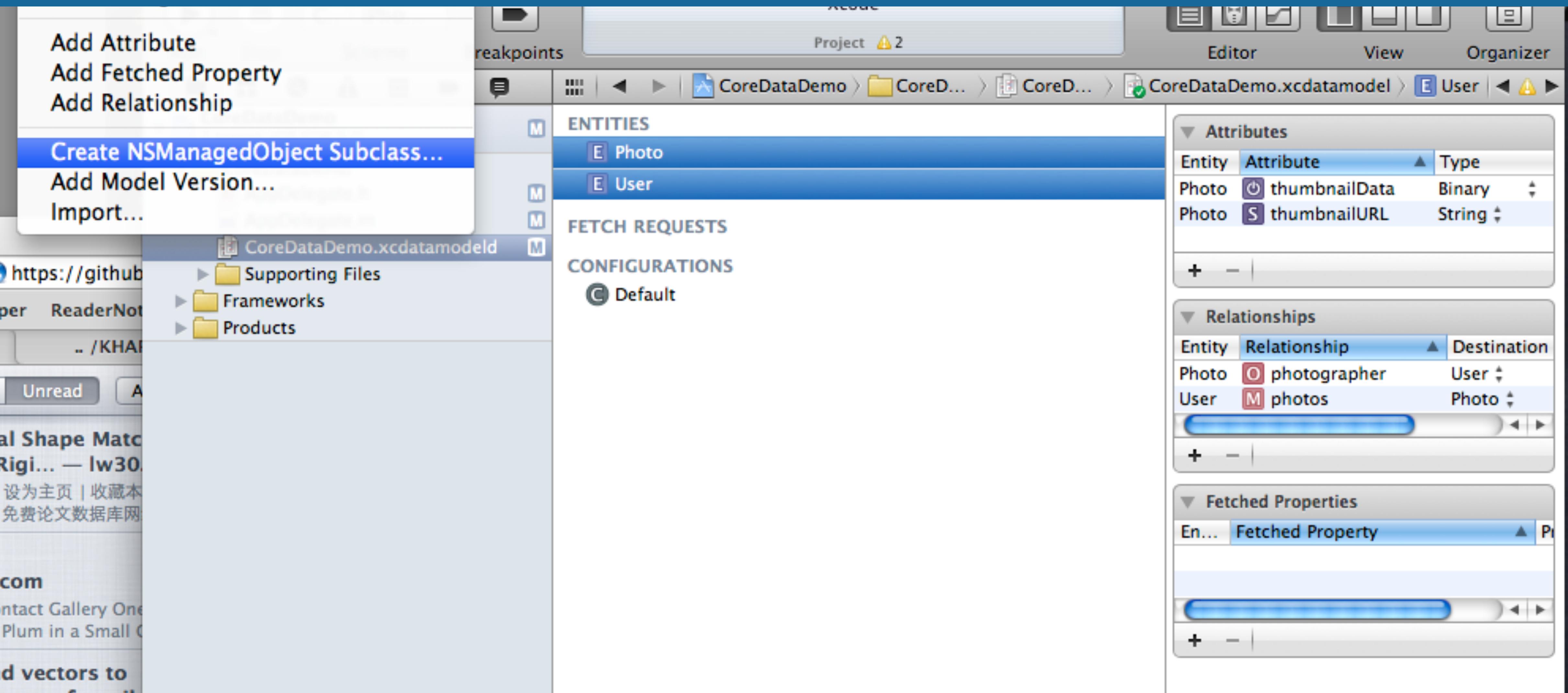
The screenshot shows the Xcode Data Model Editor with the following details:

- Entities:** Session, User
- Attributes:**

Entity	Attribute	Type
Session	S endTime	String
User	S id	String
Session	B isActive	Boolean
Session	D startTime	Date
Session	U uuid	Undefined
- Relationships:**

Entity	Relationship	Destination	Inverse
User	M sessions	Session	user
Session	O user	User	sessions
- Fetched Properties:** None listed.

NSMANAGED OBJECT SUBCLASS



NSMANAGED OBJECT SUBCLASS

Entities

- E Session
- E User

Fetch Requests

Configurations

C Default

Select the data models with entities you would like to manage

Select | Data Model

SimpleCoreData

Cancel Previous Next

Values

ManagedObject

: Entity

Entity

f - Define a block as

The screenshot shows the Xcode interface for creating an NSManagedObject subclass. On the left, there's a sidebar with sections for Entities, Fetch Requests, Configurations, and a selected item 'Default'. The main area is titled 'Select the data models with entities you would like to manage' and contains a 'Select | Data Model' dropdown. The 'SimpleCoreData' option is selected with a checkmark. At the bottom are 'Cancel', 'Previous', and 'Next' buttons. To the right, there are several partially visible text fields and dropdown menus, likely for defining entity properties.

NSMANAGED OBJECT SUBCLASS

ENTITIES
E Session
E User

FETCH REQUESTS

CONFIGURATIONS

C Default

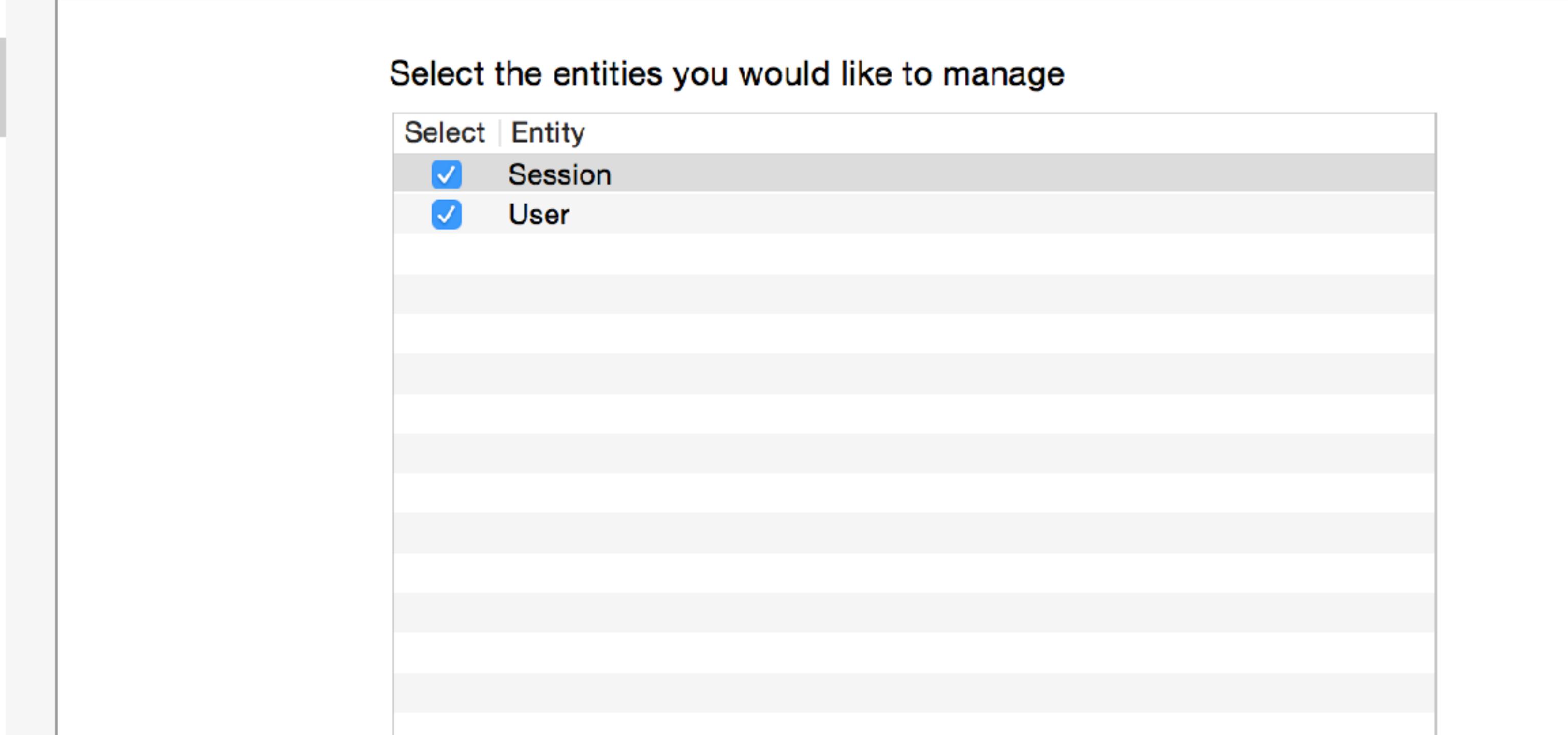
Select the entities you would like to manage

Select Entity

Session

User

Cancel Next



Multiple Values
NSManagedObject

Abstract Entity

o Parent Entity

Value

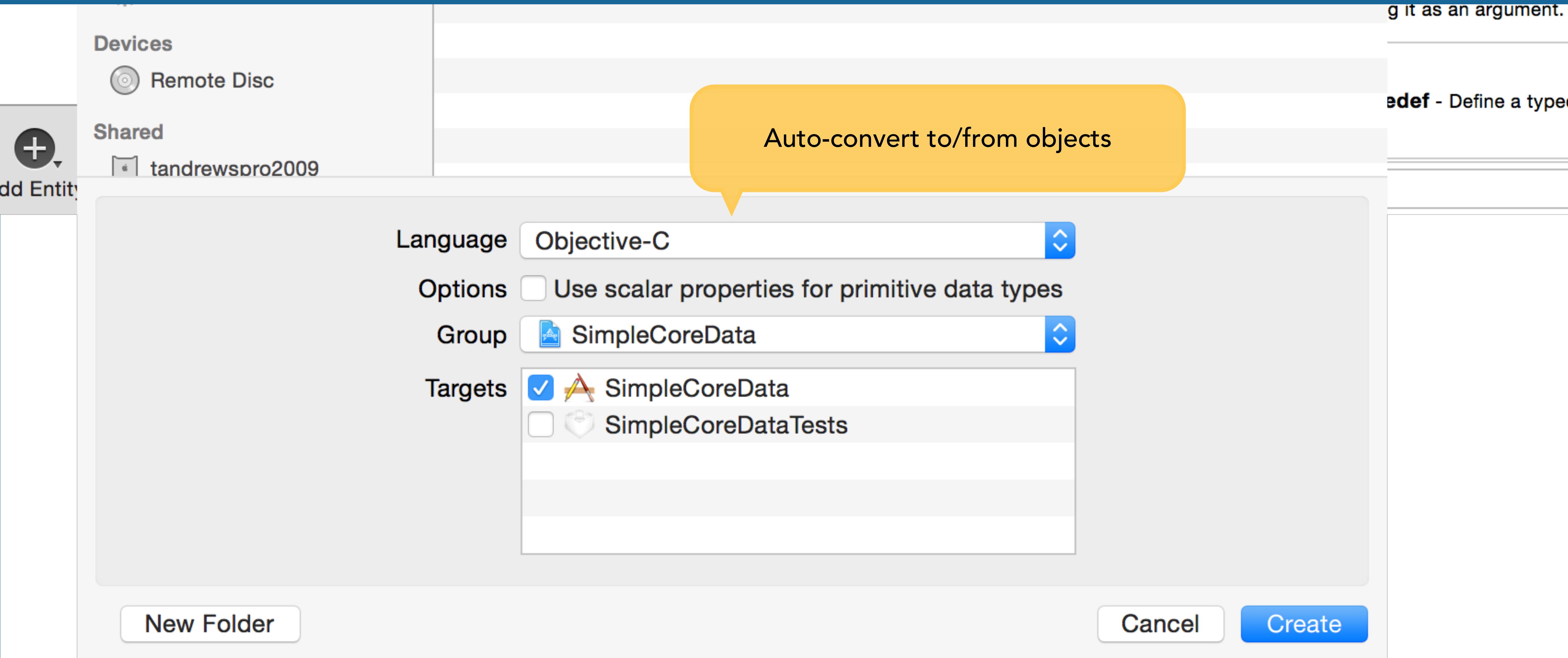
{ } ◻ ◻

< **typedef** - Define a block as



C Inline Block as Variable - Save a block to a variable to allow reuse or passing it as an argument.

NSMANAGED OBJECT SUBCLASS



NSMANAGED OBJECT SUBCLASS

```
User+CoreDataProperties.swift  
User.swift  
2016-CoreDataStack  
AppDelegate.swift  
ViewController.swift  
Main.storyboard  
Assets.xcassets  
LaunchScreen.storyboard  
Info.plist  
_016_CoreDataStack.xcdatamodeld  
Products
```

```
2 // User.swift  
3 // 2016-CoreDataStack  
4 //  
5 // Created by T. Andrew Binkowski on 4/11/16.  
6 // Copyright © 2016 The University of Chicago, Department of Computer Science. All rights reserved.  
7 //  
8  
9 import Foundation  
10 import CoreData  
11  
12  
13 class User: NSManagedObject {  
14  
15 // Insert code here to add functionality to your managed object subclass  
16  
17 }  
18
```

ADD METHODS FOR USER CLASS (DON'T HAVE TO BE RELATED TO CORE DATA)

NSMANAGED OBJECT SUBCLASS

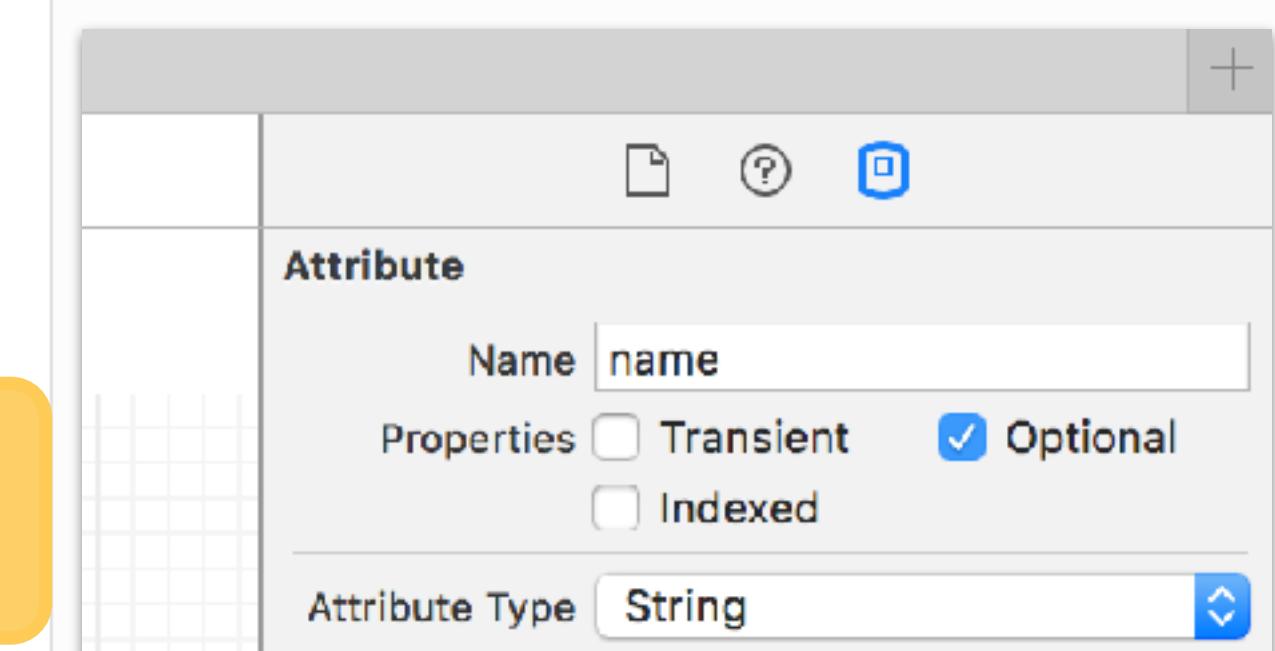
```
User+CoreDataProperties.swift  
User.swift  
2016-CoreDataStack  
AppDelegate.swift  
ViewController.swift  
Main.storyboard  
Assets.xcassets  
LaunchScreen.storyboard  
Info.plist  
_016_CoreDataStack.xcdatamodeld  
Products
```

```
2 // User+CoreDataProperties.swift  
3 // 2016-CoreDataStack  
4 //  
5 // Created by T. Andrew Binkowski on 4/11/16.  
6 // Copyright © 2016 The University of Chicago, Department of Computer Science. All rights reserved.  
7 //  
8 // Choose "Create NSManagedObject Subclass..." from the Core Data editor menu  
9 // to delete and recreate this implementation file for your updated model.  
10 //  
11  
12 import Foundation  
13 import CoreData  
14  
15 extension User {  
16     @NSManaged var name: String?  
17     @NSManaged var sessions: NSSet?  
18 }  
19  
20  
21
```

Extension on the User class

NSSet for relationships

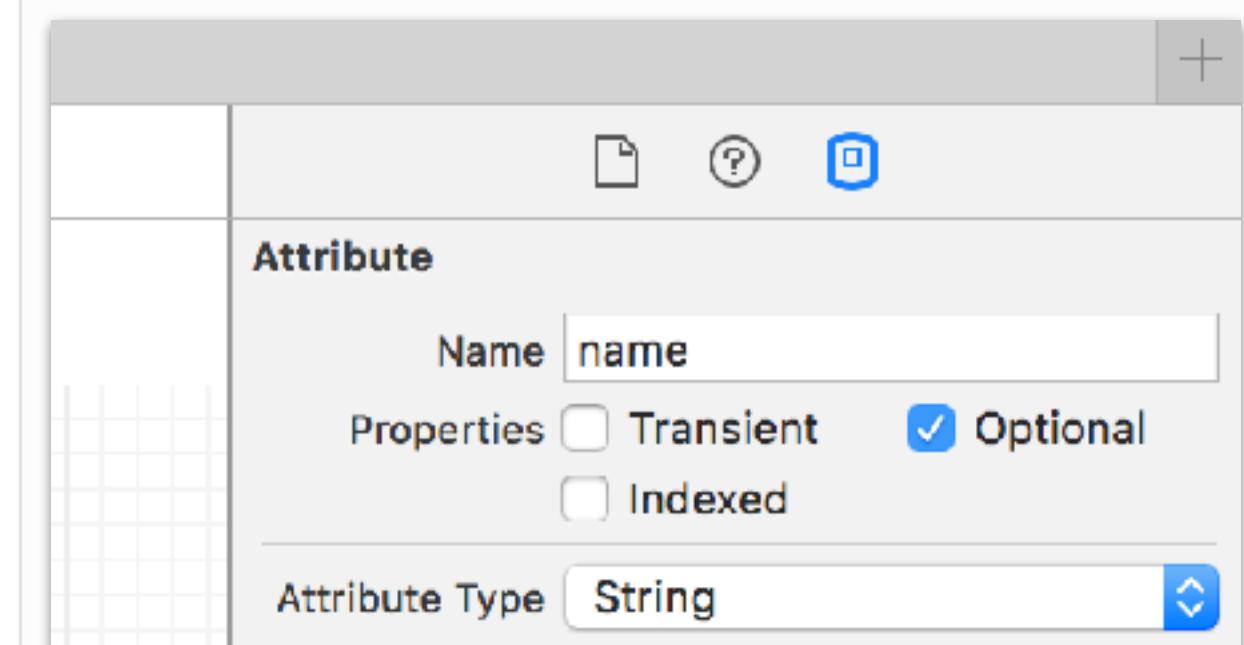
Optionals



NSMANAGED OBJECT SUBCLASS

```
2 // User+CoreDataProperties.swift
3 // 2016-CoreDataStack
4 //
5 // Created by T. Andrew Binkowski on 4/11/16.
6 // Copyright © 2016 The University of Chicago, Department of Computer Science. All rights reserved.
7 //
8 // Choose "Create NSManagedObject Subclass..." from the Core Data editor menu
9 // to delete and recreate this implementation file for your updated model.
10 //
11
12 import Foundation
13 import CoreData
14
15 extension User {
16
17     @NSManaged var name: String?
18     @NSManaged var sessions: NSSet?
19
20 }
21
```

Only generate this
one time



NSMANAGED OBJECT SUBCLASS

```
17 13:04:52.939 MyFavoriteApps[2503:fb03] Unresolved error Error Domain=NSCocoaErrorDomain Code=134100 "The operation couldn't be completed. (Cocoa error 134100)" UserInfo={metadata=<CFBasicHash 0x6b67b20 [0x173fb48]>{type = immutable dict, count = 7, bytes = 0x6b67b20, length = 7}, <CFString 0x6b67ee0 [0x173fb48]>{contents = "NSStoreModelVersionIdentifiers"} = <CFArray 0x6b68340 [0x173fb48]>{type = immutable, count = 1, values = 0x173acd8 [0x173fb48]}, <CFString 0x173acd8 [0x173fb48]>{contents = ""}, <CFString 0x6b67f10 [0x173fb48]>{contents = "NSPersistenceFrameworkVersion"} = <CFNumber 0x6b67dd0 [0x173fb48]>{value = +386, type = kCFNumberSInt64Type}, <CFString 0x6b682e0 [0x173fb48]>{contents = "NSStoreModelVersionHashes"} = <CFBasicHash 0x6b683c0 [0x173fb48]>{type = immutable dict, count = 1, bytes = 0x6b683c0, length = 1}, <CFString 0x6b68360 [0x173fb48]>{contents = "Event"} = <CFData 0x6b68370 [0x173fb48]>{length = 32, cString = 0x16d30e7f32c2cc809958addle7 ... 846e97d7af01cc79}, <CFString 0x10f8ad8 [0x173fb48]>{contents = "NSStoreUUID"} = <CFString 0x6b680c0 [0x173fb48]>{contents = "838007D0-B9E9-4CE6-9379-4394F44B051D"}, <CFString 0x10f8978 [0x173fb48]>{contents = "NSStoreType"} = <CFString 0x10f8988 [0x173fb48]>{contents = "SQLite"}, <CFString 0x6b67d90 [0x173fb48]>{contents = "_NSAutoVacuumLevel"} = <CFString 0x6b68420 [0x173fb48]>{value = 2}, <CFString 0x6b68310 [0x173fb48]>{contents = "NSStoreModelVersionHashesVersion"} = <CFNumber 0x6b2290 [0x173fb48]>{value = +3, type = kCFNumberSInt32Type} }

=The model used to open the store is incompatible with the one used to create the store}, {
data = {
    NSPersistenceFrameworkVersion = 386;
    NSStoreModelVersionHashes = {
        Event = <5431c046 d30e7f32 c2cc8099 58addle7 579ad104 a3aa8fc4 846e97d7 af01cc79>;
    };
    NSStoreModelVersionHashesVersion = 3;
    NSStoreModelVersionIdentifiers = (
        ""
    );
    NSStoreType = SQLite;
    NSStoreUUID = "838007D0-B9E9-4CE6-9379-4394F44B051D";
    '_NSAutoVacuumLevel' = 2;
}

on = "The model used to open the store is incompatible with the one used to create the store";
```

NEED TO REGENERATE MANAGED
OBJECTS AFTER CHANGE

NSMANAGEDOBJECT SUBCLASS IN IOS10+

NSMANAGEDOBJECT SUBCLASS IN IOS10

- Xcode 8+
`codegen` will
create subclasses
for you at compile
time
 - You don't ever
see them

The screenshot shows the Xcode Model Editor with the following configuration:

- Parent Entity:** No Parent Entity
- Class:**
 - Name:** Person
 - Module:** Global namespace
 - Codegen:** Class Definition (highlighted)
- Indexes:** No Content
- Constraints:** No Content
- User Info:** Key Value

NSMANAGEDOBJECT SUBCLASS IN IOS10

- Thats it 😊



apple WATCH APPLICATION DEVELOPMENT

MPCS 51032 • SPRING 2020 • SESSION 5