THE UNIVERSITY OF CHICAGO

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?

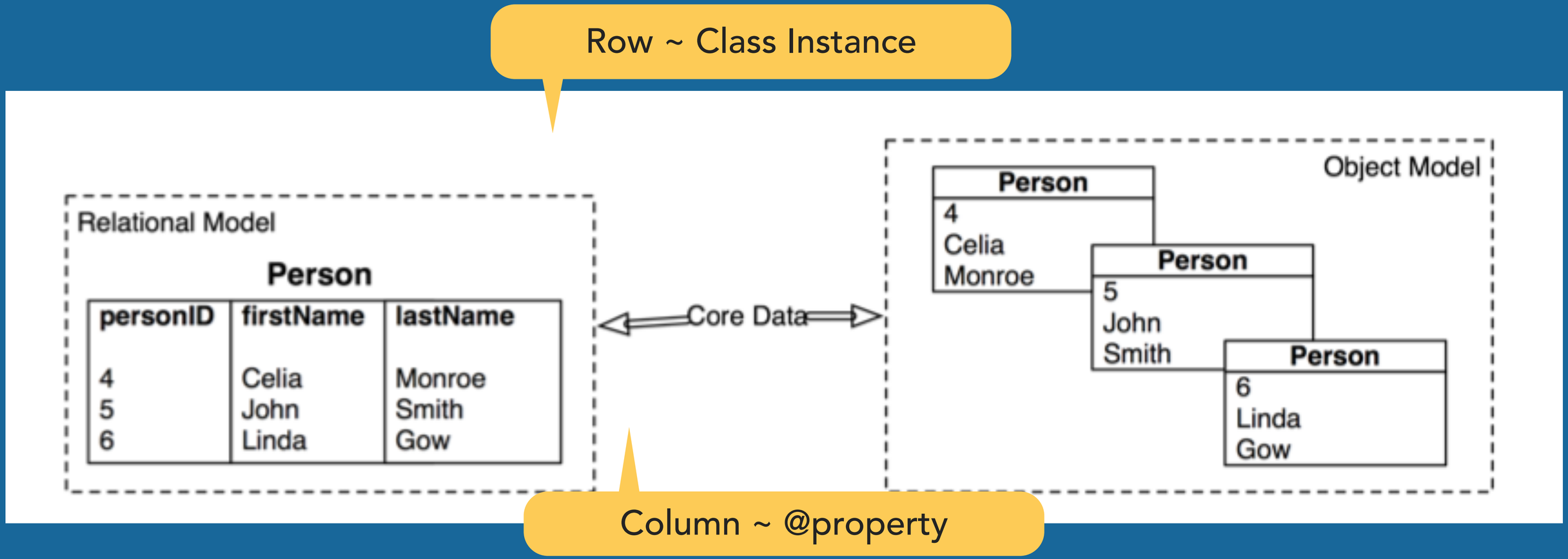
- An object-oriented API that conforms to MVC paradigm
- API for constructing detailed queries
- Access the data using properties on the objects

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

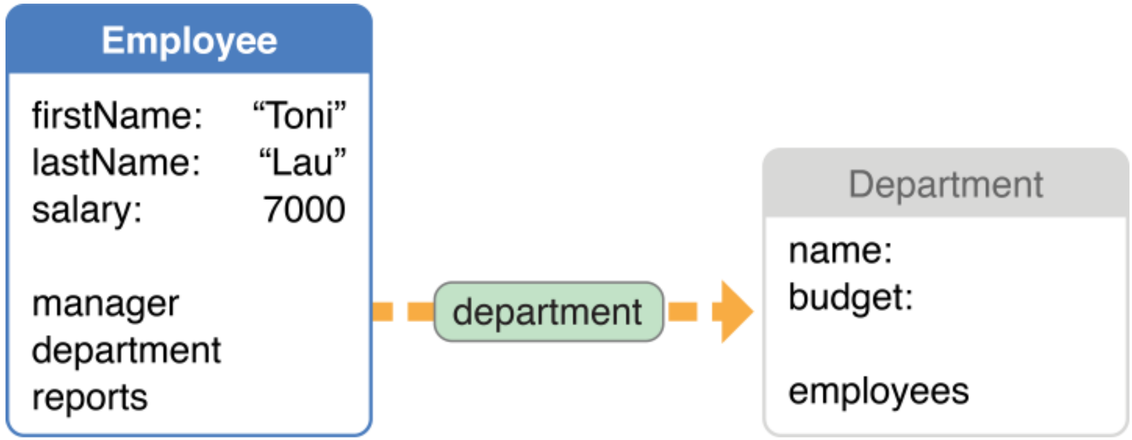
WHAT YOU SHOULD KNOW ABOUT CORE DATA

- 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

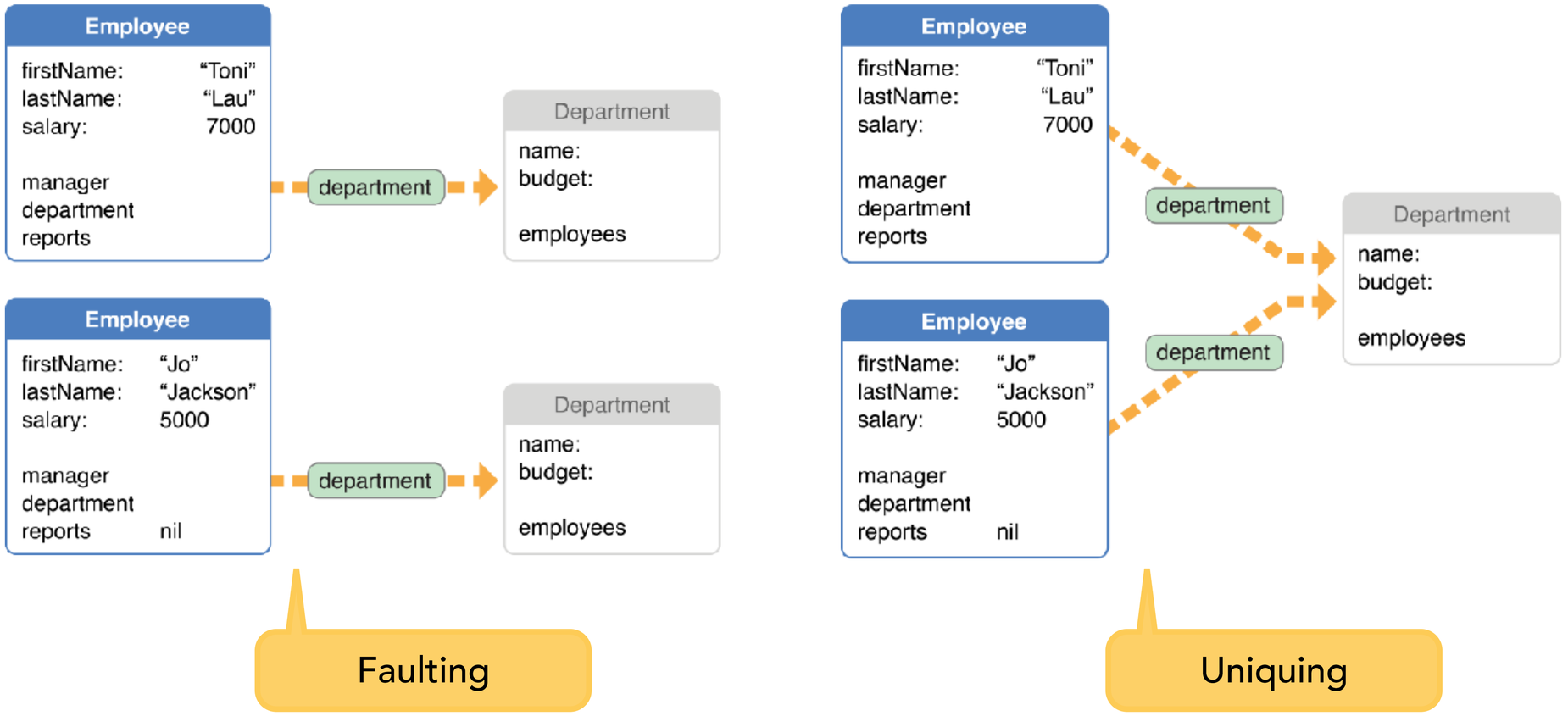
CORE DATA

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



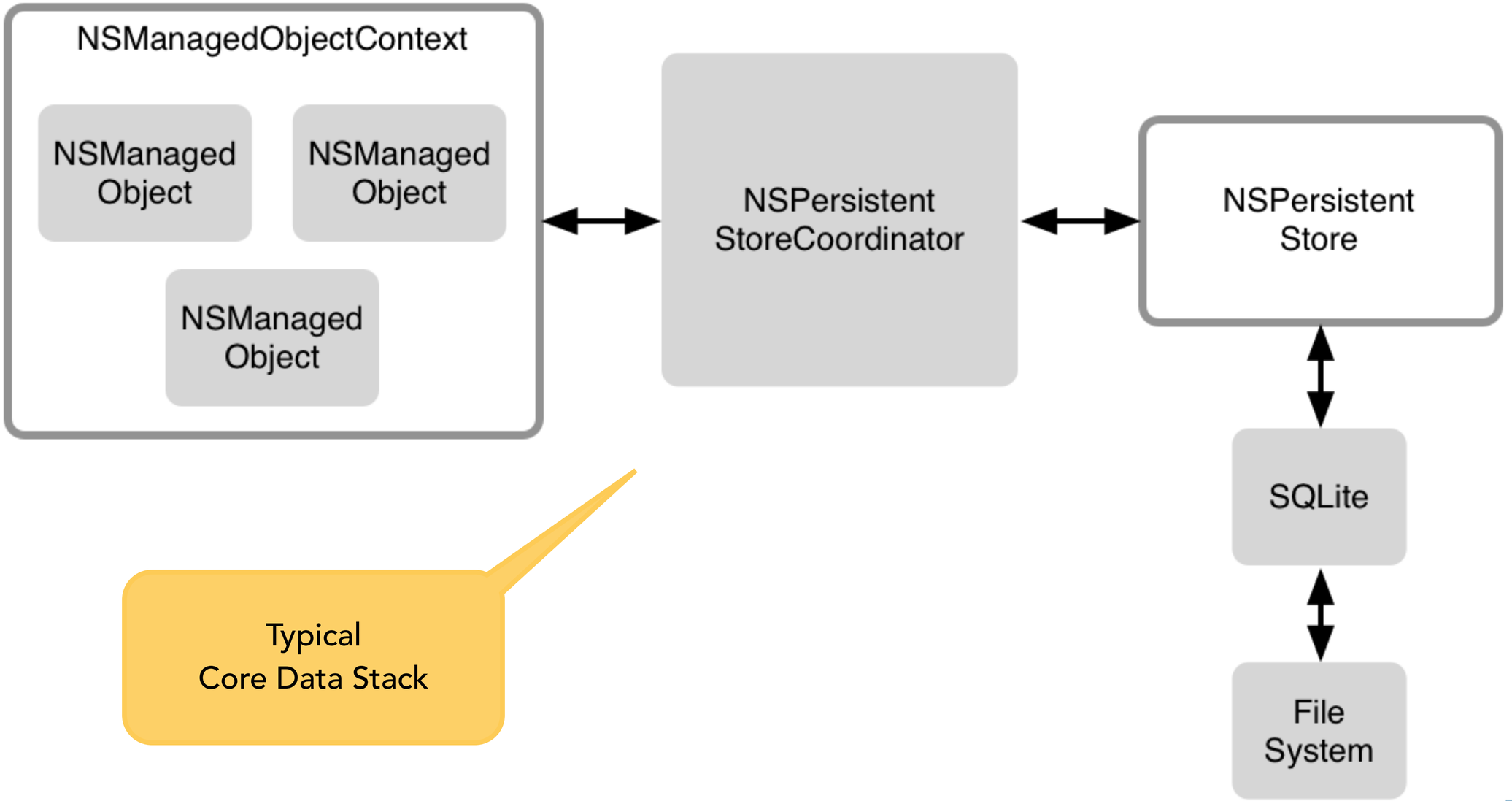
CORE DATA



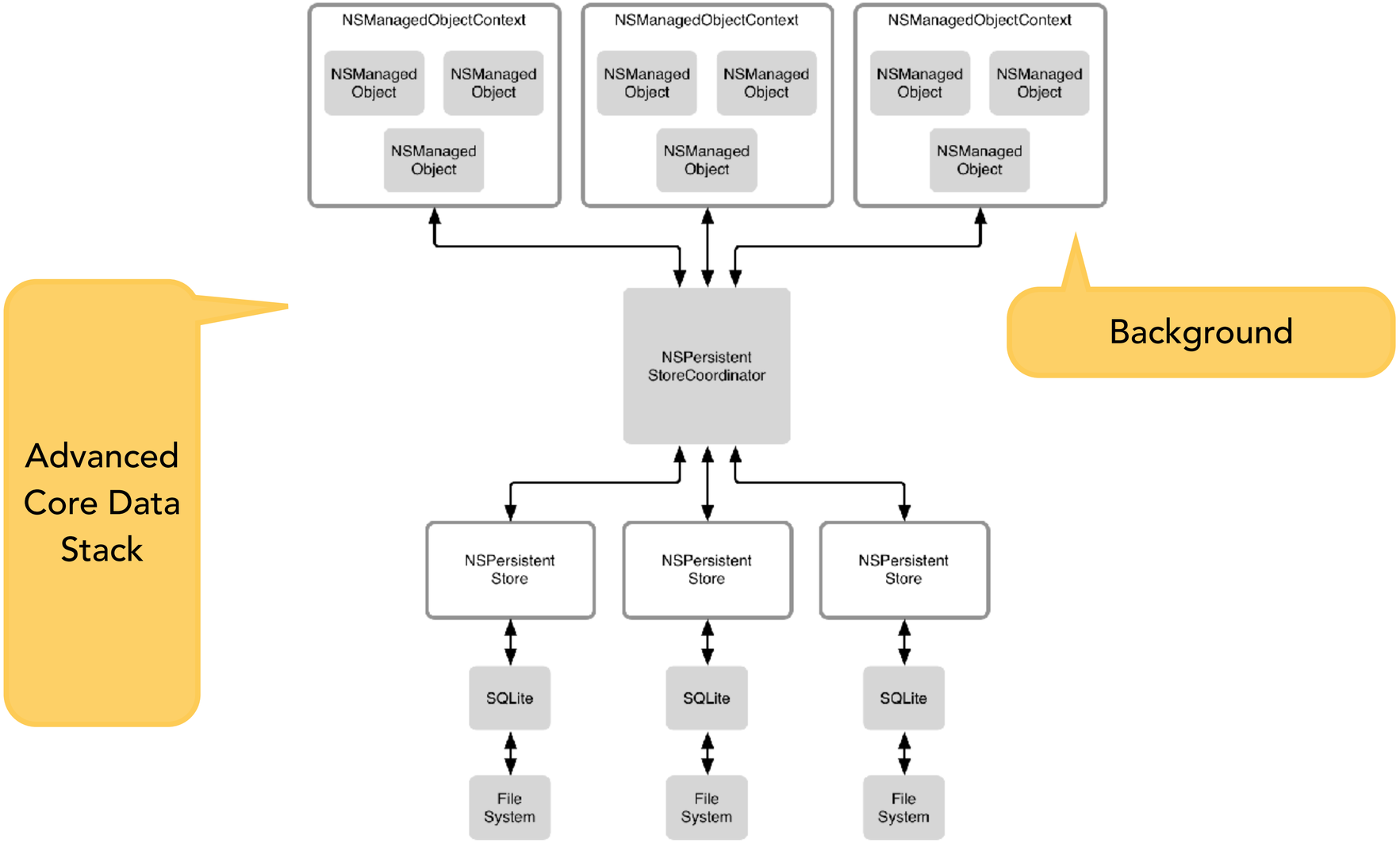
Faulting is a mechanism Core Data employs to reduce your application's memory usage

Uniquing ensures that you never have more than one managed object to represent a given record

CORE DATA


NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSPersistent StoreCoordinator
NSPersistent Store
SQLite
File System
Typical
Core Data Stack

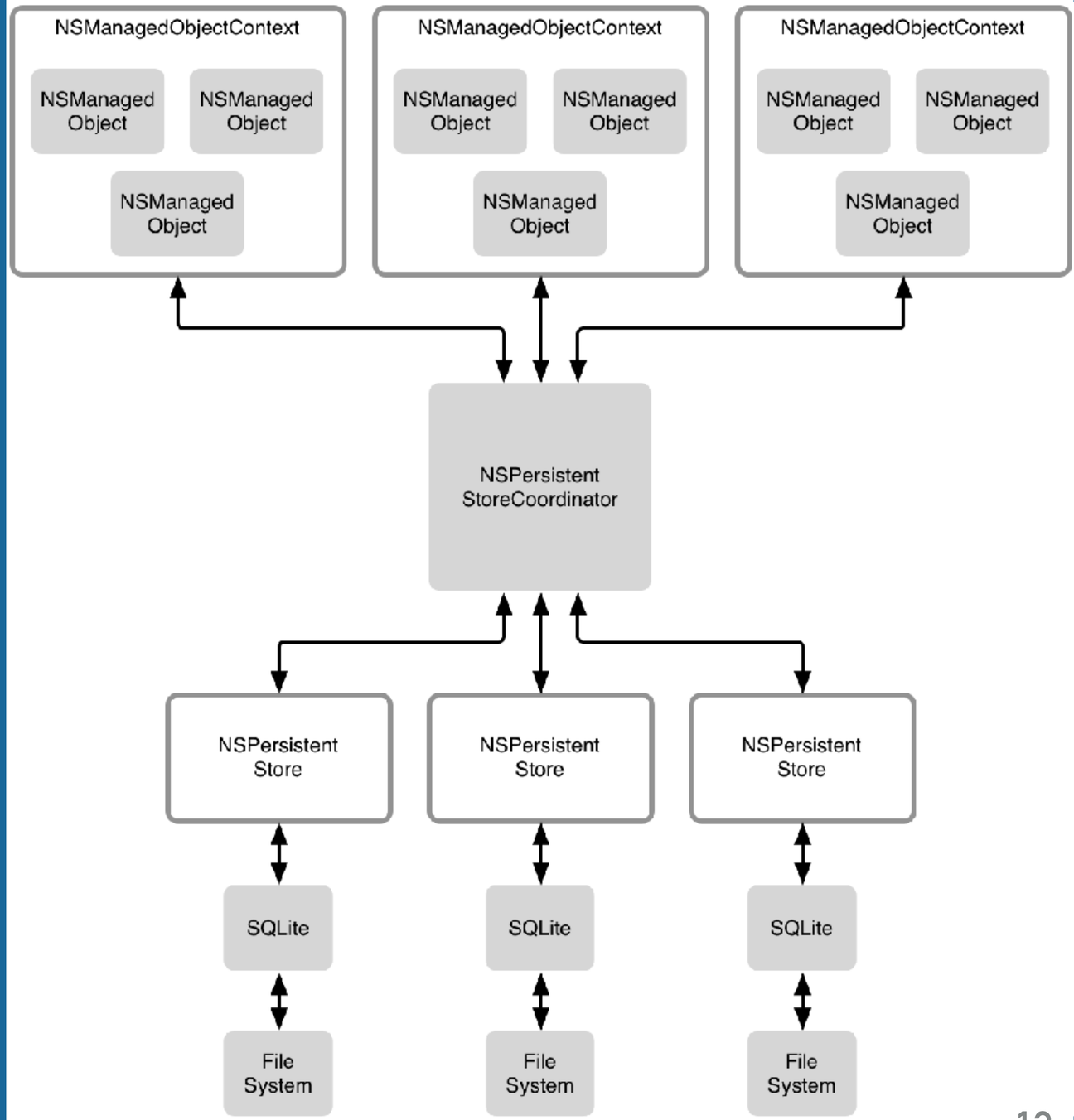
CORE DATA


NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSPersistent StoreCoordinator
NSPersistent Store
NSPersistent Store
NSPersistent Store
SQLite
SQLite
SQLite
File System
File System
File System
Advanced Core Data Stack
Background

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

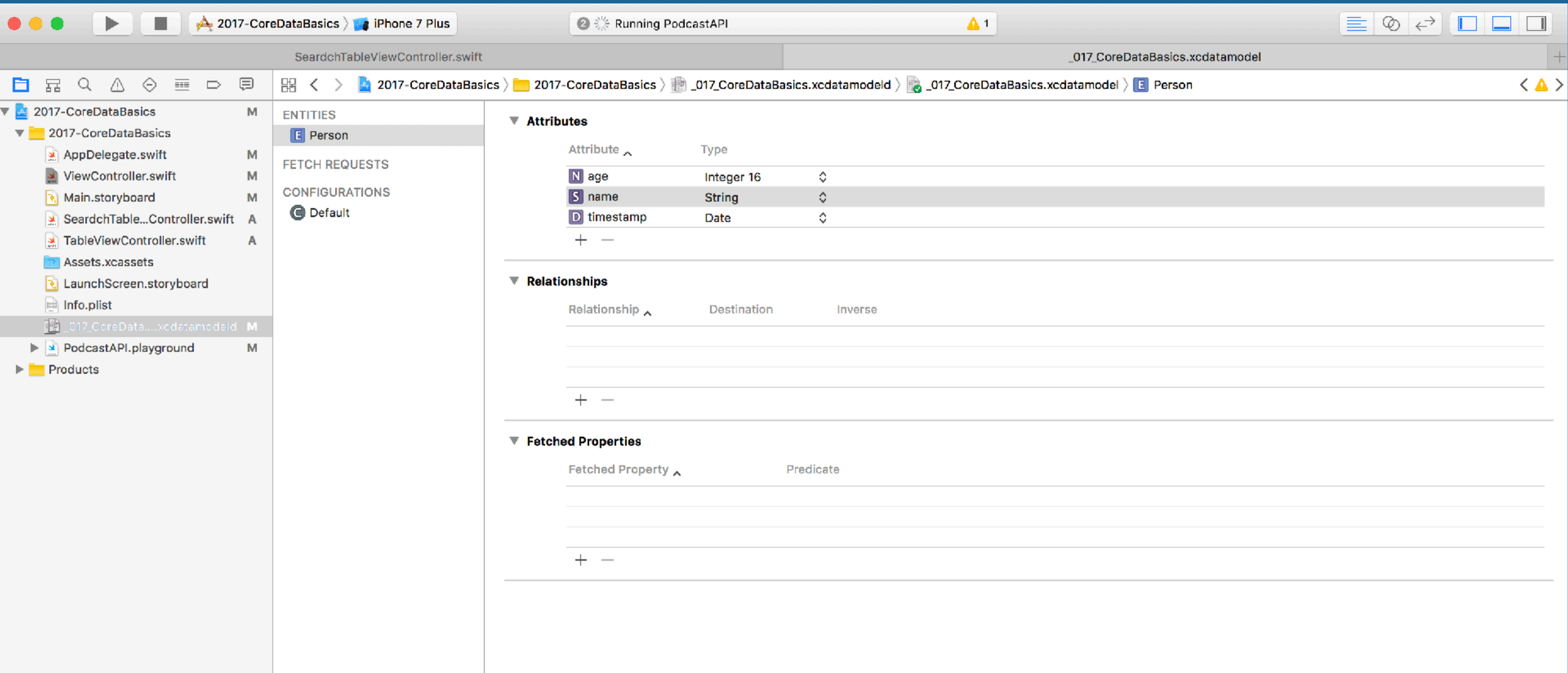
APPLICATION FLOW

- 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



CORE DATA

APPLICATION FLOW

- Create in 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: NSFetchRequest<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?

Already done for you

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.

It's a mature technology

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.

Xcode integration

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 Interface Builder to allow you to create user interfaces from your model. These aspects help to further shorten the application design, implementation, and debugging cycles.

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

CORE DATA APPLICATION FLOW

SUBTITLE

- 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 name
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMas
    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 coord
    // 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:
    } 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 fun
        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 stor
    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 = NSFetchRequest(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!
```

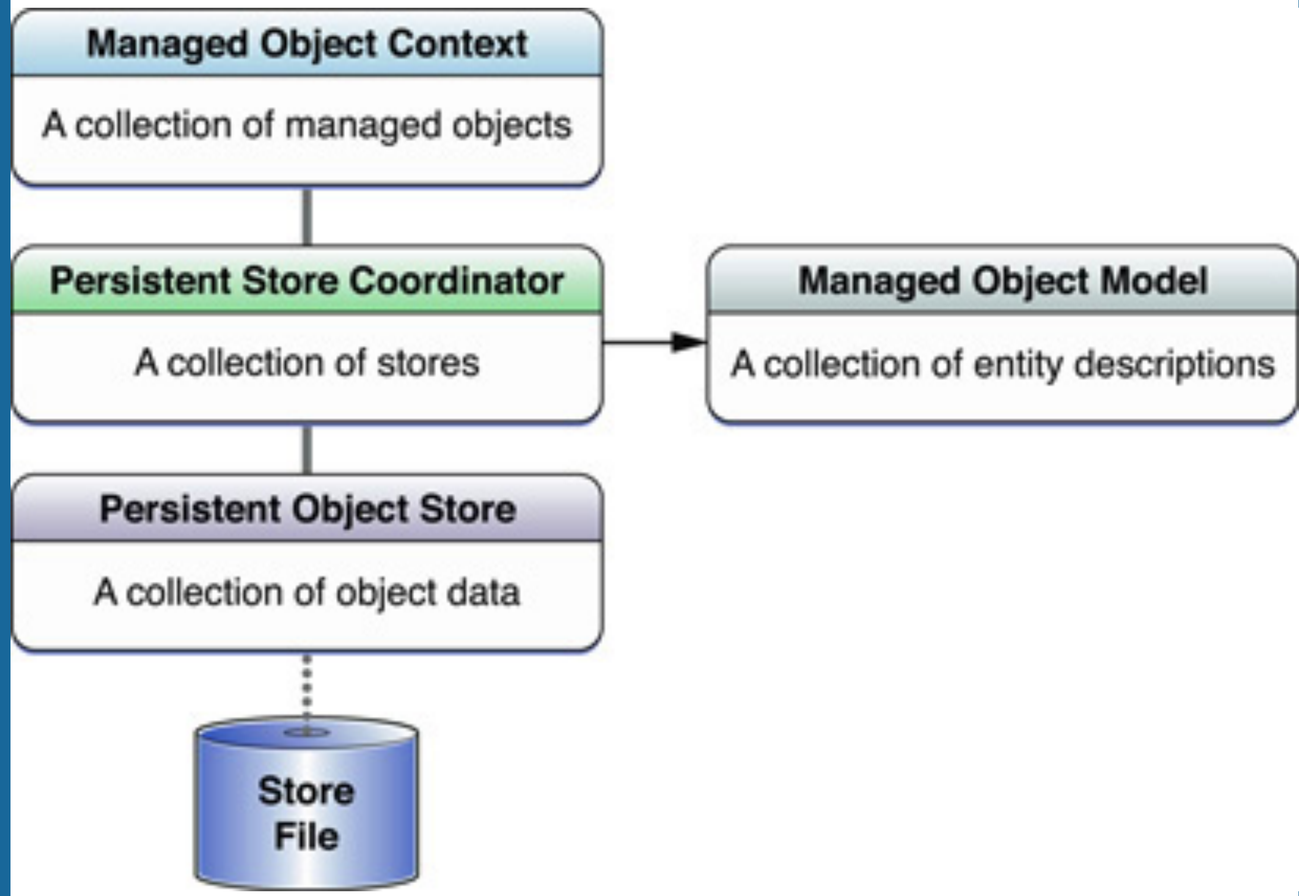
- Fetch

CORE DATA STACK

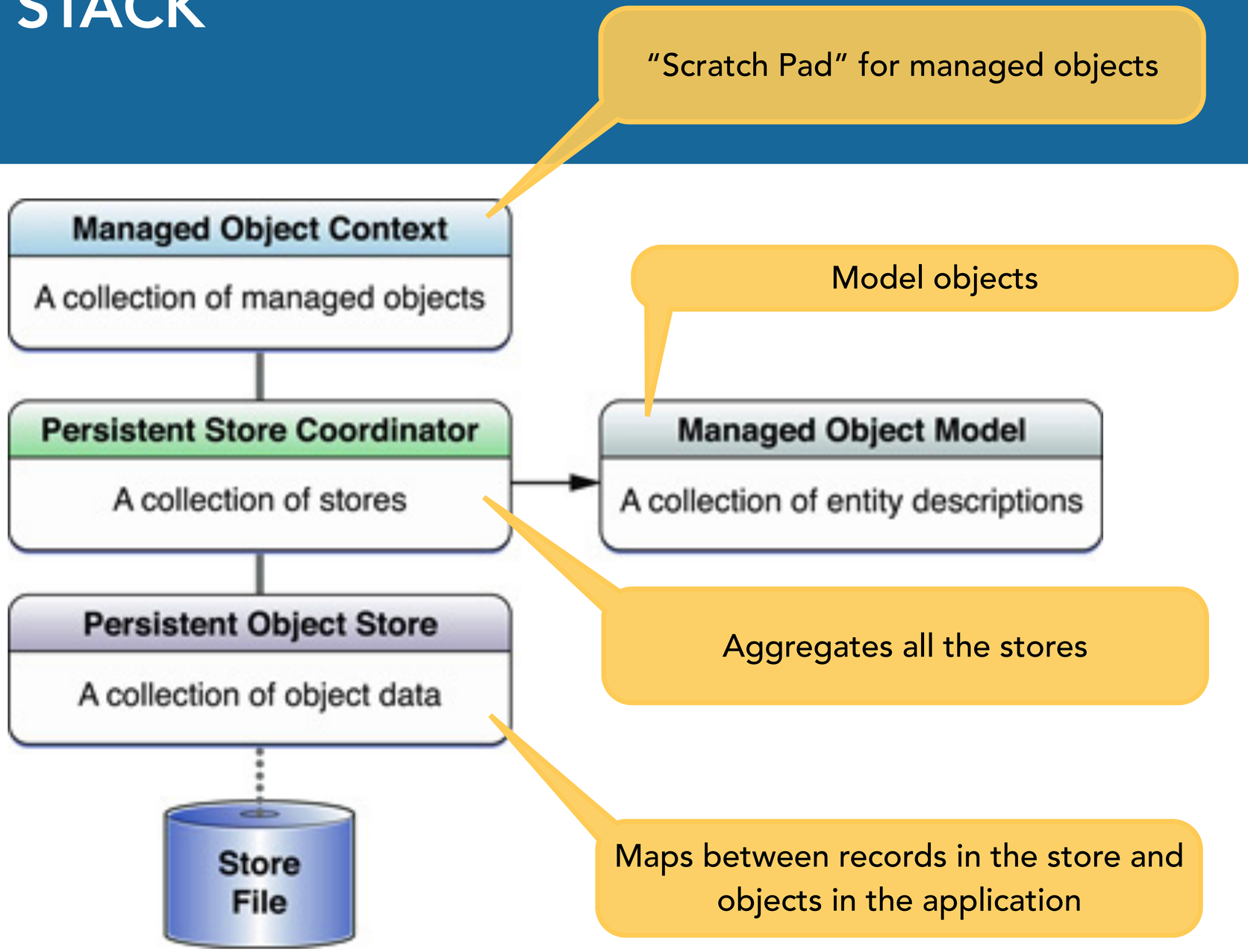
CORE DATA STACK

SUBTITLE

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



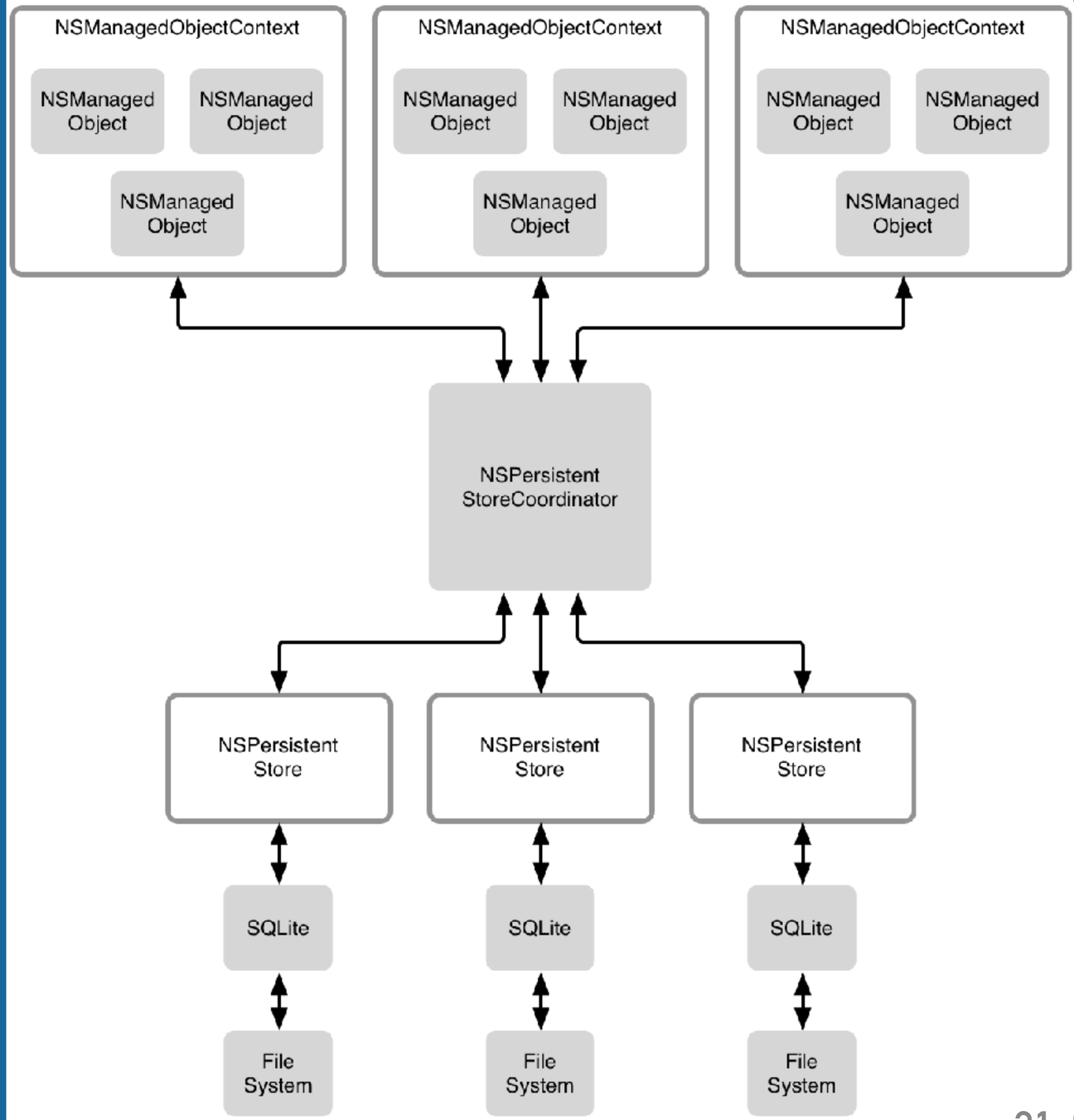
CORE DATA STACK



CORE DATA STACK

SUBTITLE

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



CORE DATA STACK

XCODE TEMPLATE

- Xcode will create codes 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 appropr
            // fatalError() causes the application to generate a crash log and t
                useful during development.

            /*
             Typical reasons for an error here include:
             * The parent directory does not exist, cannot be created, or disall
             * The persistent store is not accessible, due to permissions or dat
             * 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
}()
```

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.
                useful during development.
            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 loose 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

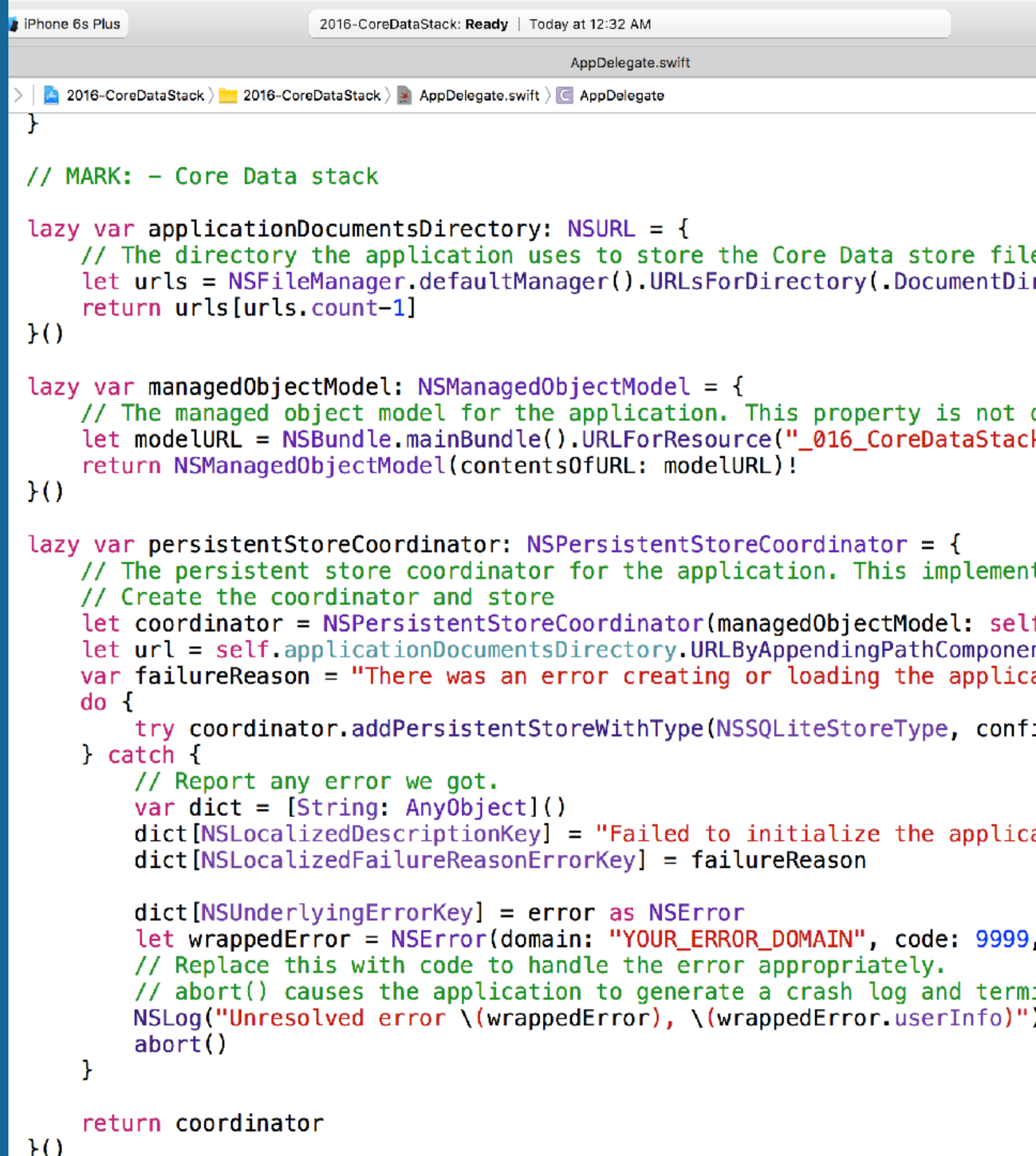
OLD CORE DATA STACK

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



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 applicat
    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, havi
    // 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 s
        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

The way people typically did it

```
// 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(.DocumentDir
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not o
    let modelURL = NSBundle.mainBundle().URLForResource("_016_CoreDataStack
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implement
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponen
    var failureReason = "There was an error creating or loading the applica
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, confi
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the applica
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999,
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and termi
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }

    return coordinator
}()
```

CORE DATA STACK

WORKING WITH DATA

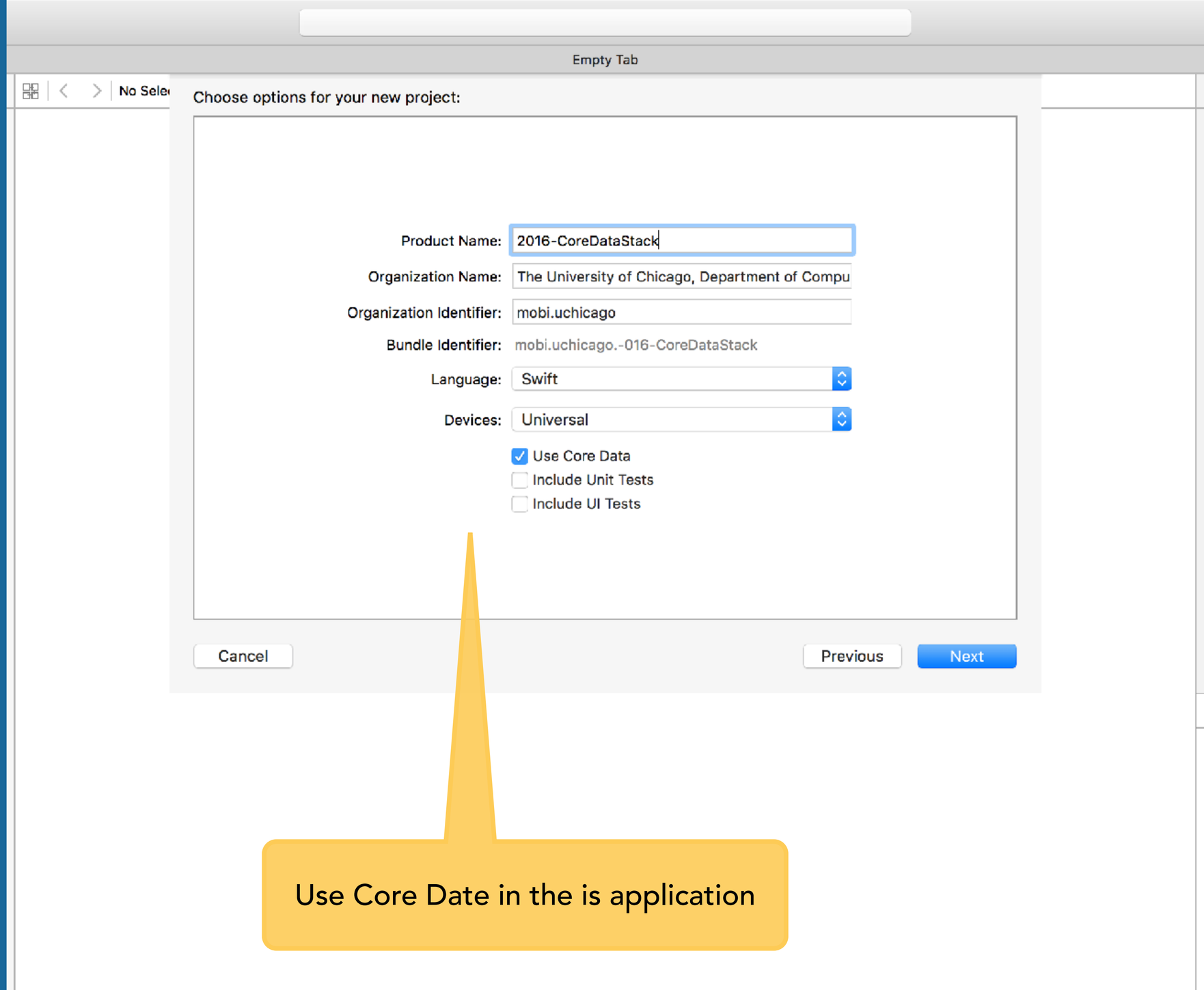
- Create objects in the database
  - insertNewObjectForEntityName()
- Access your data
  - - (id)valueForKey:(NSString *)key;
- Set your data
  - - (void)setValue:(id)value forKey:(NSString *)key;

Without a custom subclass

CREATE A CORE DATA MODEL

CORE DATA MODEL

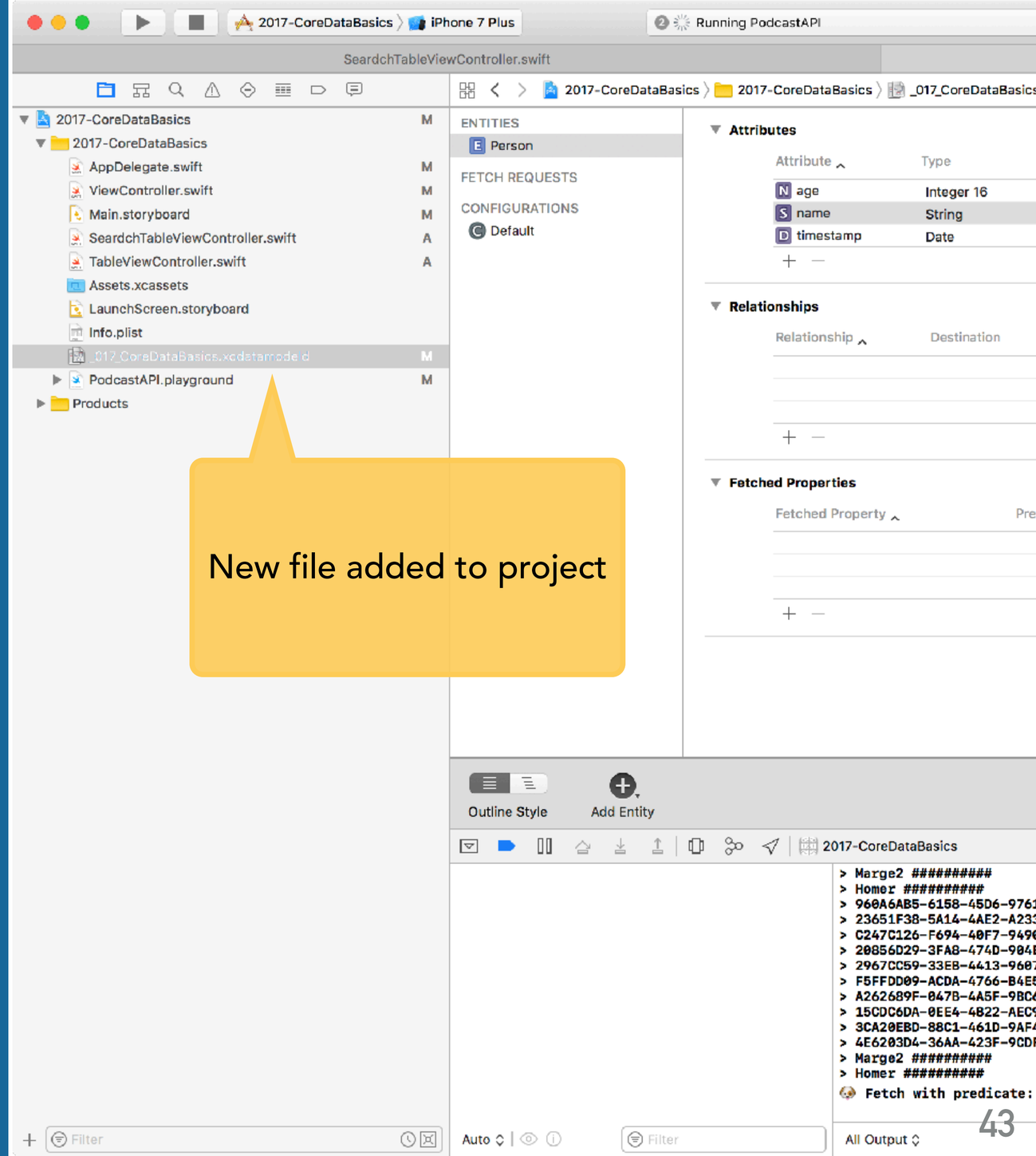
- Provide a simple data model (.xcdatamodel)
- Xcode will generate the boiler plate methods



CORE DATA

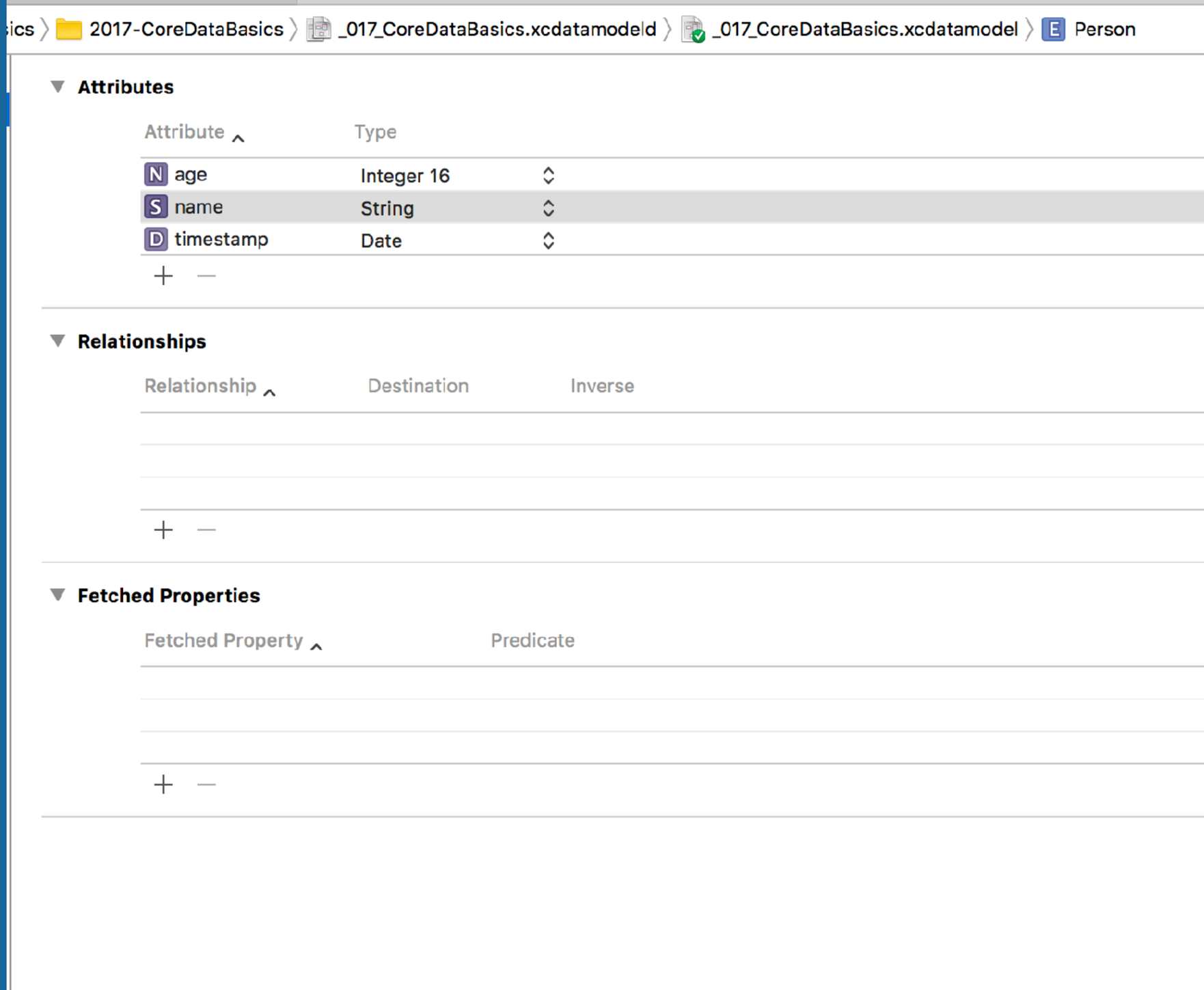
SUBTITLE

- .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

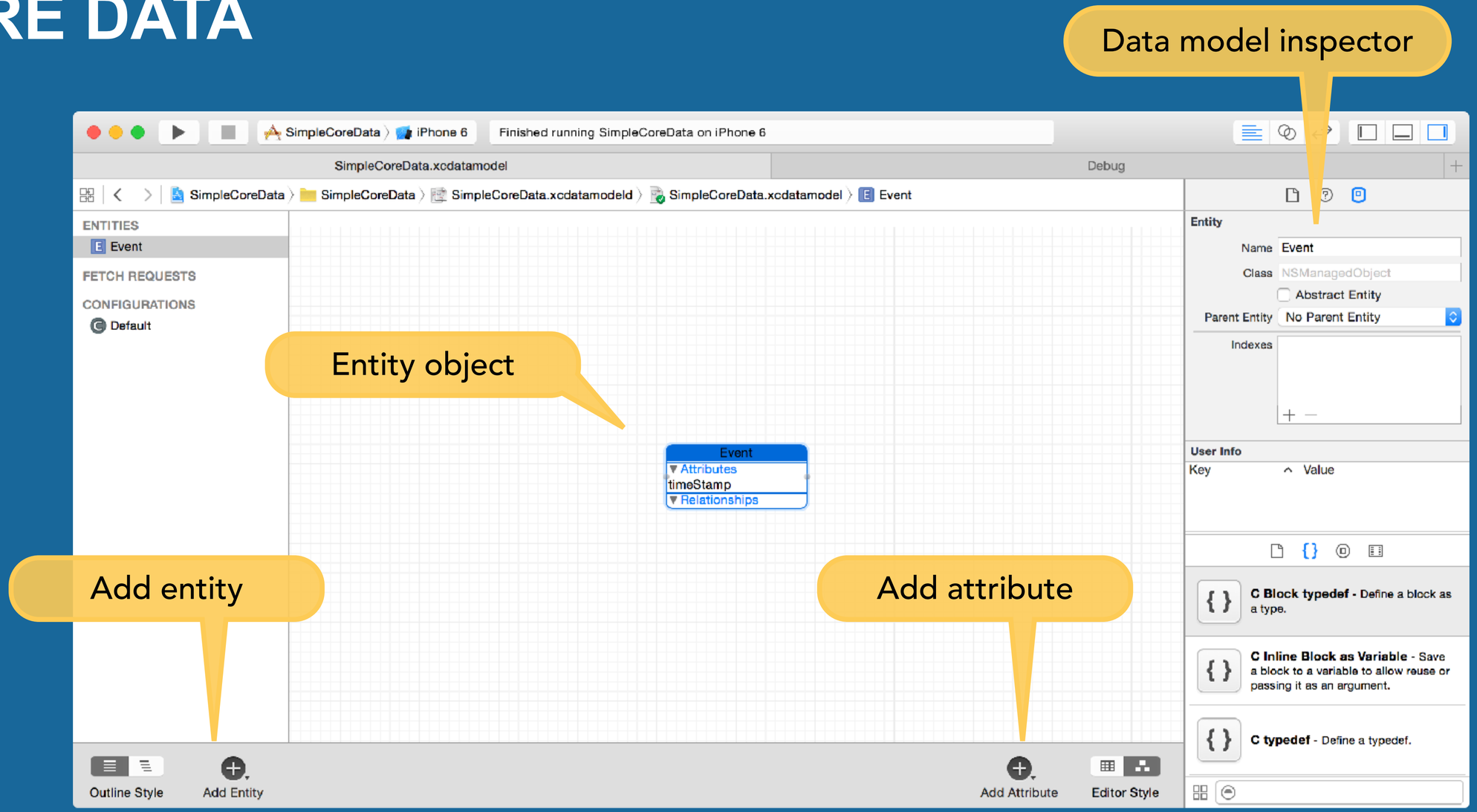


CORE DATA

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



CORE DATA

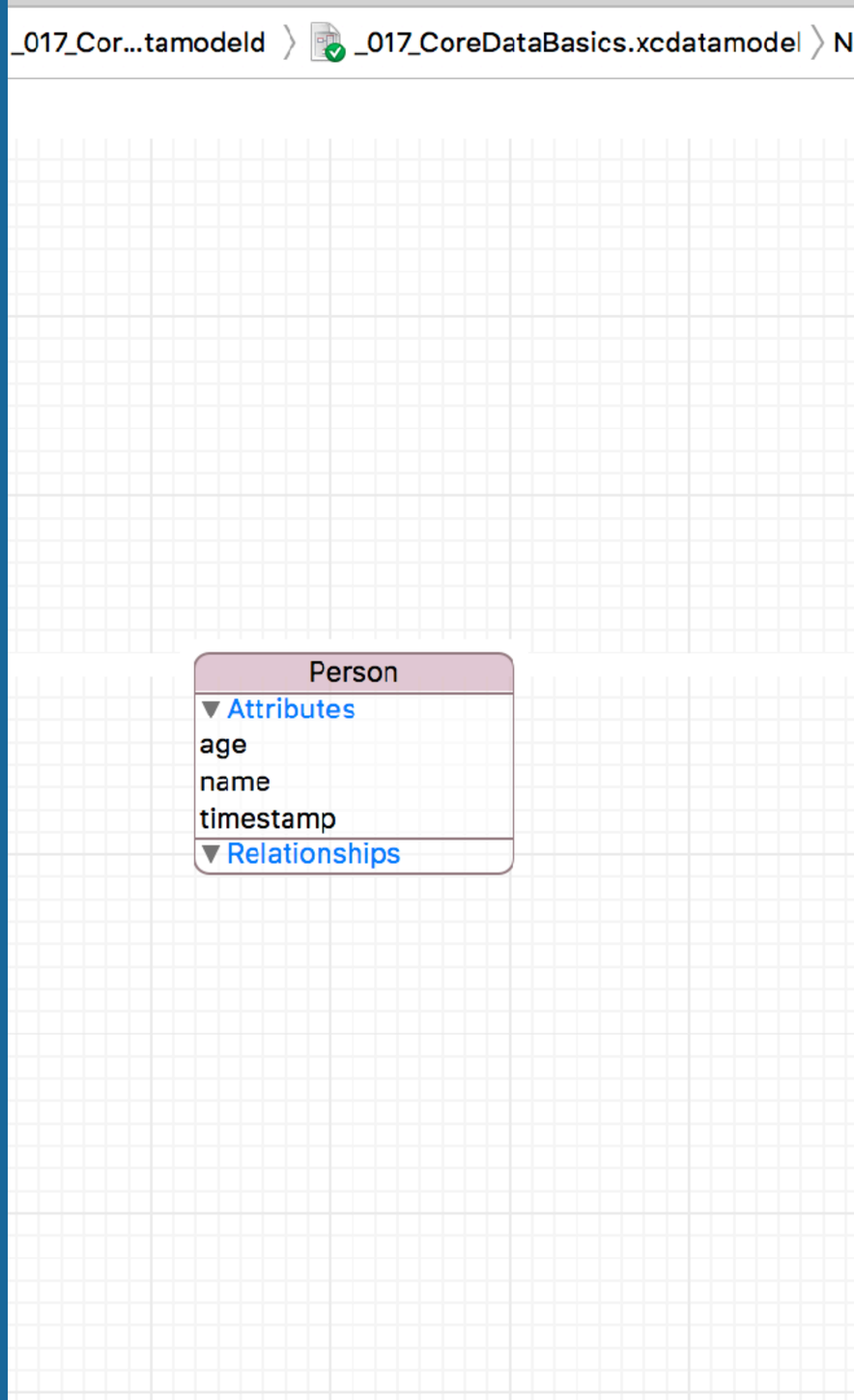


- Entity maps a database table to an object in our code

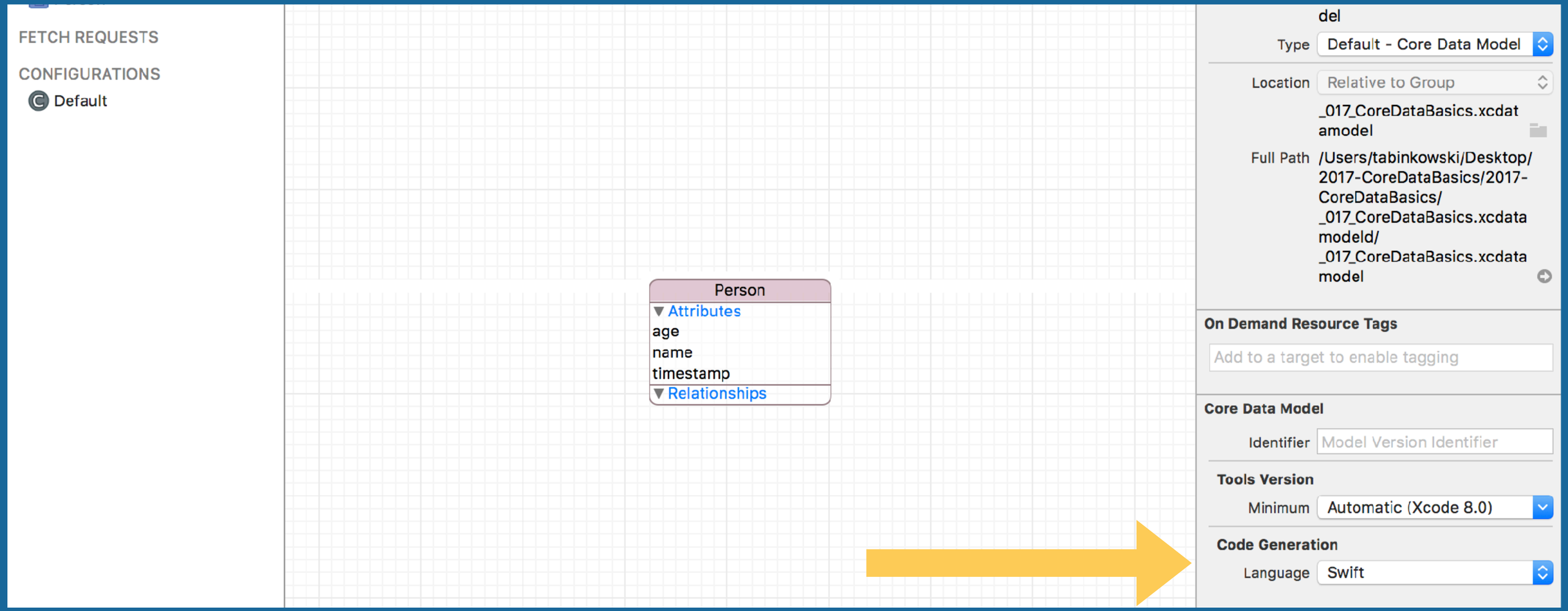
CORE DATA

- NSManagedObject
  - Subclass of NSObject
  - Access you 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)

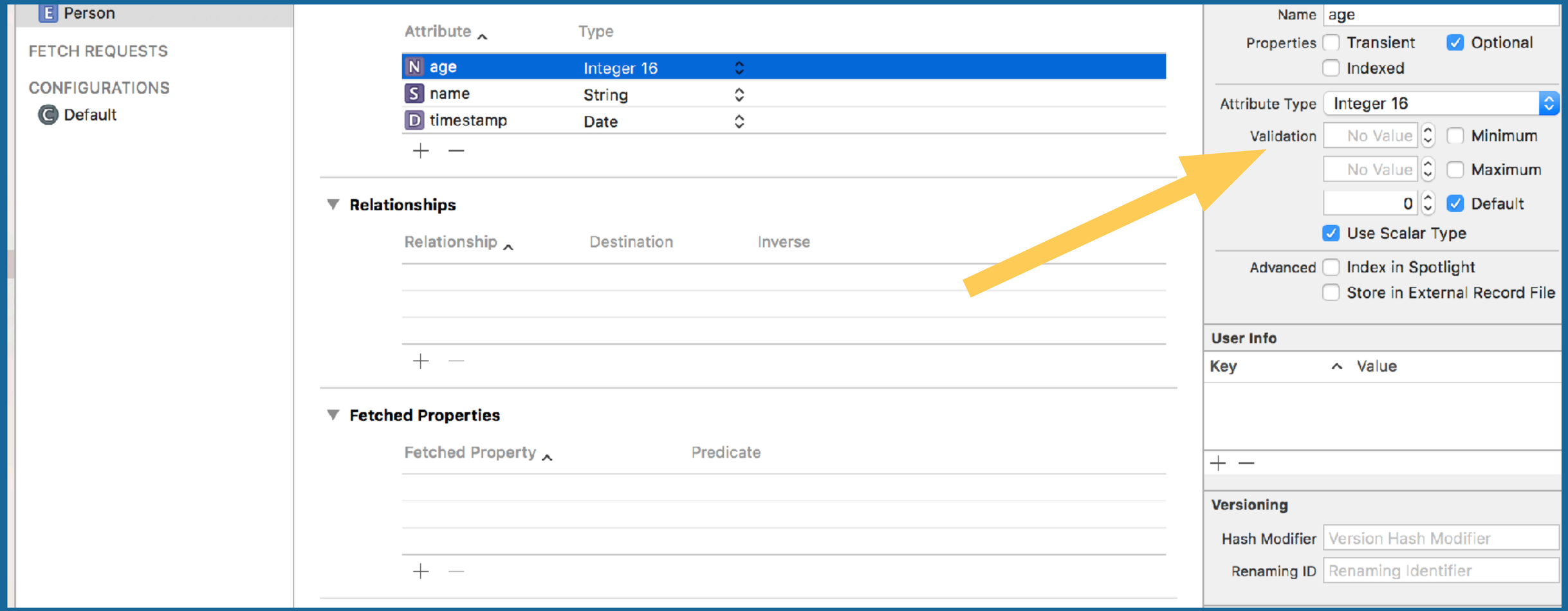


CORE DATA



- Xcode will generate subclass files of NSManagedObject for you

CORE DATA

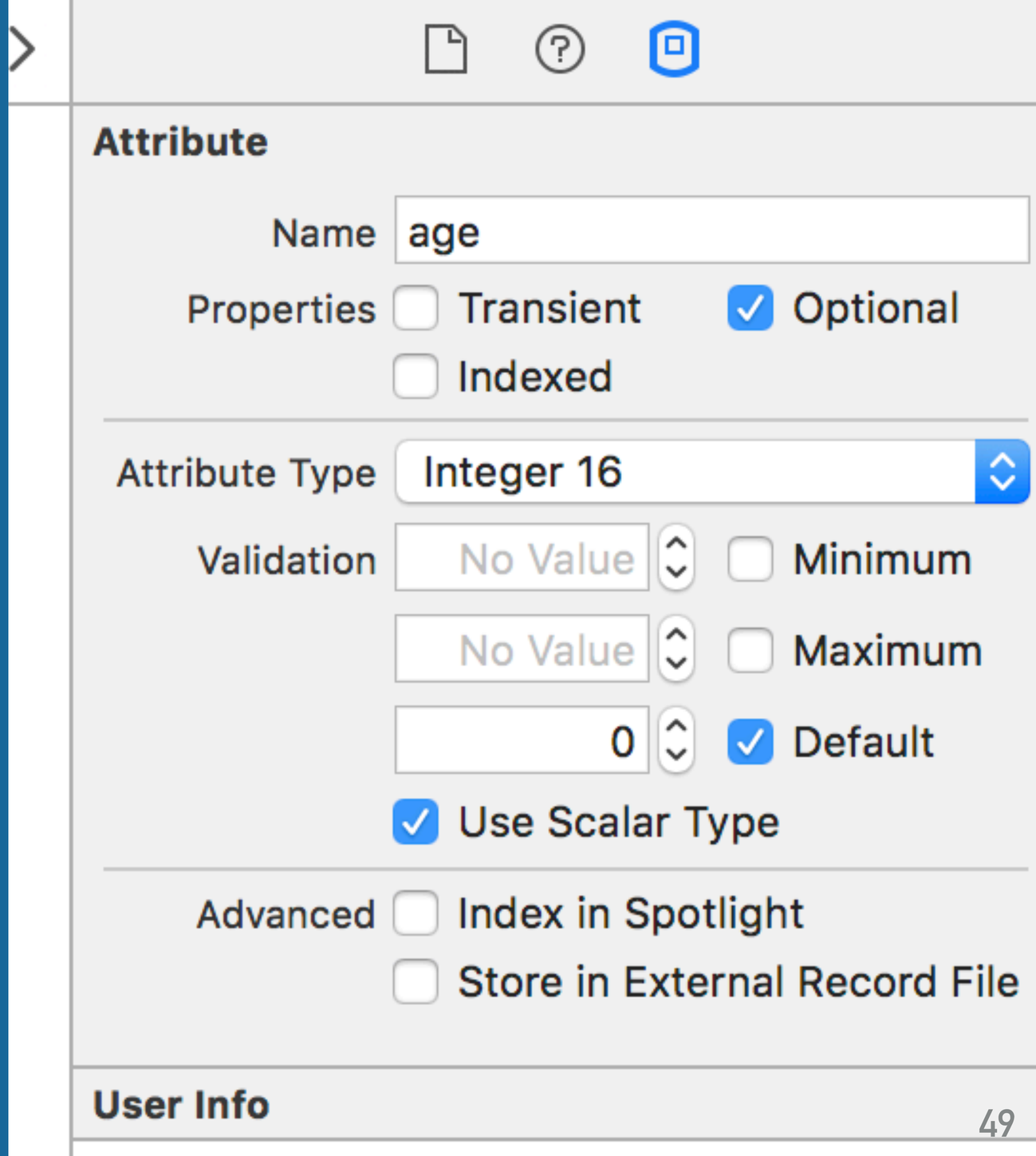


- Attributes map to a database column
- Will be able to access via @property in code

CORE DATA

SUBTITLE

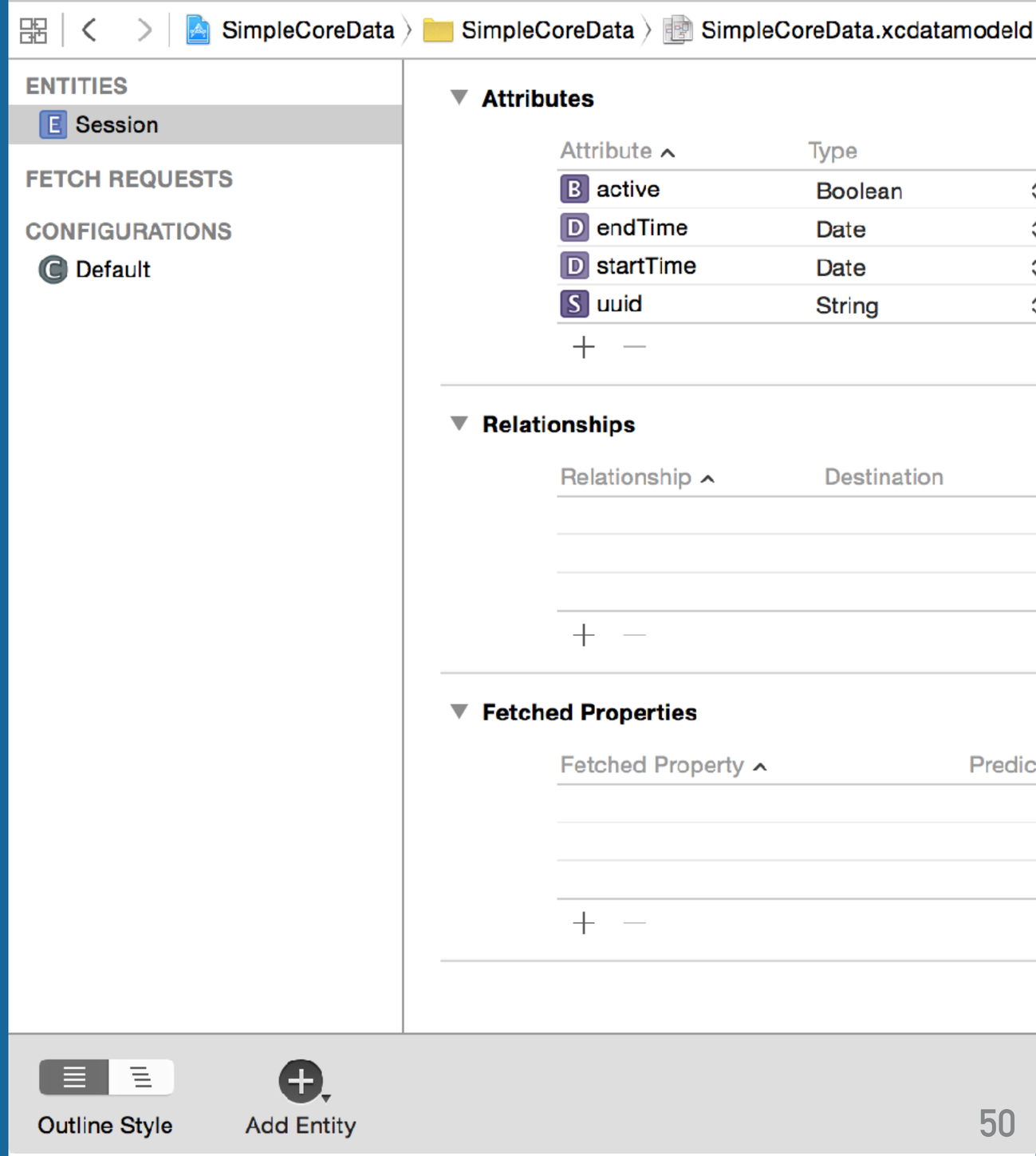
- Attributes
  - Type
    - Maps to Foundation object
  - Optional
  - Transient
    - Create a property that exist on object, but have no backing
  - Indexed
    - Create an index in the SQLite backend



CORE DATA

SUBTITLE

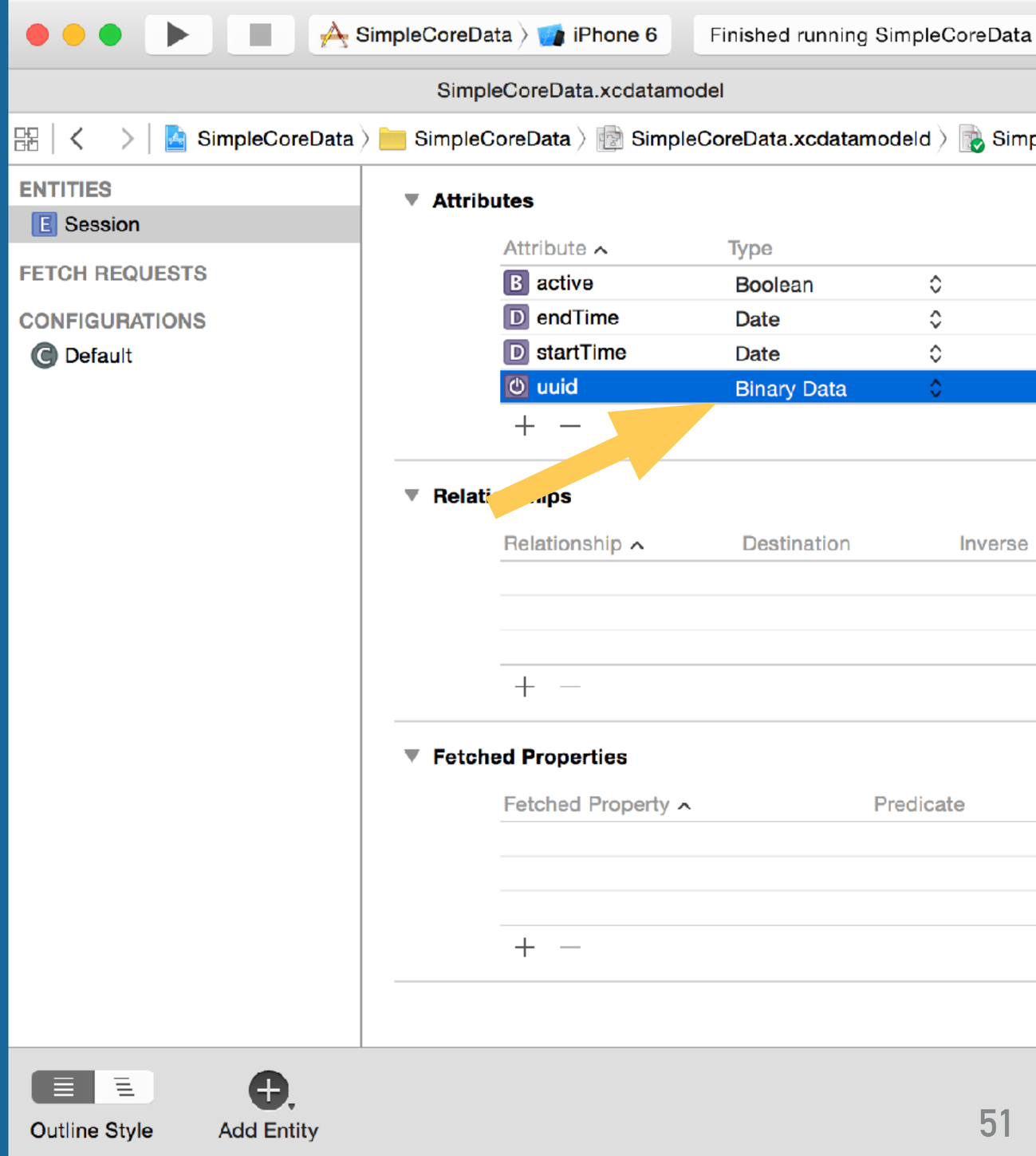
- Attribute types
  - Foundation types
    - NSData for everything else
  - Binary Data for holding images
  - Transformable for nonstandard object types



CORE DATA

SUBTITLE

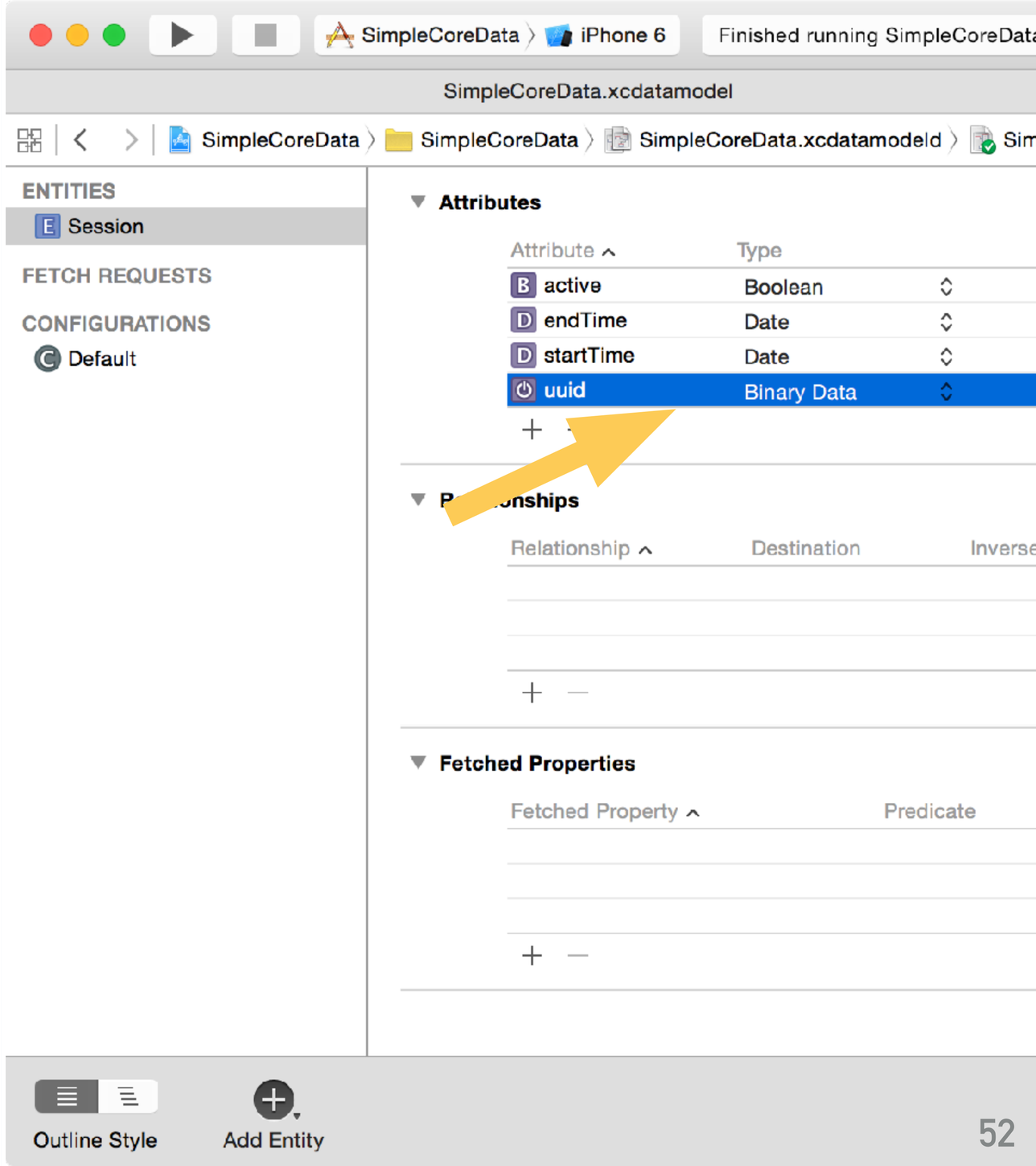
- External Binary Data
  - New in iOS5
- When enabled, Core Data heuristically decides on a per-value basis
  - Save the data directly in the database
  - Store a URI to a separate file



CORE DATA

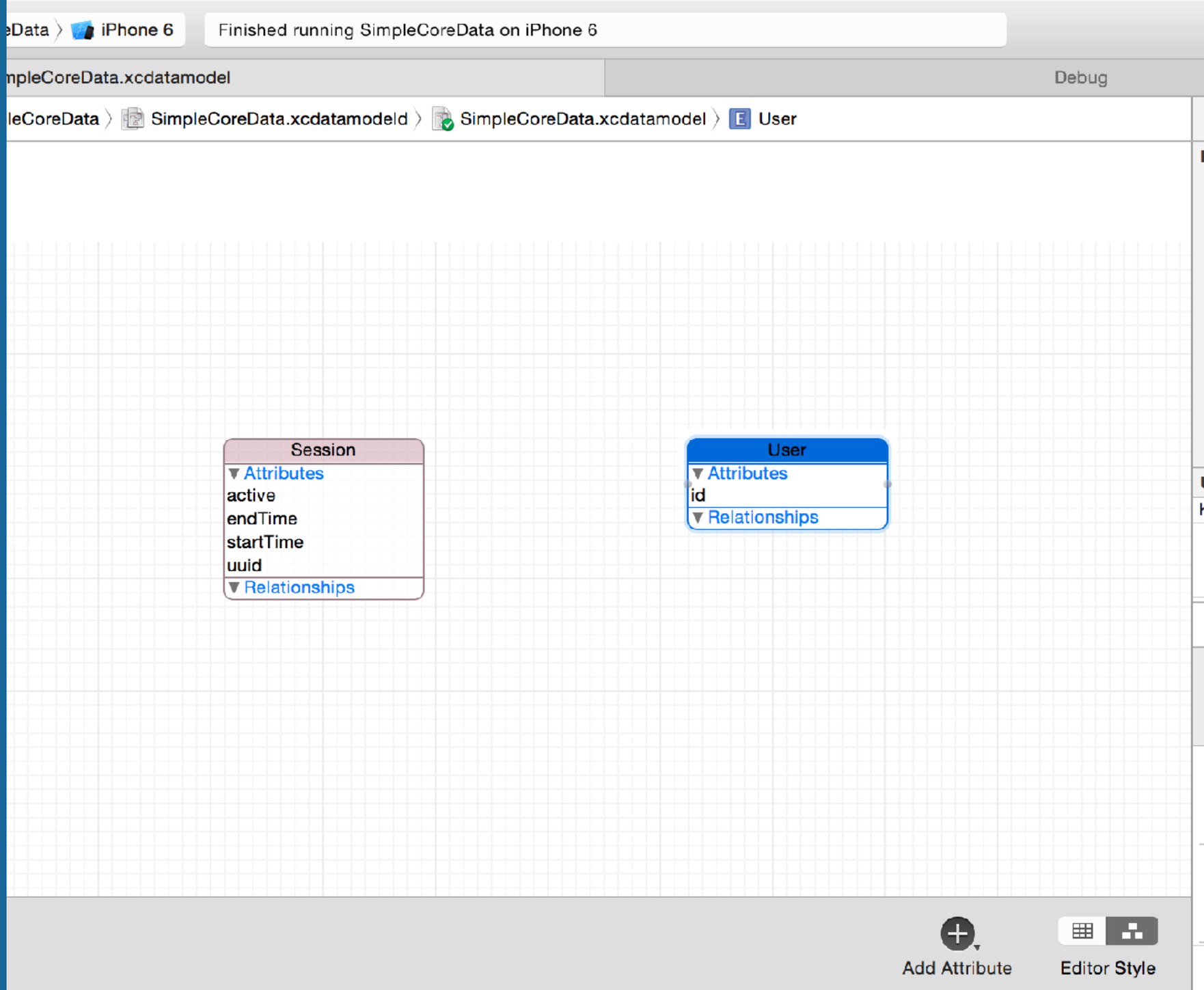
SUBTITLE

- 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

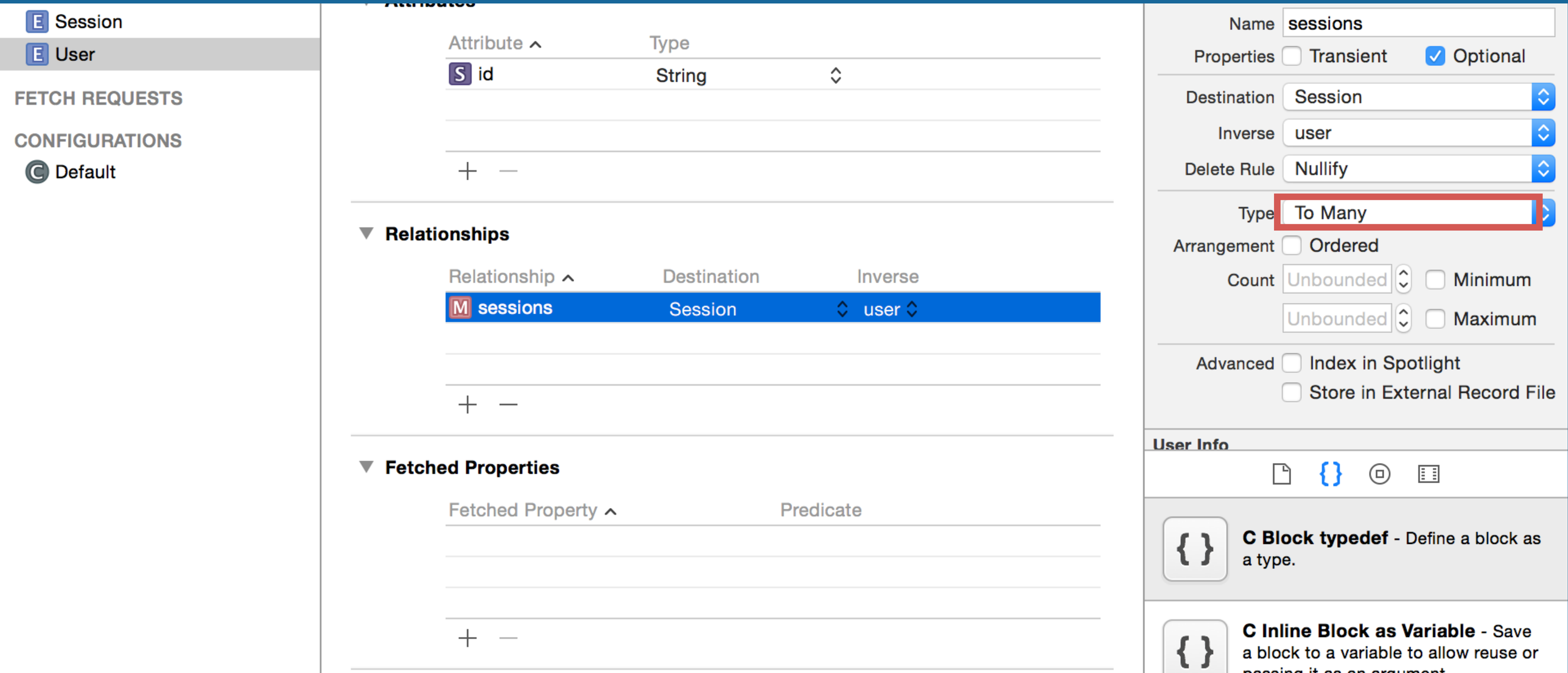


CORE DATA

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



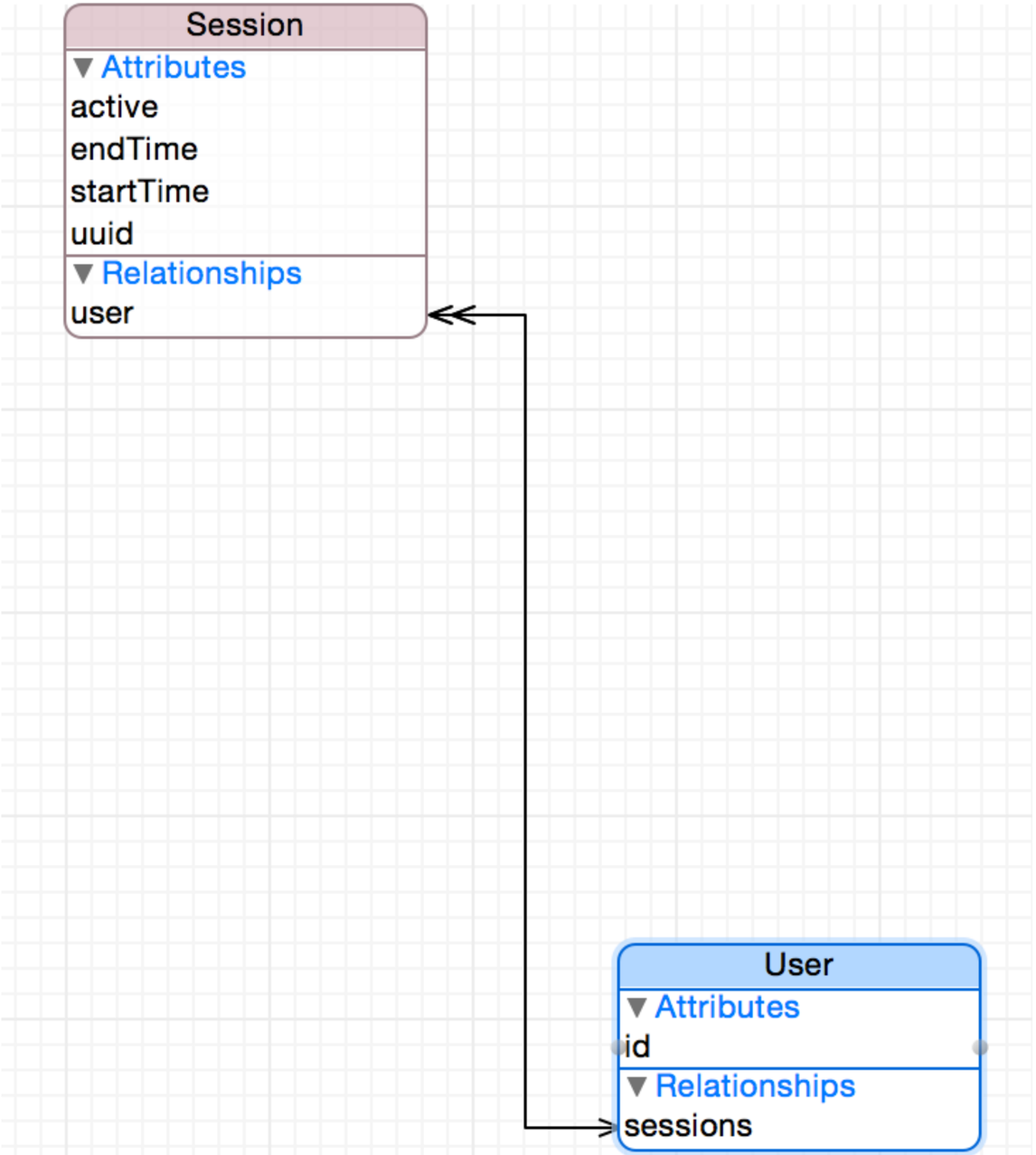
CORE DATA


Session
User
FETCH REQUESTS
CONFIGURATIONS
Default
Attribute
Type
id
String
Relationships
Relationship
Destination
Inverse
sessions
Session
user
Fetched Properties
Fetched Property
Predicate
Name
sessions
Properties
Transient
Optional
Destination
Session
Inverse
user
Delete Rule
Nullify
Type
To Many
Arrangement
Ordered
Count
Unbounded
Minimum
Unbounded
Maximum
Advanced
Index in Spotlight
Store in External Record File
User Info
C Block typedef - Define a block as a type.
C Inline Block as Variable - Save a block to a variable to allow reuse or

CORE DATA

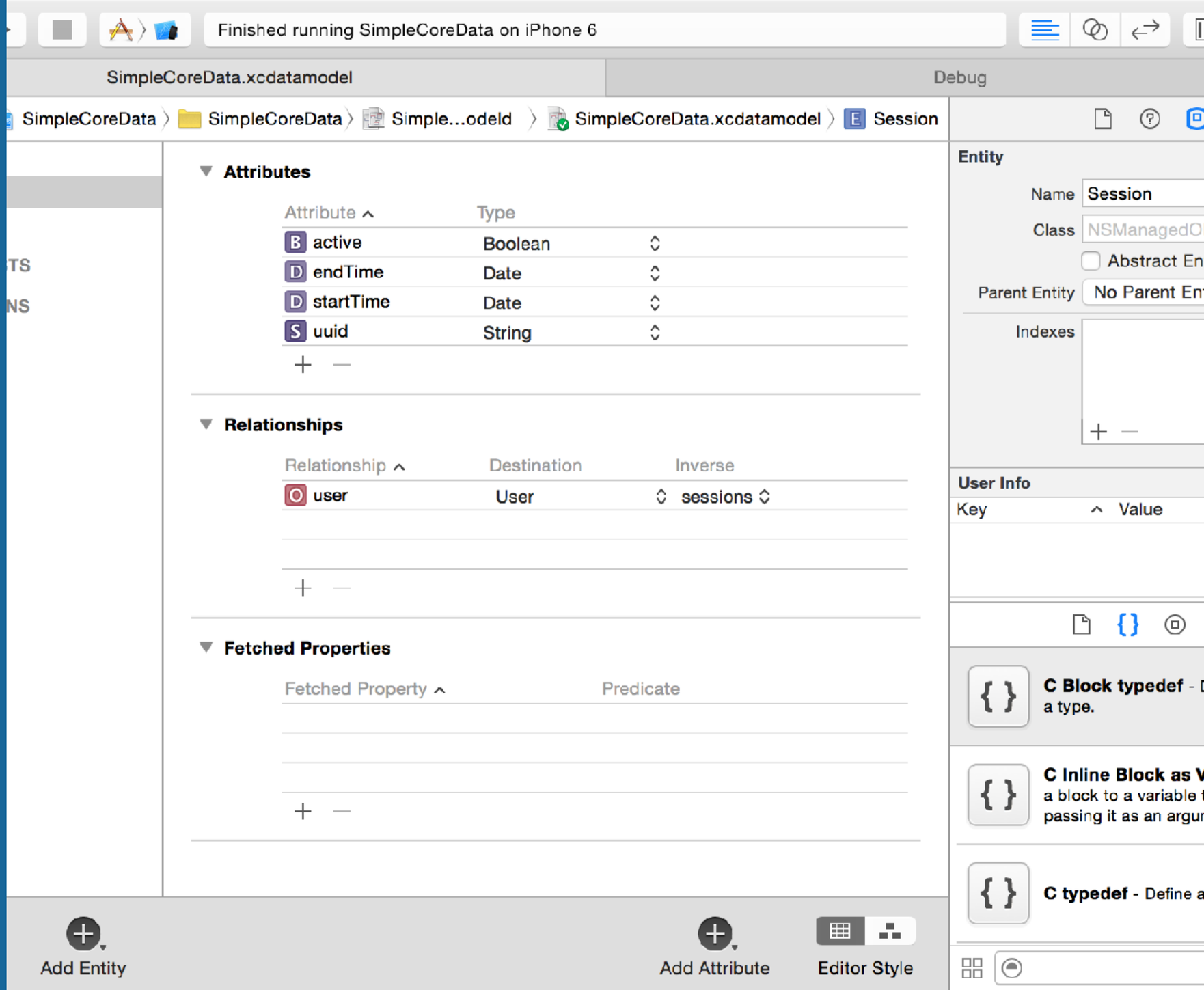
SUBTITLE

- “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

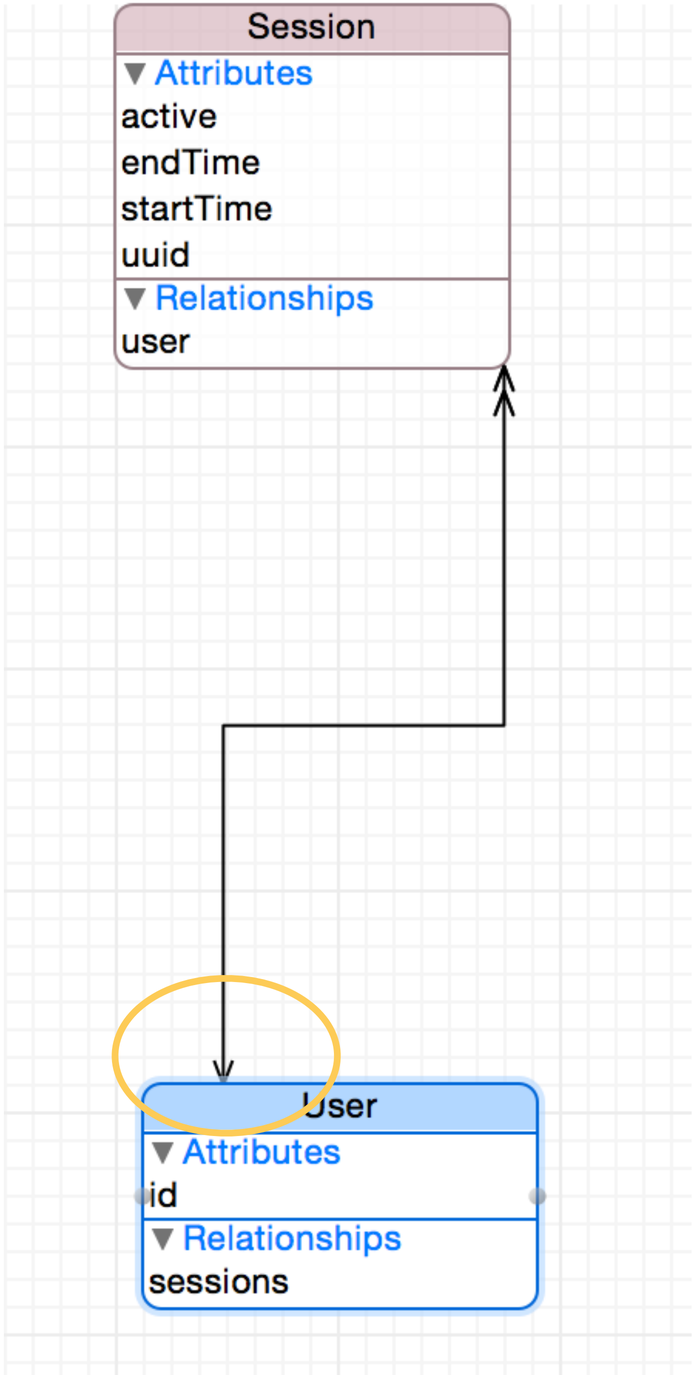
- Inverse relationship from user to sessions
  - Each session can have only 1 user



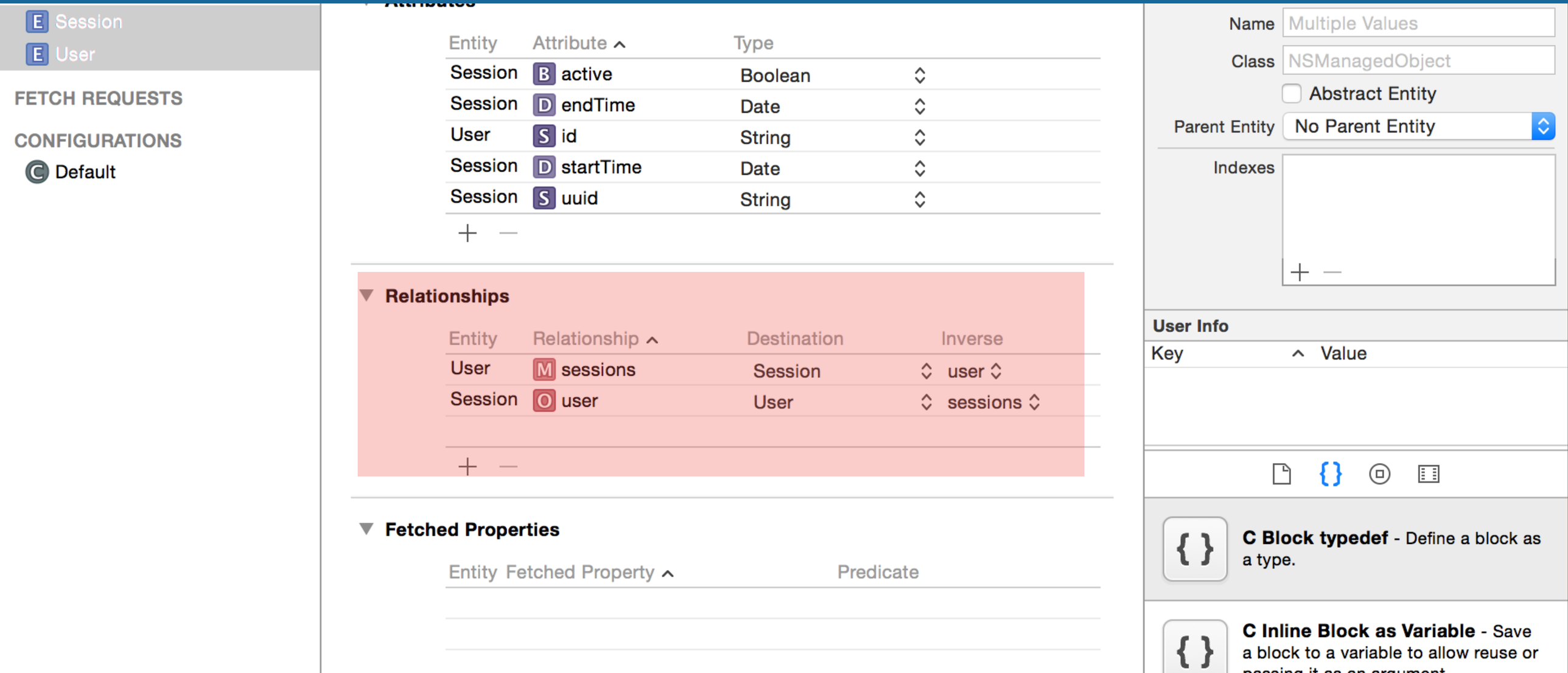
CORE DATA

SUBTITLE

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



CORE DATA

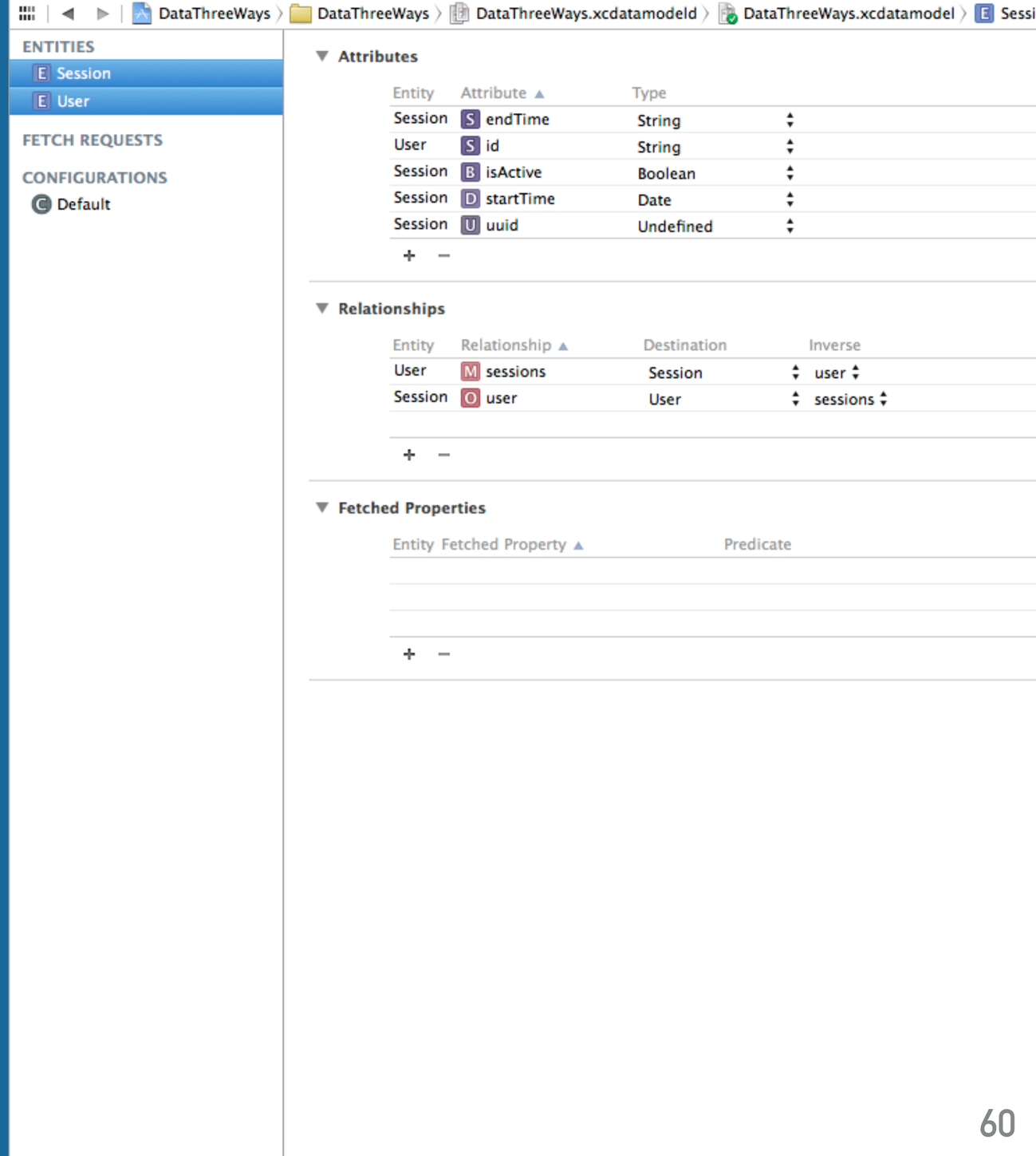

Session
User
FETCH REQUESTS
CONFIGURATIONS
Default
Entity Attribute Type
Session active Boolean
Session endTime Date
User id String
Session startTime Date
Session uuid String
Relationships
Entity Relationship Destination Inverse
User sessions Session user
Session user User sessions
Fetched Properties
Entity Fetched Property Predicate
Name Multiple Values
Class NSManagedObject
Abstract Entity
Parent Entity No 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

CREATE NSMANAGEDOBJECT SUBCLASS PRIOR TO IOS10

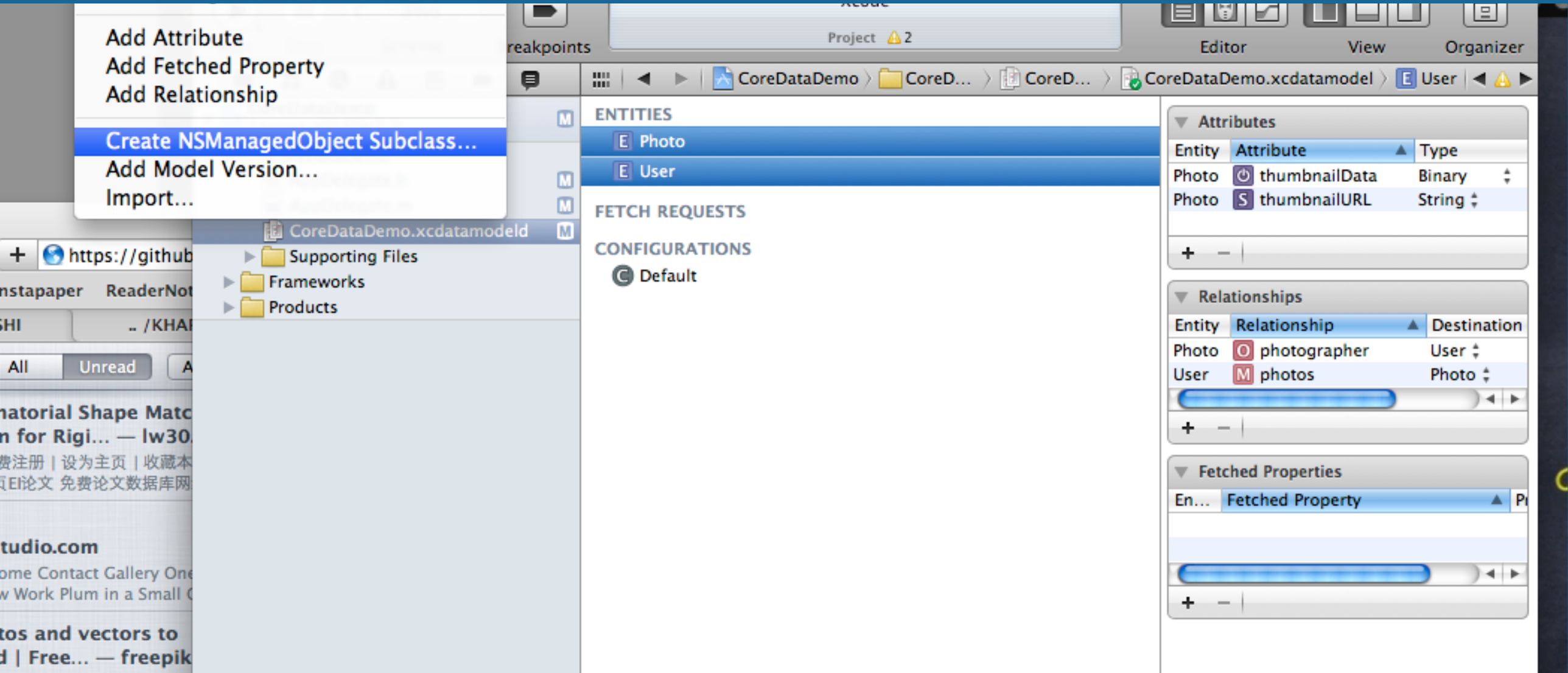
NSMANAGED OBJECT SUBCLASS

SUBTITLE

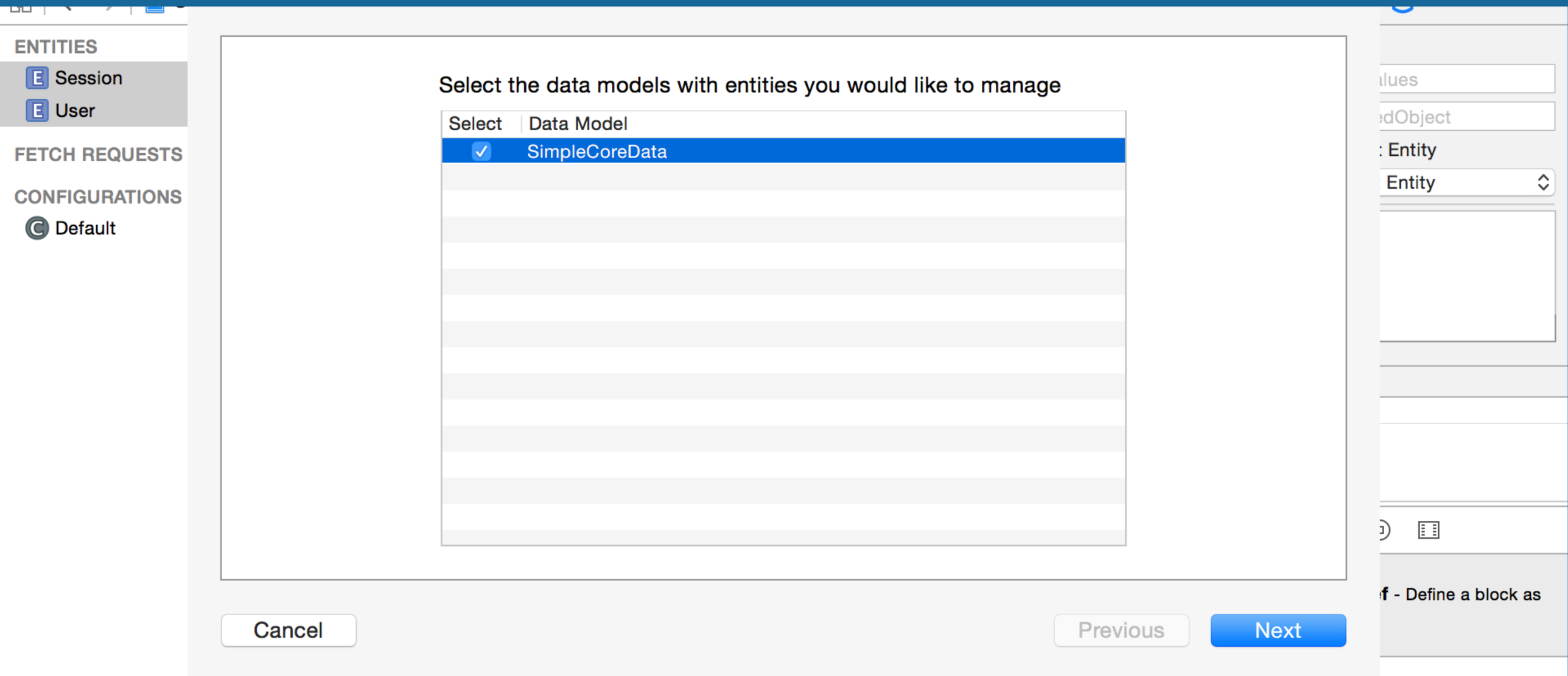
- Simplify access of object attributes
  - No type checking
- Create a subclass of NSManagedObject
  - @property for each attribute
- Xcode will auto-generate them for you



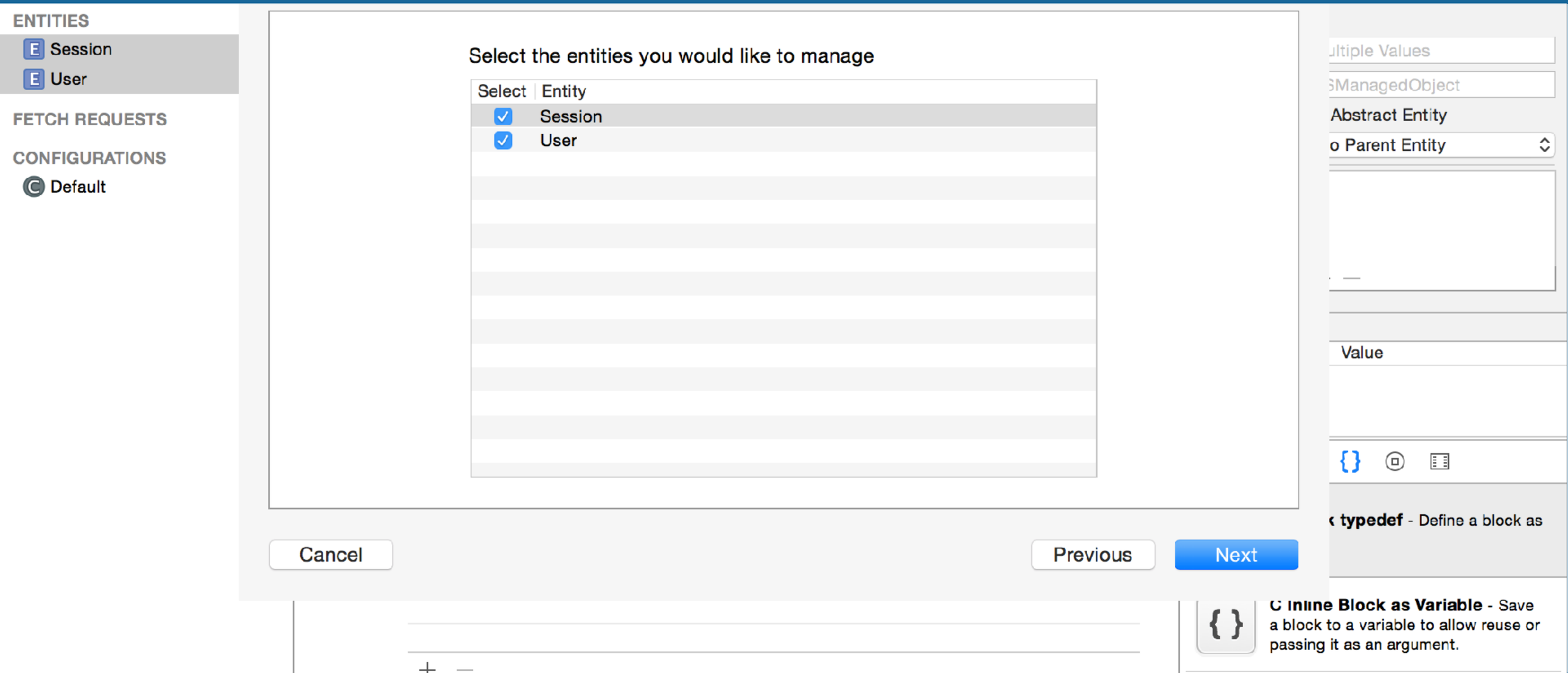
NSMANAGED OBJECT SUBCLASS



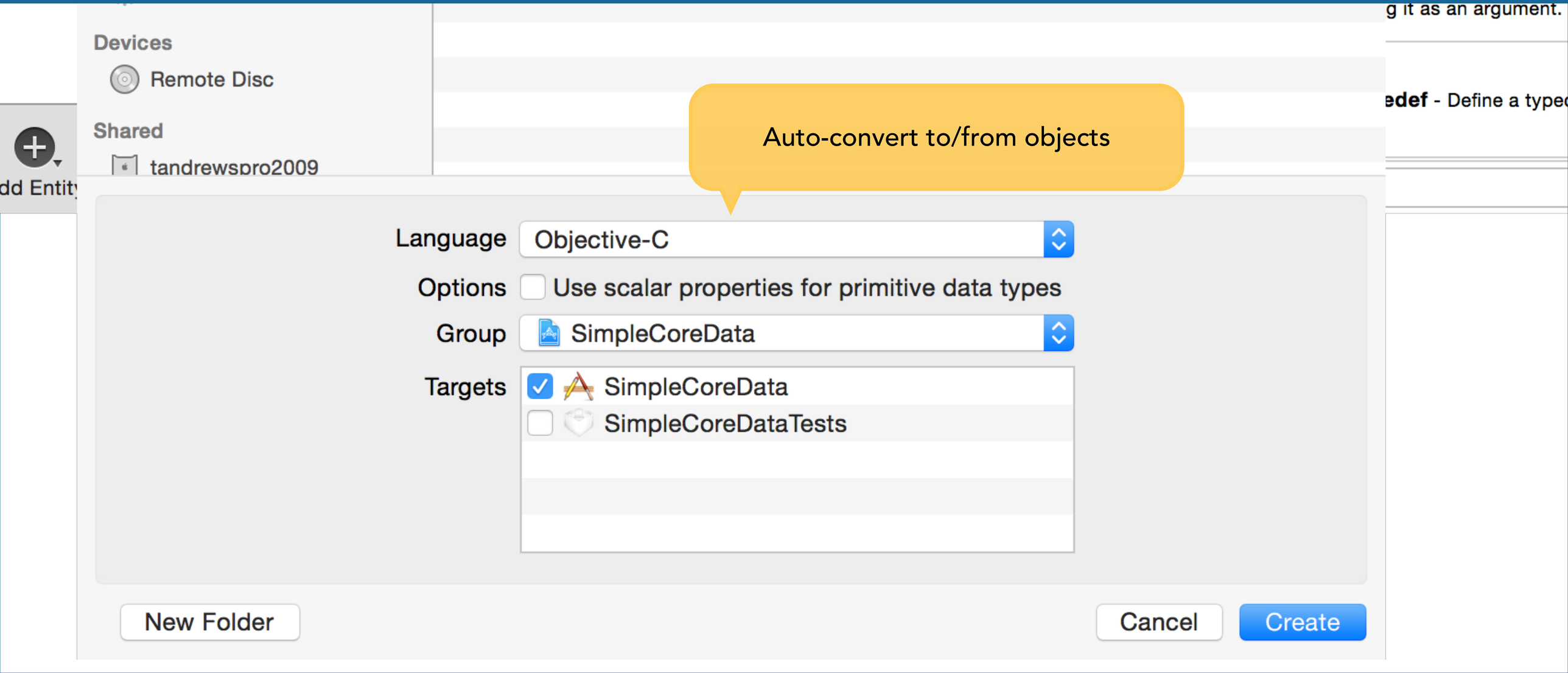
NSMANAGED OBJECT SUBCLASS



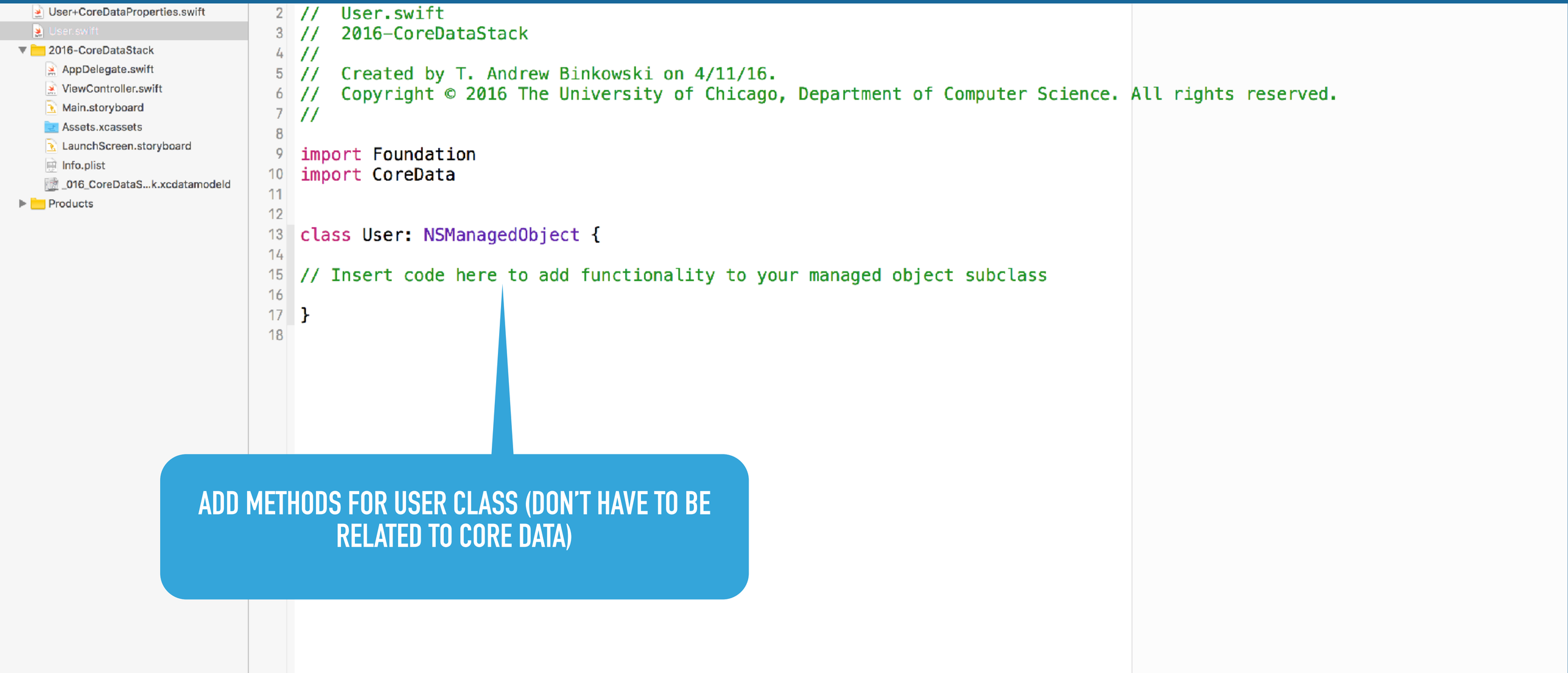
NSMANAGED OBJECT SUBCLASS



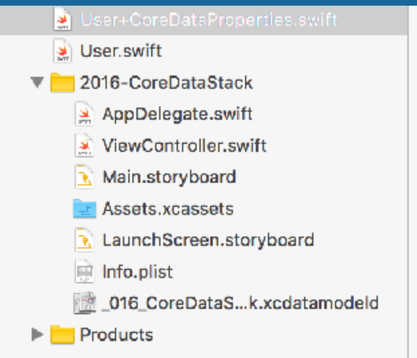
NSMANAGED OBJECT SUBCLASS



NSMANAGED OBJECT SUBCLASS



NSMANAGED OBJECT SUBCLASS

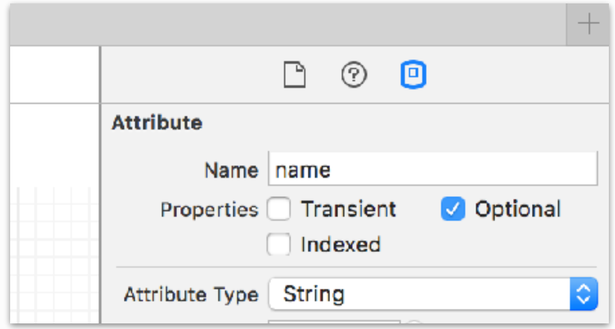



```swift
//  User+CoreDataProperties.swift
//  2016-CoreDataStack
//
//  Created by T. Andrew Binkowski on 4/11/16.
//  Copyright © 2016 The University of Chicago, Department of Computer Science. All rights reserved.
//
//  Choose "Create NSManagedObject Subclass…" from the Core Data editor menu
//  to delete and recreate this implementation file for your updated model.
//

import Foundation
import CoreData

extension User {

    @NSManaged var name: String?
    @NSManaged var sessions: NSSet?

}
```

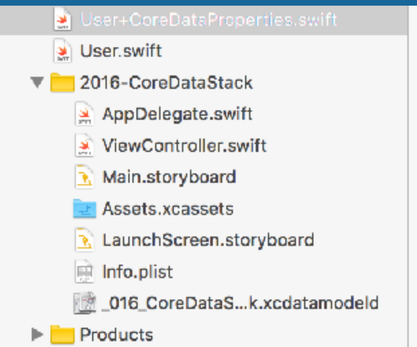
NSSet for relationships

Extension on the User class

Optionals

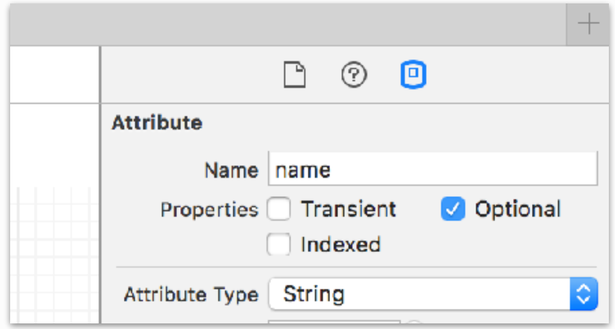


NSMANAGED OBJECT SUBCLASS




```swift
//  User+CoreDataProperties.swift
//  2016-CoreDataStack
//
//  Created by T. Andrew Binkowski on 4/11/16.
//  Copyright © 2016 The University of Chicago, Department of Computer Science. All rights reserved.
//
//  Choose "Create NSManagedObject Subclass…" from the Core Data editor menu
//  to delete and recreate this implementation file for your updated model.
//

import Foundation
import CoreData

extension User {

    @NSManaged var name: String?
    @NSManaged var sessions: NSSet?

}
```

Only generate this one time



NSMANAGED OBJECT SUBCLASS

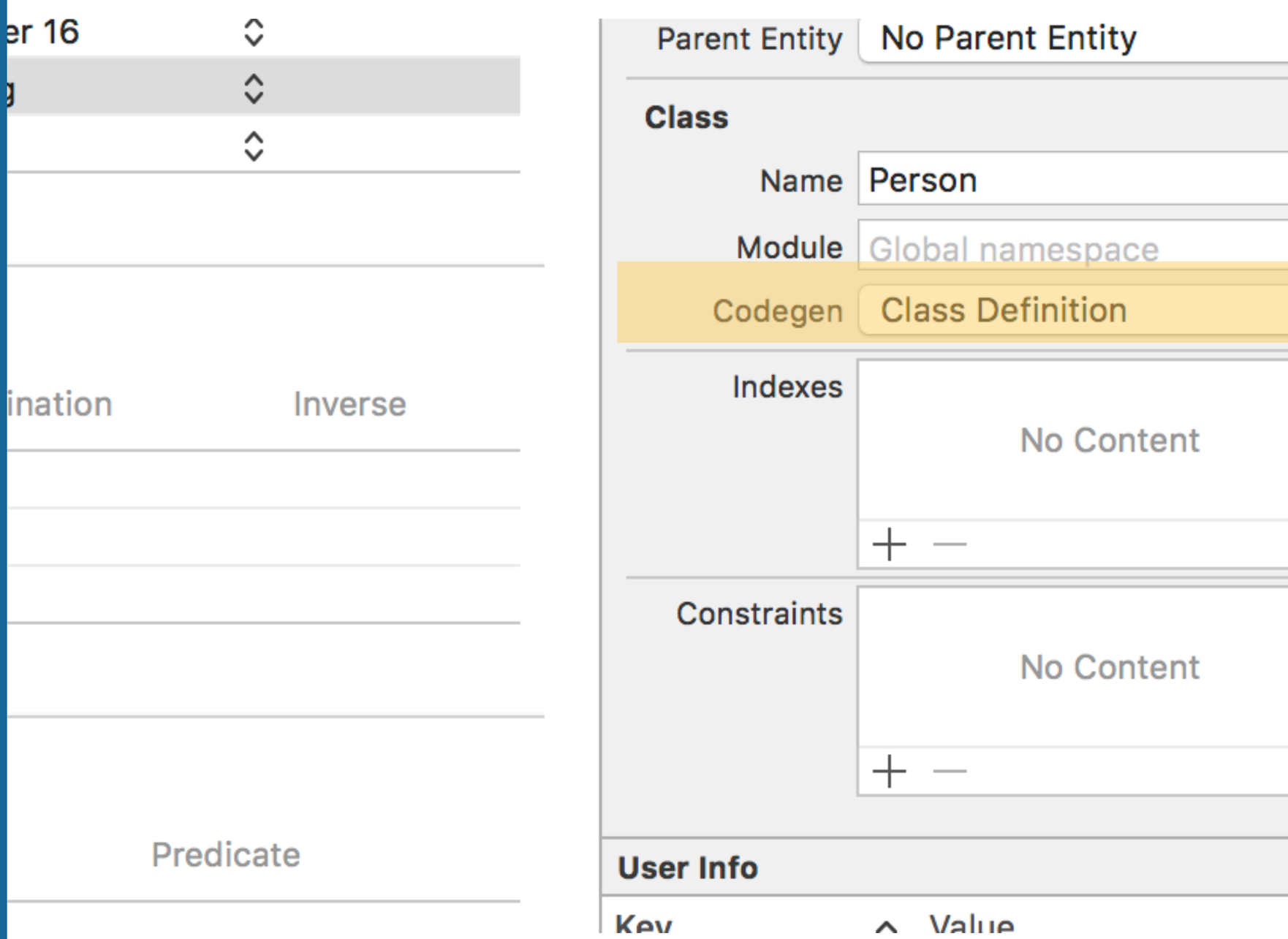
```
17 13:04:52.939 MyFavoriteApps[2503:fb03] Unresolved error Error Domain=NSCocoaErrorDomain Code=134100 "The operation couldn't be completed. (Cocoa e
' UserInfo=0x6b63560 {metadata=<CFBasicHash 0x6b67b20 [0x173fb48]>{type = immutable dict, count = 7,
=>
<CFString 0x6b67ee0 [0x173fb48]>{contents = "NSStoreModelVersionIdentifiers"} = <CFArray 0x6b68340 [0x173fb48]>{type = immutable, count = 1, values =
<CFString 0x173acd8 [0x173fb48]>{contents = ""}

<CFString 0x6b67f10 [0x173fb48]>{contents = "NSPersistenceFrameworkVersion"} = <CFNumber 0x6b67dd0 [0x173fb48]>{value = +386, type = kCFNumberSInt64T
<CFString 0x6b682e0 [0x173fb48]>{contents = "NSStoreModelVersionHashes"} = <CFBasicHash 0x6b683c0 [0x173fb48]>{type = immutable dict, count = 1,
=>
<CFString 0x6b68360 [0x173fb48]>{contents = "Event"} = <CFData 0x6b68370 [0x173fb48]>{length = 32, c
46d30e7f32c2cc809958add1e7 ... 846e97d7af01cc79}


<CFString 0x10f8ad8 [0x173fb48]>{contents = "NSStoreUUID"} = <CFString 0x6b680c0 [0x173fb48]>{conter                                    "}
<CFString 0x10f8978 [0x173fb48]>{contents = "NSStoreType"} = <CFString 0x10f8988 [0x173fb48]>{conter
<CFString 0x6b67d90 [0x173fb48]>{contents = "_NSAutoVacuumLevel"} = <CFString 0x6b68420 [0x173fb48]>
 <CFString 0x6b68310 [0x173fb48]>{contents = "NSStoreModelVersionHashesVersion"} = <CFNumber 0x6b229                                    erSInt3

=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 58add1e7 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



NSMANAGEDOBJECT SUBCLASS IN IOS10

- Thats it 😀

THE UNIVERSITY OF CHICAGO

WATCH APPLICATION DEVELOPMENT

MPCS 51032 • SPRING 2020 • SESSION 5