Getting to know Core Data

Whitney Young

FadingRed
@wbyoung

wbyoung.github.com

Overview

Where it came from

What it is & how to use it

How to use it well

Who uses Core Data in a production app?

History

NeXT DBKit 1992

Enterprise Objects Framework 1994

Core Data 2005

Core Data
- Mac OS X 10.4 Tiger 2005: Introduced
- Mac OS X 10.5 Leopard 2007: Migration
- Mac OS X 10.7 Lion 2011: Concurrency

What it is...

What it is not...

Core Data is *not* a database

What it is...

MVC

What it is...

Model

View

Controller

What it is...

Object Graph Persistence Framework

Core Data Design

Flexibility, clarity

Harder to set up, verbose

`NSPersistentStoreDidImportUbiquitousContentChangesNotification`

Access data transparently from multiple files
Create your own storage backend
  NSAtomicStore
  NSIncrementalStore (could, for instance write a backend for CouchDB)

Setup

Template projects

NSManagedObjectModel, NSManagedObjectContext

NSPersistentStoreCoordinator, NSPersistentStore

Setup

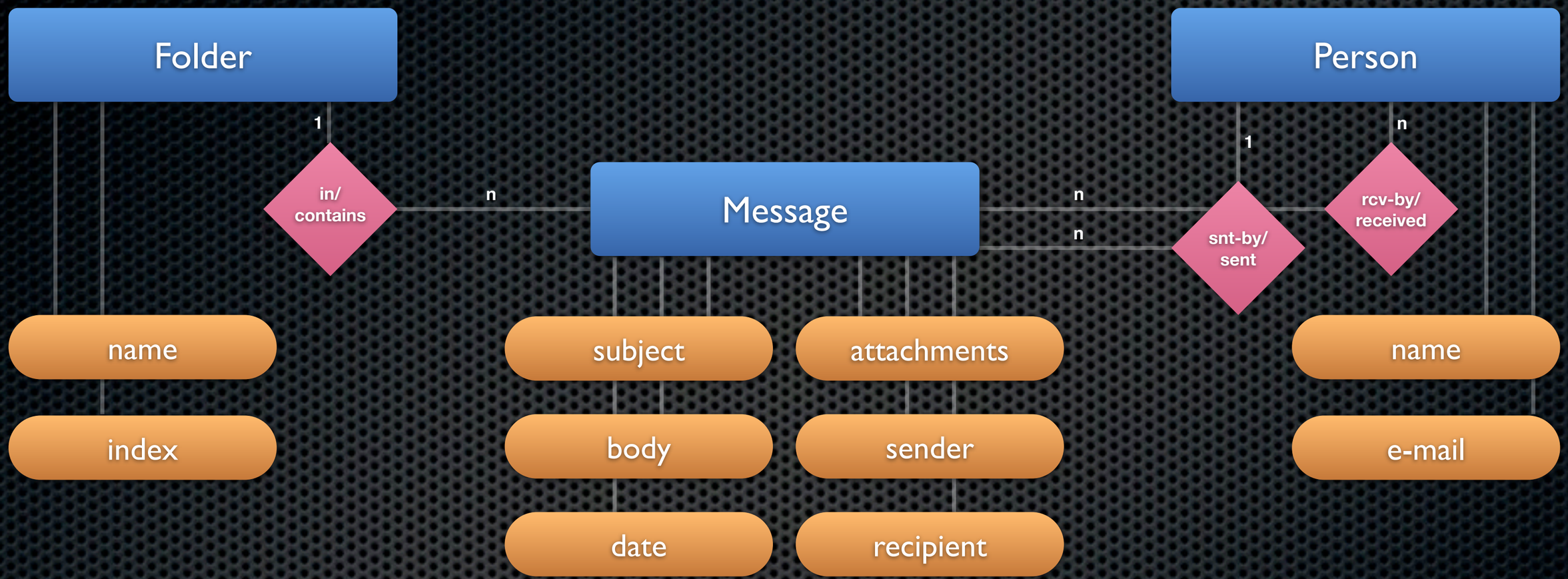
Template projects

Model, Context

Coordinator, Store

Example Application

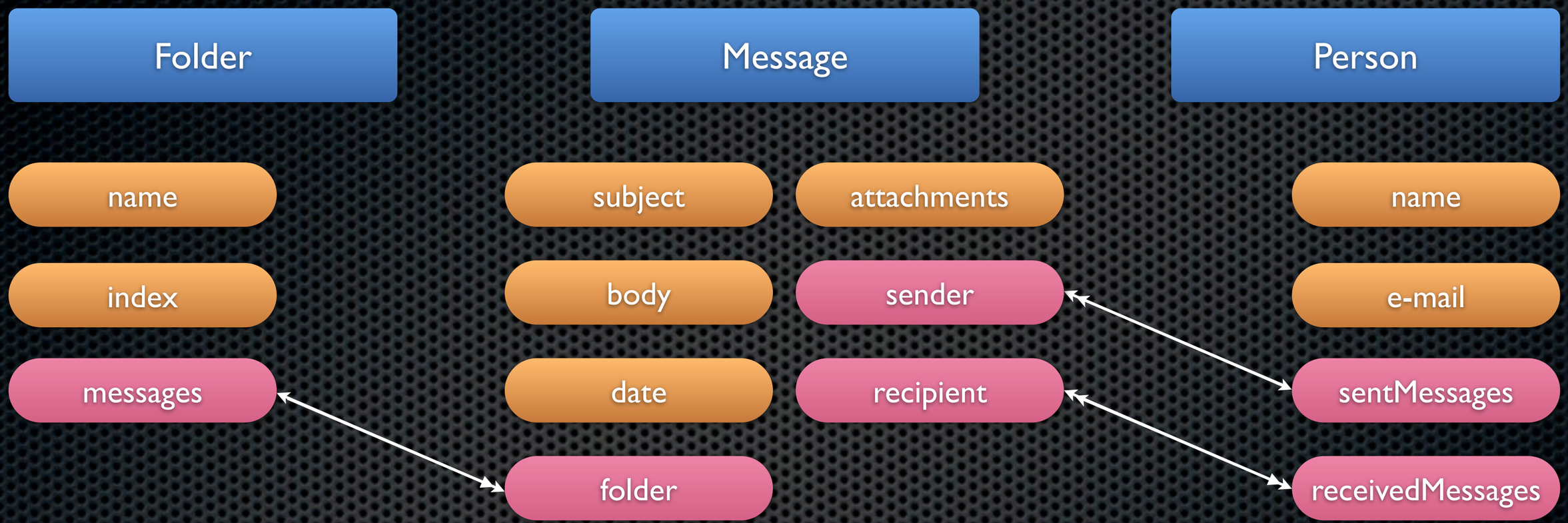
E-mail



Example Application

E-mail

Entity relationship diagram

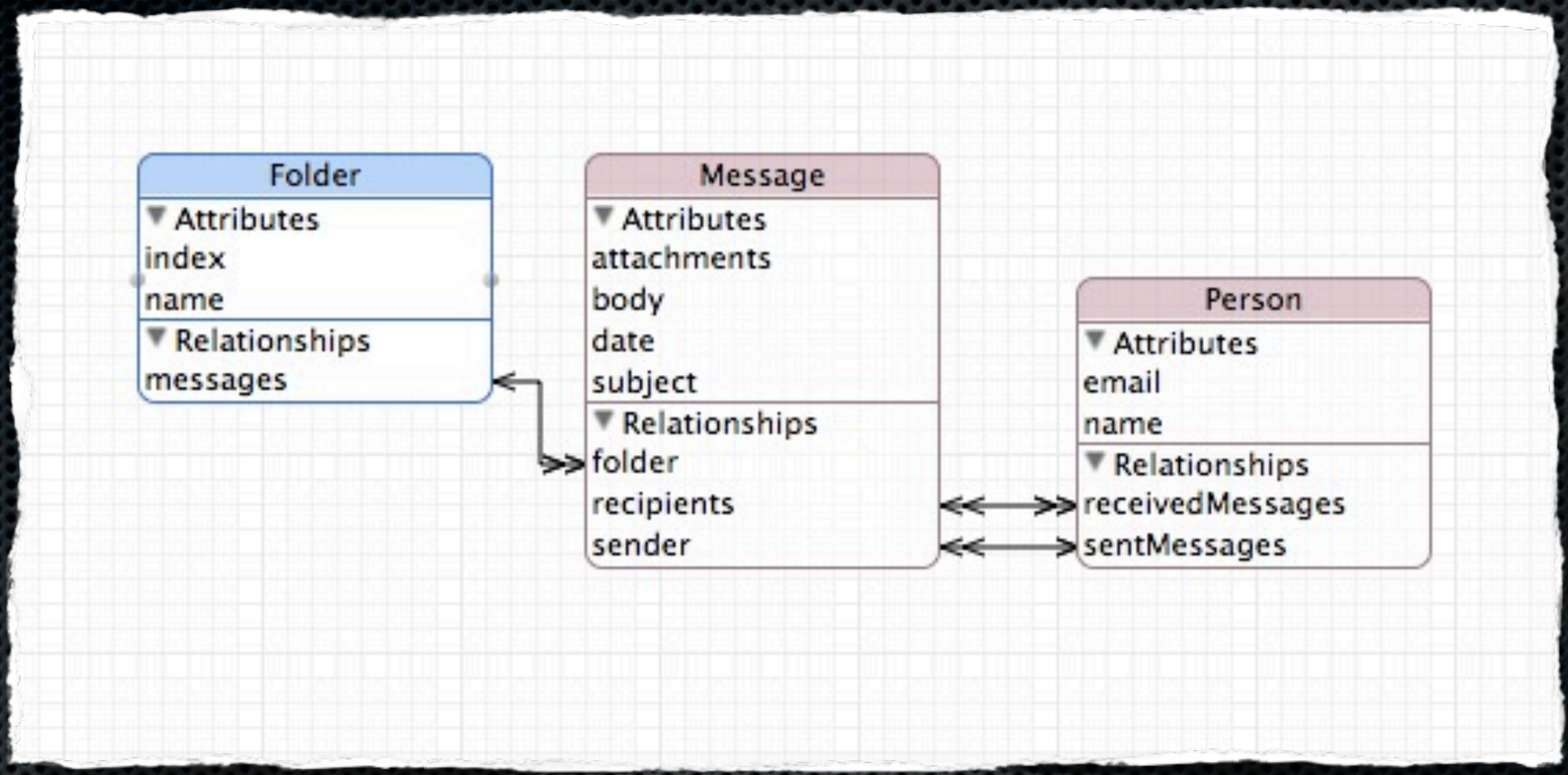


Example Application

E-mail

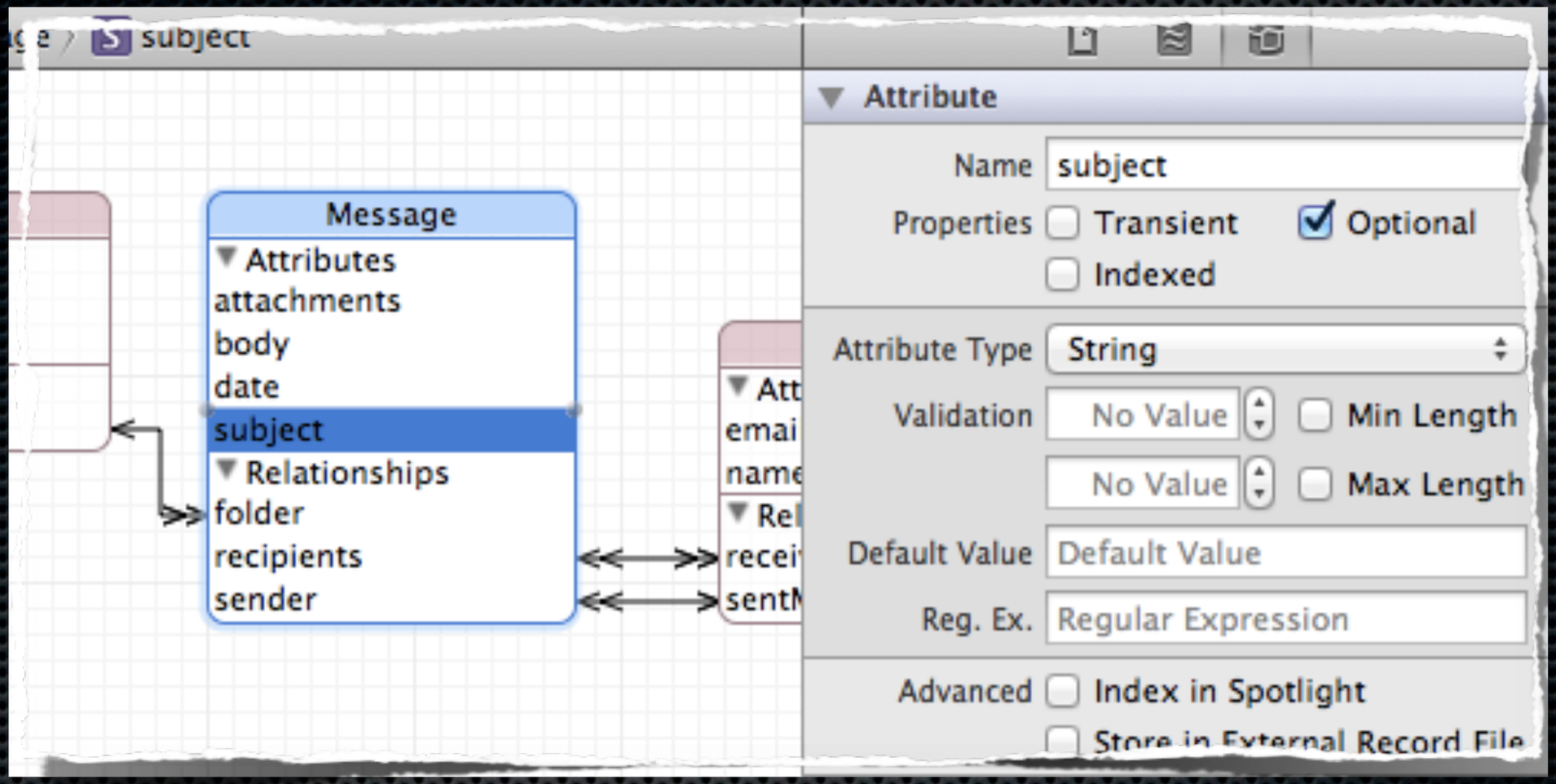
Inverse relationships are required

You get a warning for missing inverses, but like all warnings, this isn't something to ignore



Modeling in Xcode

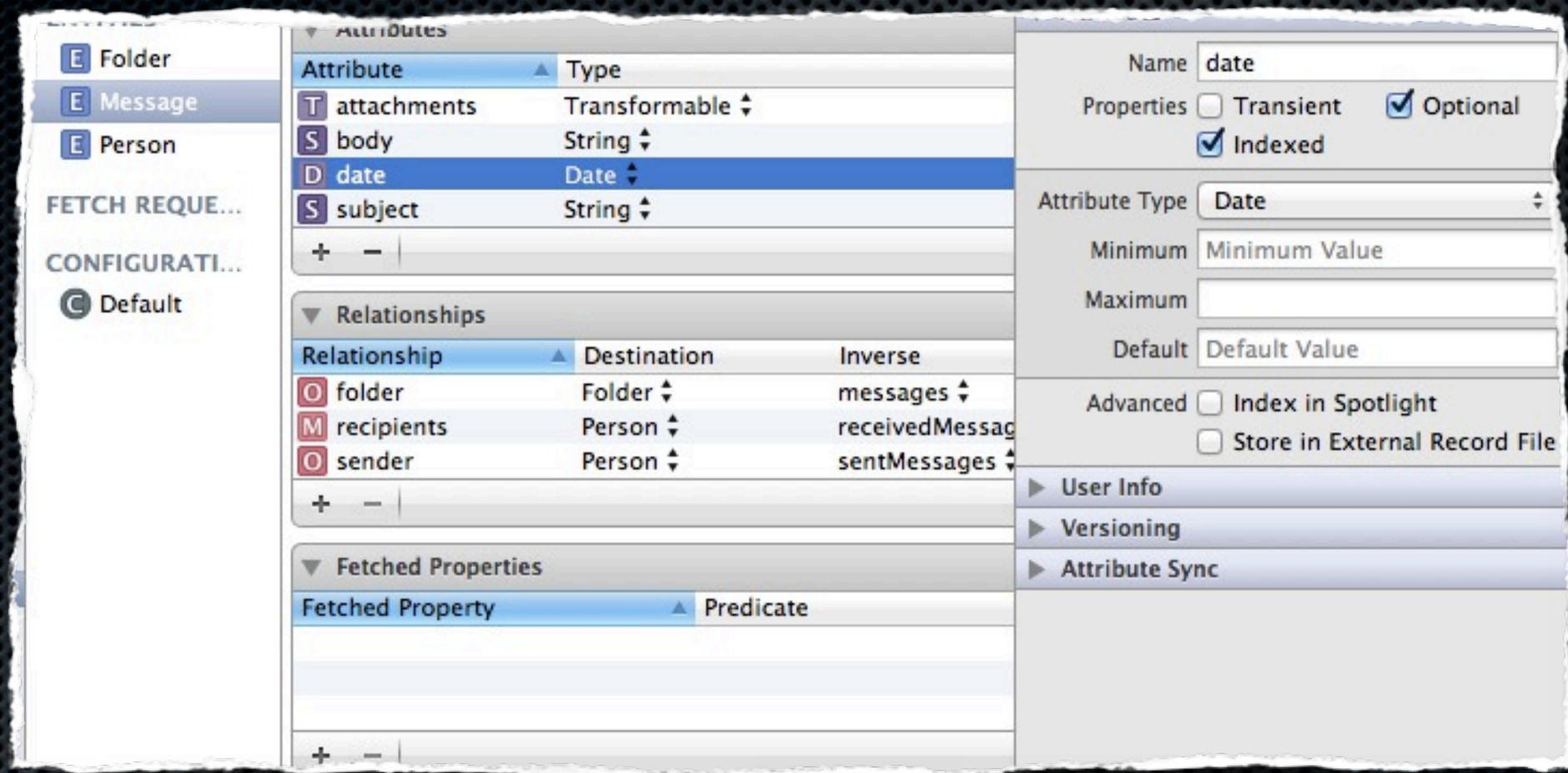
Building a managed object model



Modeling in Xcode

Editing Attributes

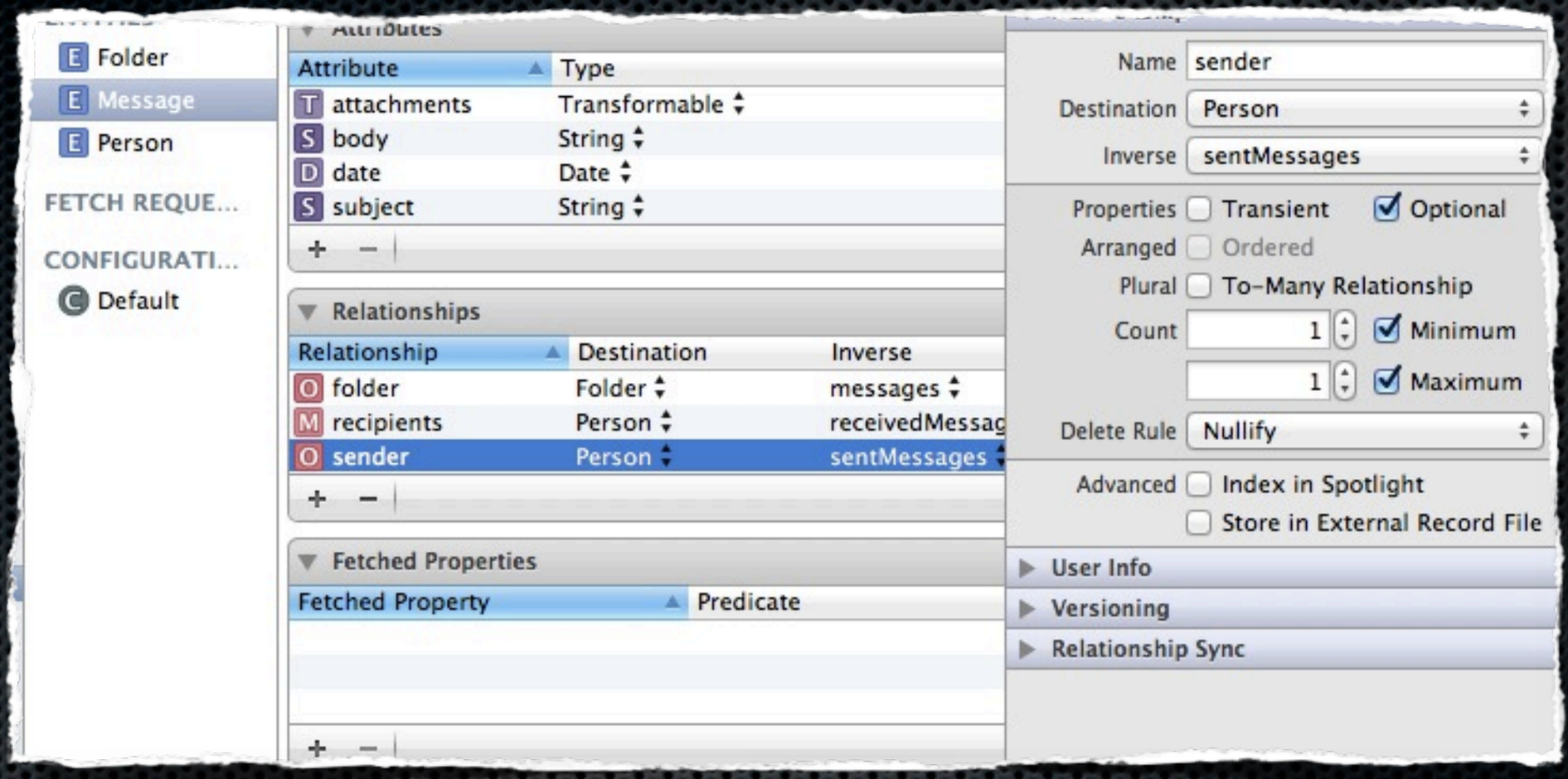
Note the validation tools



Modeling in Xcode

Editing Attributes

Different data view
Different attribute has different validation tools
Data types include: integer, decimal (base 10), floating point, string, boolean, date, binary, transformable



Modeling in Xcode

Editing Relationships

This is the 1 side of a 1-to-many relationship
Also can handle many-to-many and 1-to-1 relationships
Ordered sets exist, but don't use them

Modeling in Xcode

Design a xcdatamodel(d)

Compiled to mom(d)

Deserialize to NSManagedObjectModel

```
[NSManagedObjectModel mergedModelFromBundles:nil]
```

Passing nil looks in the main bundle for compiled models

Working with data

NSManagedObjectContext

In memory

Insert, update, delete

Assume, for now, that the context exists
Contexts are not thread safe

Creating Objects

Pretty simple

```
[[NSManagedObject alloc] initWithEntity:entity
          insertIntoManagedObjectContext:context];
```

Creating Objects

Wait, what entity?

```
[[NSManagedObject alloc] initWithEntity:entity
                insertIntoManagedObjectContext:context];
```

Entities

Already created in the model

```
[NSEntityDescription entityForName:@”Person”
                inManagedObjectContext:context];
```

Updating Objects

Use Key Value Coding

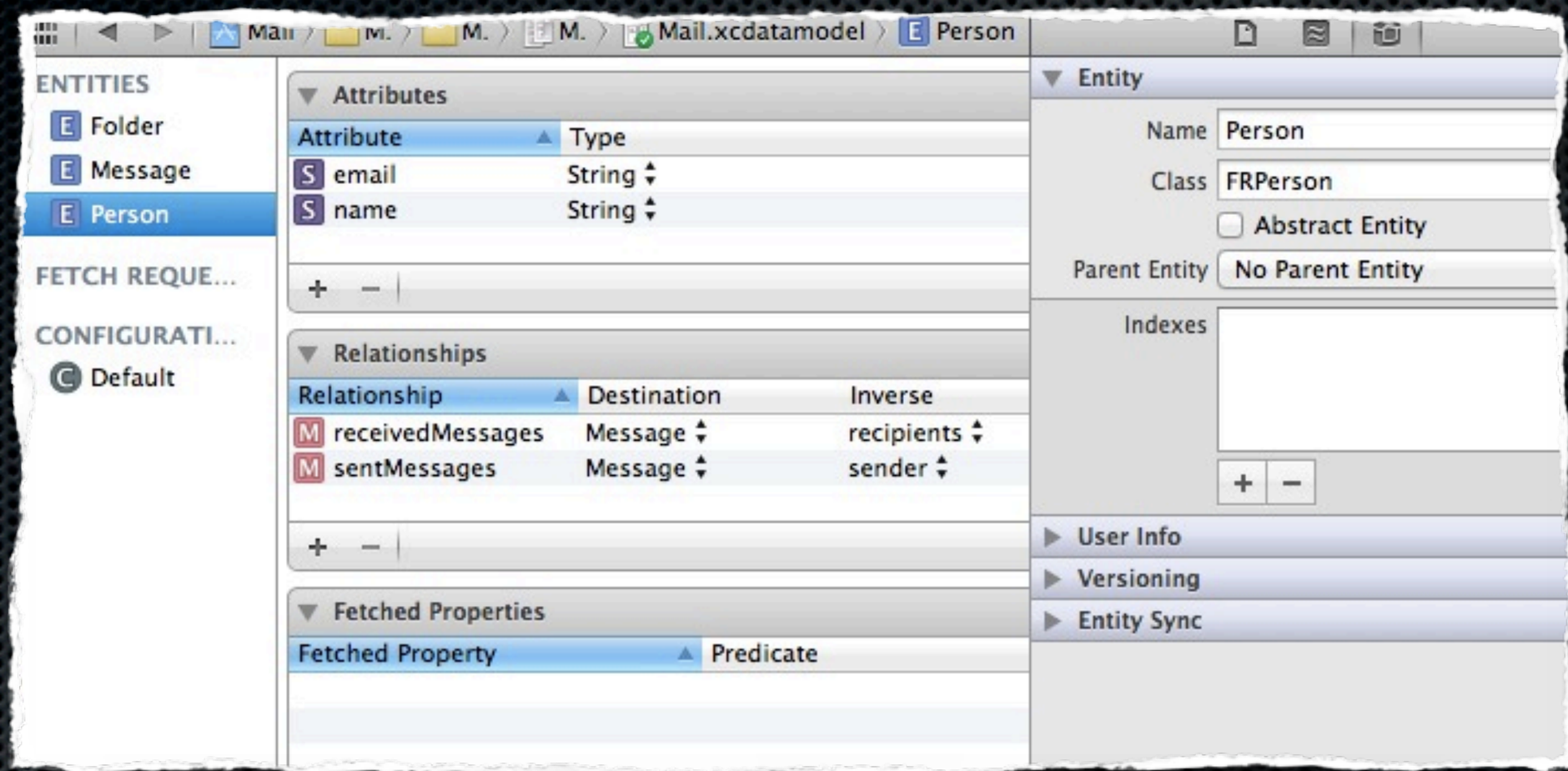
```
NSManagedObject *person = …;
[person setValue:@”Whitney” forKey:@”name”];
[person setValue:@”wbyoung@fadingred.com”
          forKey:@”email”];
```

This is clunky & the compiler won’t see your typos

Updating Objects

Use custom classes

```
FRPerson *person = …;
person.name = @”Whitney”;
person.email = @”wbyoung@fadingred.com”;
```



Custom classes

Model setup

Custom Classes

Interface

```
@interface FRPerson : NSManagedObject
@property (nonatomic, retain) NSString *name;
@property (nonatomic, retain) NSString *email;
@end
```

Custom Classes

Interface for relationships

```
@interface FRPerson (CoreDataGenerated)
- (NSSet *)sentMessages;
- (void)addSentMessagesObject:(FRMessage *)m;
- (void)removeSentMessagesObject:(FRMessage *)m;
@end
```

Same thing for received messages
Can also define:

```
- (void)addSentMessages:(NSSet *)messages;
- (void)removeSentMessages:(NSSet *)messages;
```

Can implement these as well to customize, but must implement both the add & remove for each type for Core Data to use them

Custom Classes

Implementation

```
@implementation FRPerson
@dynamic name, email;
@end
```

No implementation
- Dynamic means that it will be created at runtime
- No implementation for relationships

Core Data generates all the methods for you dynamically at runtime

Custom Classes

Custom accessors

Validations

Transient attributes

mogenerator

Customizing accessors allows your model to be a bit smarter
You can do complex validations that aren't possible in the model designer
You can define transient attributes to combine or store data that isn't a built-in type
If you create a lot of entities, consider mogenerator (Managed Object Generator) to save you time

Fetching Objects

Use Fetch Requests

```
NSFetchRequest *request = [[NSFetchRequest alloc] init]];
[request setEntity:messageEntity];
[request setPredicate:predicate];
```

Fetching Objects

Predicates (not SQL)

```
[NSPredicate predicateWithFormat:
 @”subject contains[cd] %@”, searchTerm];
```

This is not SQL: you don’t specify joins & aggregates in the same way
Can’t use block predicates for SQLite stores

[cd] is a modifier that indicates that case (c) and diacritics (d) should be ignored when comparing

Fetching Objects

Key paths for relationships

```
[NSPredicate predicateWithFormat:
 @”sender.name beginswith[cd] %@”, searchTerm];
```

Fetching Objects

Matching collections

```
[NSPredicate predicateWithFormat:
 @”sender in %@”, collectionOfPeople];
```

The collection can be a collection of Managed Objects or Managed Object IDs

Fetching Objects

Lots of possibilities

```
[NSPredicate predicateWithFormat:
 @"SUBQUERY(recipients, $r, “
 @”  $r.name = %@ and “
 @”  $r.email = %@) != 0", rName, rEmail]
```

This will probably be slow & you should avoid it.

For comparison, consider this query which is a bit different:

```
ANY recipients.name = %@ or ANY recipients.email = %@
```

The subquery is matching both name & email on an individual recipient — both must match on one person. The ANY version checks if any of the recipients have the name or email — either one must match on any one person. The subquery is slow because it’s evaluated in SQLite as an actual subquery. The predicate using ANY is evaluated as a join & is much faster, but does return different results. See NSExpression for subquery information.

Deleting Objects

```
[context deleteObject:object];
```

Nullify, Cascade, Deny, No action

No action is not recommended as it could leave your data in an inconsistent state
Deletion is propagated at the end of the run loop (or the next call to processPendingChanges)

Saving Changes

All changes are in memory until…

```
[context save:&error];
```

Storage

NSPersistentStore

SQLite, Binary, XML, Memory

SQLite is what you'll usually be using
Binary is smaller, but is atomic (whole file will be read to get data — good for small data sets)
XML is not available on iOS & is generally recommended only for development
The memory store type will not be persisted (good for unit testing)

Coordination

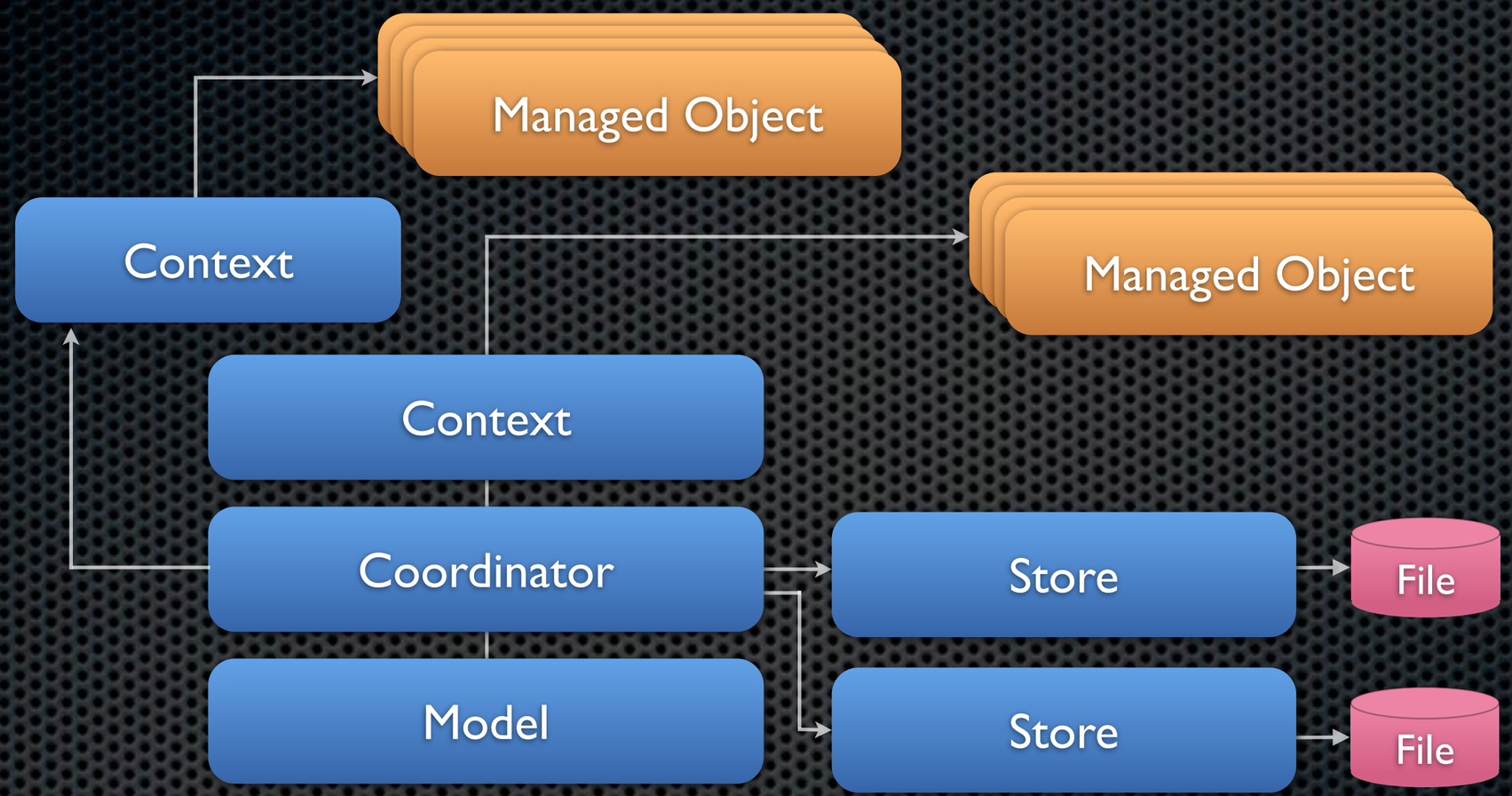
NSPersistentStoreCoordinator

```
[[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:mom];
[context setPersistentStoreCoordinator:coordinator];
```

Context saves through coordinator

Tied to model & store

Coordinator allows thread safety, locking to avoid read/write concurrency issues



How it all connects

The Core Data stack

Full stack setup:

```
NSURL *storeURL = nil; /* set this to something in the documents directory, for example */
NSManagedObjectModel *model = [NSManagedObjectModel mergedModelFromBundles:nil];
NSPersistentStoreCoordinator *coordinator = [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:model];
[coordinator addPersistentStoreWithType:NSSQLiteStoreType configuration:nil URL:storeURL options:nil error:NULL];
NSManagedObjectContext *context = [[NSManagedObjectContext alloc] initWithConcurrencyType:NSMainQueueConcurrencyType];
[context setPersistentStoreCoordinator:coordinator];
```

Fetch Performance

Objects returned as faults

Accessing data causes a fault to fire

```
-com.apple.CoreData.SQLDebug 1
```

When a fault fires, Core Data may go to the disk to fulfill the fault which is slow

Fetches are set up to fill the SQLite row cache by default (see includesPropertyValues) which basically means that Core Data has a memory cursor, so firing faults on objects that have been returned through a fetch request is fast. Firing faults on objects that have been obtained through relationships, though, will likely hit the disk.

Fetch Performance

Fetch batch size

Batch faulting

Prefetching relationships

Asynchronous fetching

Batch size: note that if context has changes in it, many fetch options that could tune performance may work against you (like batch size)… consider setIncludesPendingChanges:NO

Asynchronous fetching is not implemented in Core Data, but is not too difficult to do

Concurrency

Contexts & objects cannot cross threads

Pass object IDs between threads

Concurrency

Context concurrency types

Confinement, private queue, main queue

Nested Contexts

```
[context performBlockAndWait:^{
}];
```

Using nested contexts is not always a suitable option (yet) for performing concurrent work. Reads block ancestor contexts while retrieving data. A parent can work asynchronously on something while a child continues, but if a child tries to work asynchronously on something while the parent continues, it will block the parent if it tries to read data from the persistent store.

Miscellaneous

Inheritance (parent entities)

De-normalization

iCloud

Inheritances flattens all descendent entities into one table which can affect performance

De-normalization could help you increase performance

iCloud with Core Data has generally not been working for people. Apple has been fairly unresponsive. Those who have gotten it working still have very slow syncing.