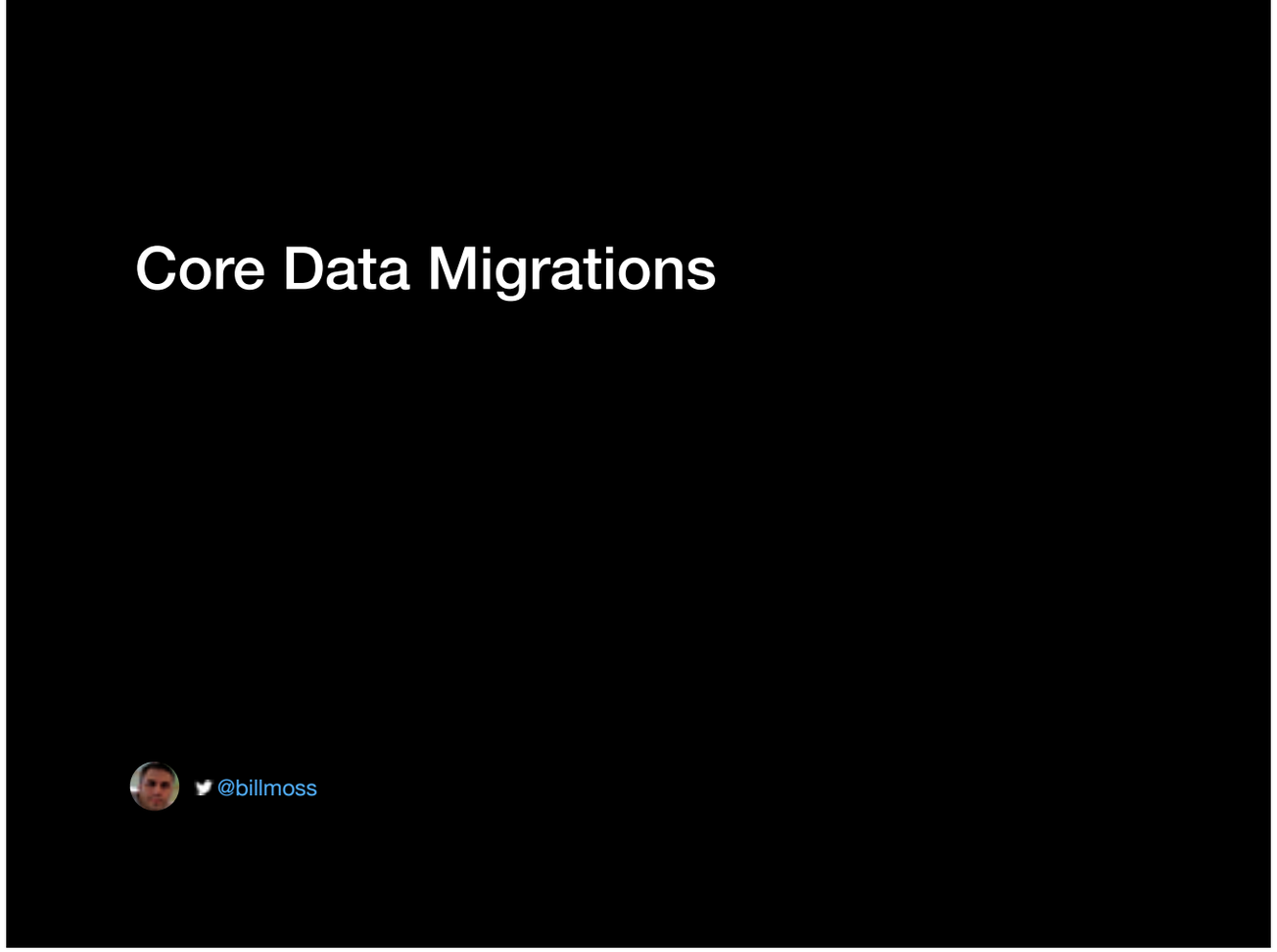


iOS applications follow a MVC design. Core Data is a Model Layer Technology from Apple that you can use to help build out that model component of your app. With Core Data you create the schema for your model, create and edit model objects in memory and then persist them to a database or a file store.

Migration is the process of updating a database or file store from one version of model schema to another.

Topics

- Role of Migration
- How to Get Started
- Options and Process
- Customizing with Mapping Models and Policy Code
- Progressive Migration
- Tips and Resources

Core Data Models

When you create a Core Data app, you typically begin with thinking about your Data Model.

What is Core Data?

A model layer technology for your app.

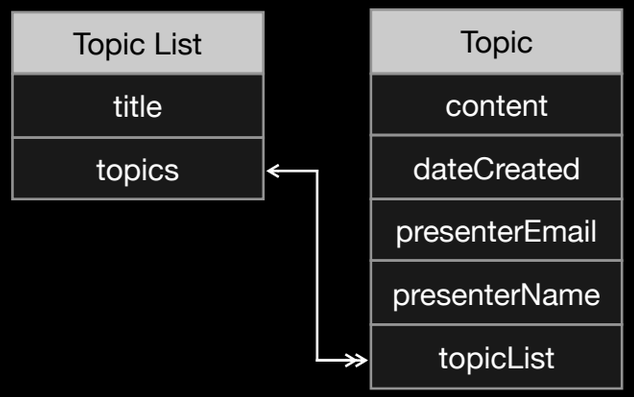
- Define entities with properties and relationships.
- Manages an in-memory object graph.
- Persist objects in SQLite, XML, and binary files.
- Sophisticated queries, KVC, and KVO.
- Change tracking and undo support.
- Property and object level validation.
- Schema migration.

Core Data is a great solution for building out your Model layer.
Other features include Merge policies, grouping, etc.
Core Data's schema migration features are the focus of this talk.

Reference: https://developer.apple.com/library/ios/documentation/cocoa/conceptual/coredata/articles/cdTechnologyOverview.html

Core Data Model

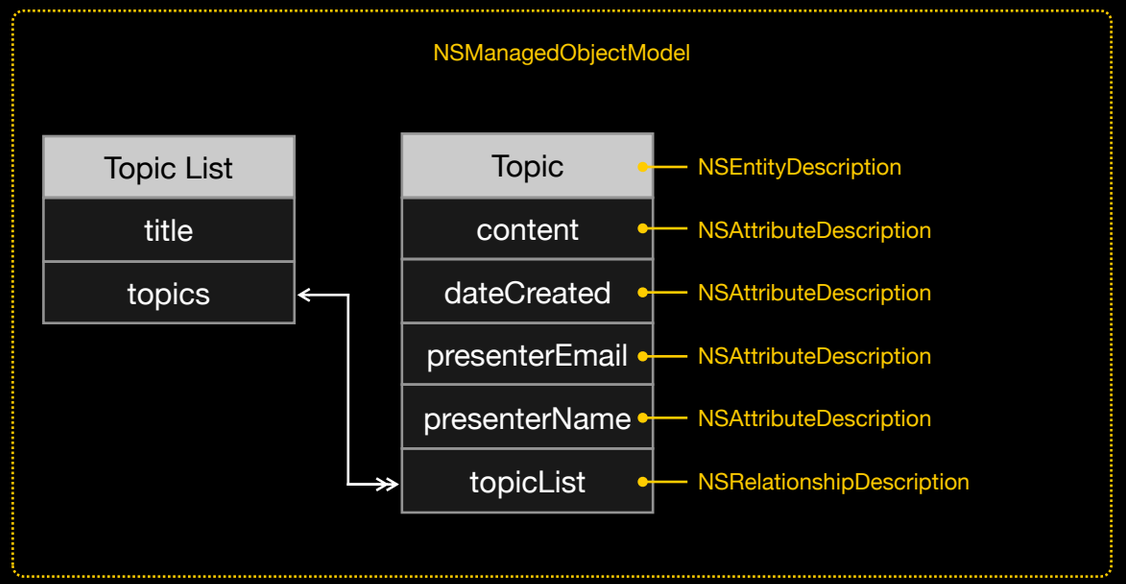
A simple model for our demo app.



Here is a simple model for a topics management app

Core Data Model

A simple model for our demo app.



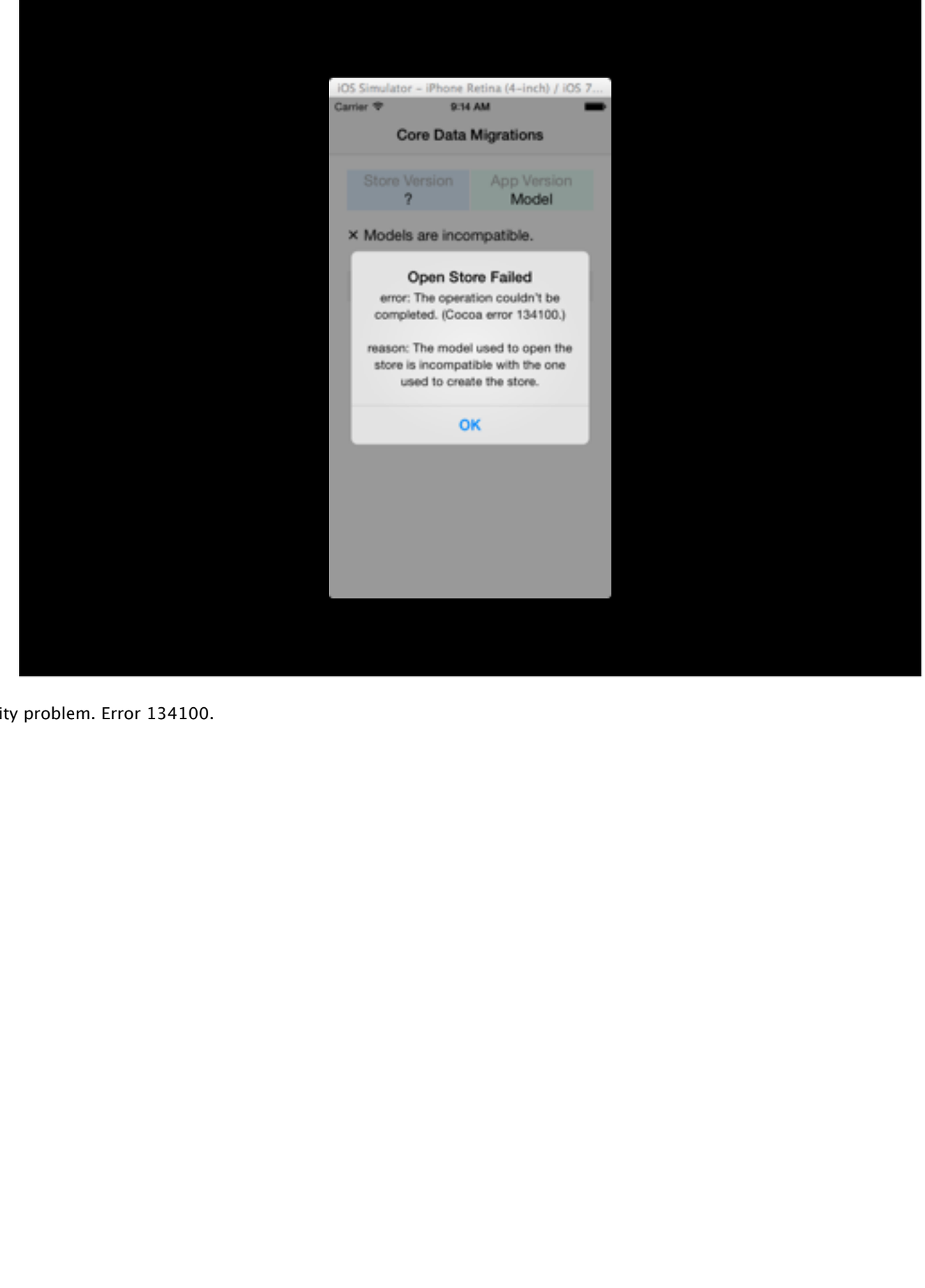
Here is a simple model for a topics management app.

In Core Data terms, the model is represented as a collection of Entities.
Entities are composed of attributes and relationships to other Entities.

When we refer to the object model or the model, this is what we are speaking about. The Managed Object Model.

Demo
A Simple Core Data App

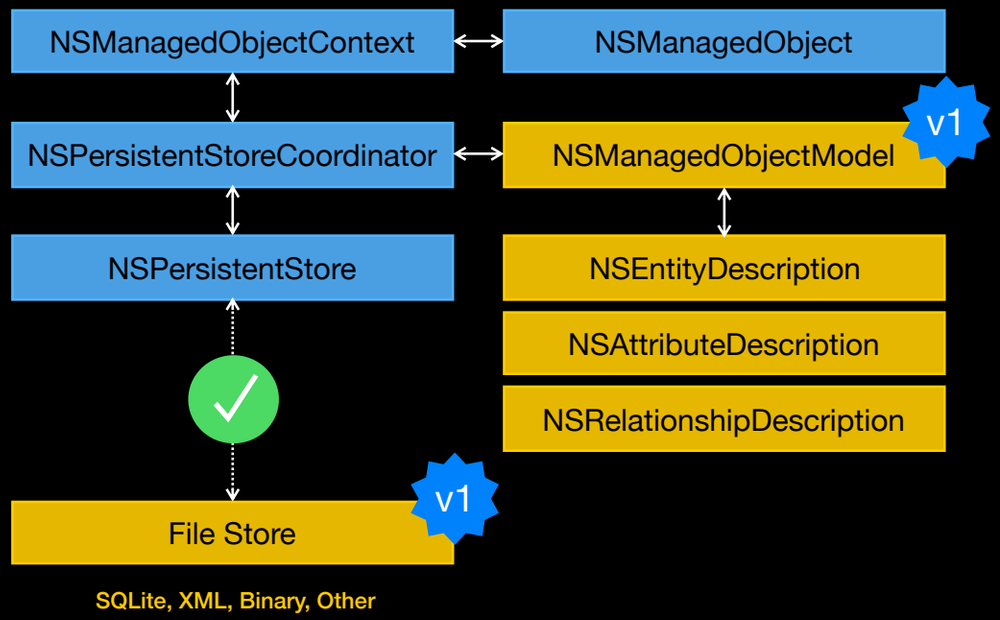
Demo 1 No Migration
Introduce our demo Core Data App. Model Editor. Core Data Stack. Managed Object Model.
Run the app to generate a sample database of topicLists and topics.
Quit the app. Modify the model by adding a new timeBudget attribute to Topic.
Set it as a Integer32 type, Required, and a Default of 5 (minutes).
Run and show the compatibility problem. Open the store and display the Core Data error.



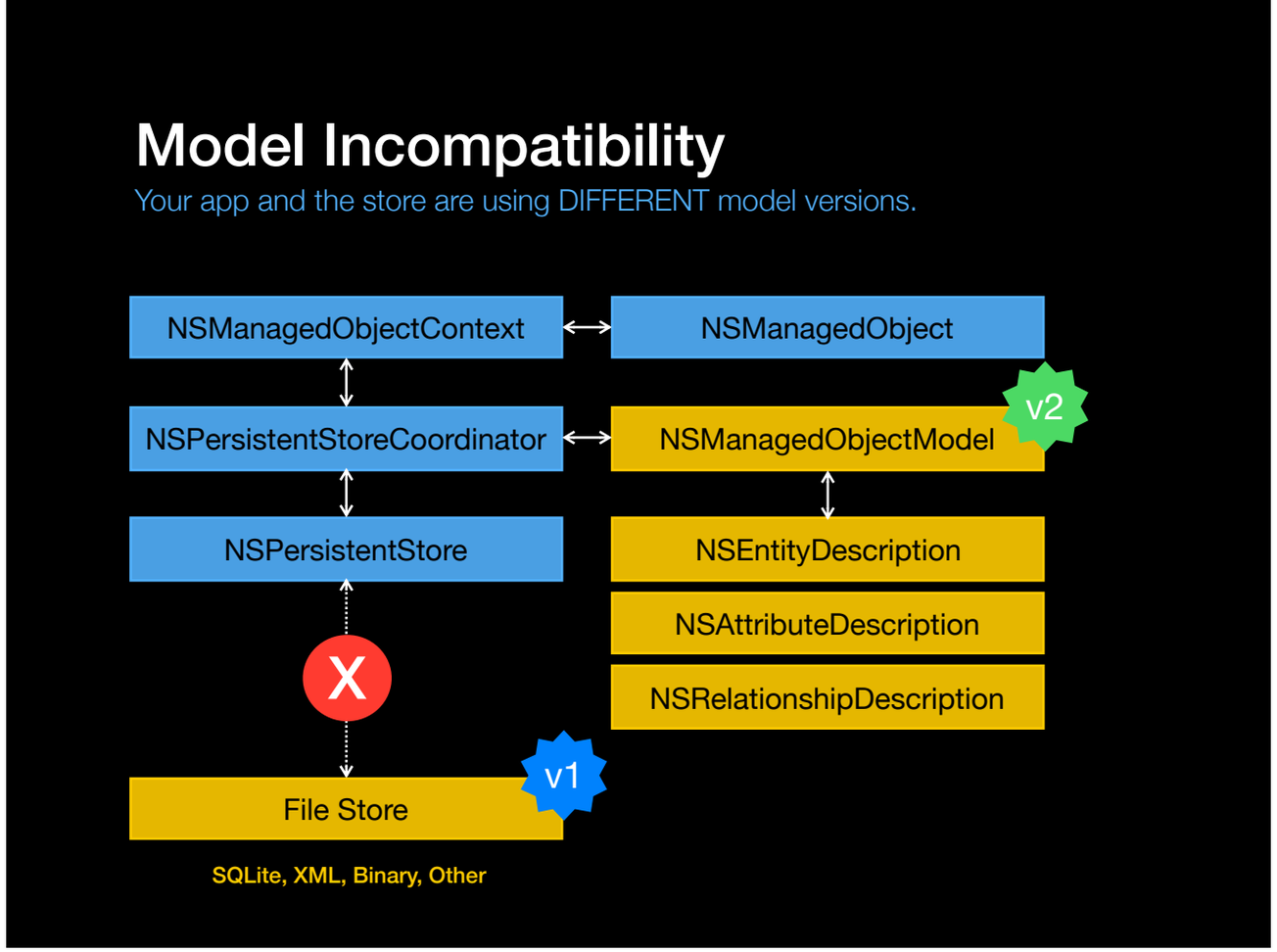
Model changes caused this Core Data compatibility problem. Error 134100.

Model Compatibility

Your app and the store are using the SAME model version.



When we first ran the app, the app model was used to create the SQLite database.
The app model and the store model are the same. This is the condition Core Data requires for accessing a store…the app model and the store model must be compatible.



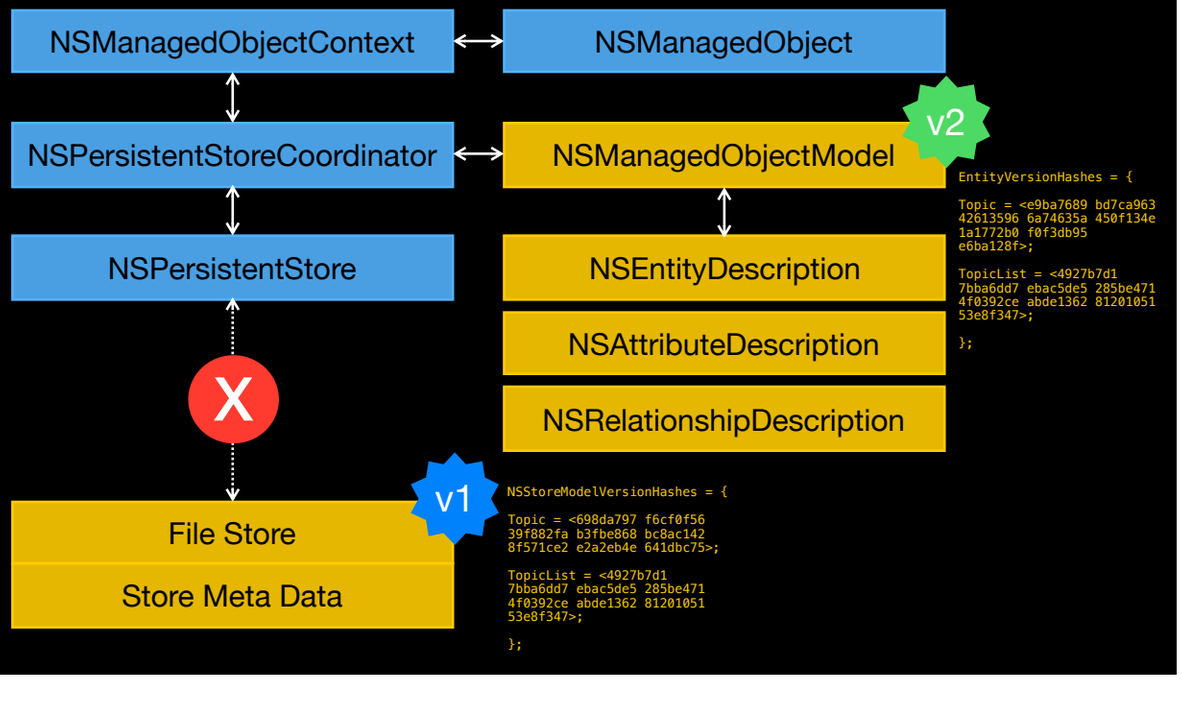
By adding a new attribute to the model, we effectively created a different model for the app to use. We can call this version 2.
However, the File Store was created with the model prior to our change.
You can only open a Core Data store using the managed object model used to create it.
If you change your model, then you need to change the data in existing stores to new version.

—changing the store format is the role of **migration**.

How is this test of compatibility and version comparison implemented?

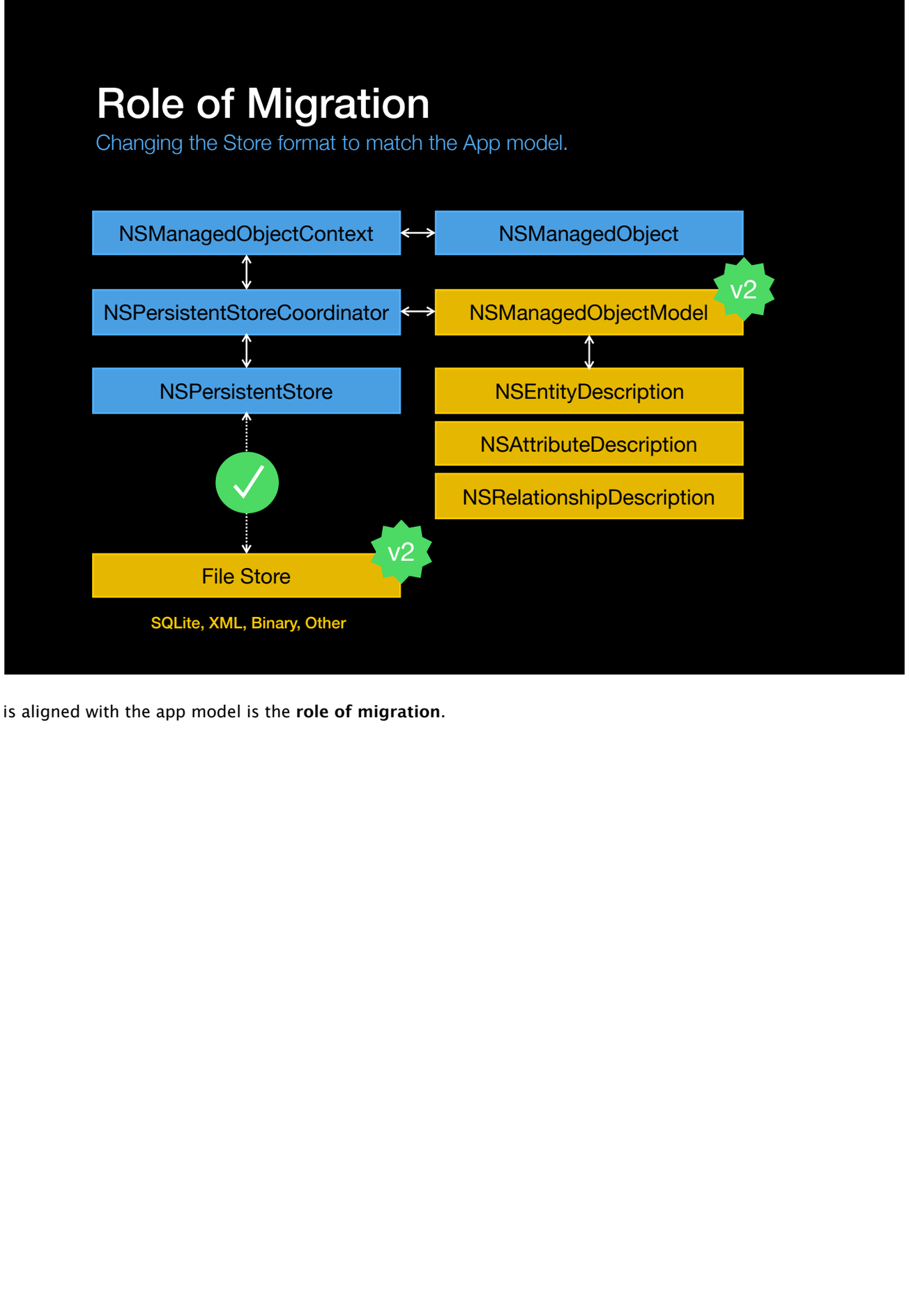
Compatibility Test

Entity Version Hashes are compared between the Store Model and App Model.



When Core Data creates a store, it is formatted according to the model it was created with. Core Data also includes meta data within the store to manage the version information. This version information is the collection of Entity Version Hashes from the ManagedObjectModel.

Core Data creates a unique hash for each attribute, relationship, and entity using the persistent elements of the model. When opening the store, Core Data will compare the Entity Version Hashes of the app model to the hashes stored within the Store's Meta Data. If they are equal, the app and the store are compatible. In our demo, we changed the app model which caused the Entity Version Hashes of the app model to be changed and no longer equal to the store's version.



Changing the store format (schema and data) so it is aligned with the app model is the **role of migration**.

Incompatible Model Changes

Any change that affects persistence is a compatibility problem.

- Add, delete, rename entity, attribute, or relationship
- Attribute type. ex. Integer 32 -> String, Integer 32 -> Integer 64
- Relationship destination, ordinality, count, ordering
- Settings for *optional*, *transient*, and *read only*

These changes affect the version hashes Core Data creates for Entities, Attributes, and Relationships. Essentially, any model element that affects persistence creates a model compatibility problem.

Compatibile Model Changes

These changes have no affect on persistence.

- Your custom Entity Class
- Default values
- Validation predicates
- userInfo dictionaries
- Relationship deleteRule

These changes do not affect the Entity, attribute, or relationship version hashes so they are safe to make without needing a migration.

In my demo app, changing deleteRule on the topics relationship under TopicList did not trigger incompatibility. Apple docs indicate deleteRule change does cause model version change.

Invisible Model Changes

Affect persistence, but go undetected by Core Data.

- Minutes to seconds
- NSValueTransformer implementation
- BLOB data format
- Business logic
- Invisible because they don't alter the *Entity Version Hashes*
- You can set a *Hash Modifier* to trigger incompatibility

These class of changes are invisible to Core Data's version and compatibility mechanism. The reason being that the model file is not changed in any way, so entity version hashes remain unchanged.

For example, we want to change the units of our timeBudget attribute from minutes to seconds example. 5 min -> 300 sec. We do not make a model change for this, rather we set a different value for timeBudget that is now in seconds. Migration is a useful method to deal with this kind of model change.

Developers can use the Hash Modifier to affect Core Data's version hashes and thus trigger incompatibility and migration.

I typically set the Hash Modifier to the version number of my model, but you can set it to any value that makes sense for your app.

Store Migration

So, now that we understand models and changes that affect compatibility, let’s focus on how we can use migration to fix our store/app compatibility problem.

Minimum Requirements

Before you can migrate a store, you need…

- The version of the model file used to create the store.
- Model you want to migrate to. *Your current model file.*
- Enable Core Data migration options.

Core data requires you have the version of the model used to create the store along with the version your app wants to use.

The version you want to migrate to is often the latest version of your model file itself.

There are a number of migration options you can choose. You need to enable one of these options at a minimum to get migration working.

So, how do we manage keeping the older versions of our model?

Model File Versioning

Keeping a version of the object model for each store you want to migrate.

- Not exactly like source control versioning. Done in addition to source control versioning.
- Versioning starts by converting a Core Data model file into a *versioned model file* using Xcode.
- Uniquely name your versioned model files. *Model1*, *Model2*, *Model3* works fine as a naming convention.

Versioning your model is quite simple. Xcode provides the tools for doing this.

Name your model file versions uniquely. Core Data does not actually have any notion of ordering model versions. One model version is only known to be different than another…Core Data does not know which is older or newer. Your naming scheme can fill in this gap.

Migration Options

If you need to support iCloud, you only have one option.

- Automatic Lightweight Migration - Simple, performant, supports iCloud. Done as needed at store opening.
- Default Migration - Handles more complex change scenarios. You provide a *mapping model* and custom code.
- Custom Migration - Similar to Default Migration but employs a NSMigrationManager you initialize.
- Progressive Migration - Solution approach mixing Lightweight and Custom Migration.

Automatic option requires store and app models be locatable in pre-defined location. Same is true for any mapping models.

Default Migration can use the same code setup in your app that is used for Automatic Lightweight Migration.

Default and Custom migration handle more complex change scenarios such as splitting or joining entities and attributes.

The primary difference between lightweight and the other migration options is the developer can have custom code executed in the migration process.

Lightweight Migration

Apple recommended migration process *if* available.

- Core Data must be able to infer a mapping between models.
- Adds and deletes of entities, attributes, and relationships.
- Renaming Entities, Attributes, and Relationships using *Renaming Identifiers*.
- Some attribute type changes. ex. Integer 32 -> double
- Required->Optional, Optional->Required with Default.
- Change relationship ordinality, ordering.
- Parent/Child Entity additions and changes.
- Automatic *if* model files are discoverable.

Lightweight migration is an available option when Core Data can “infer” the mapping model. These are a list of the kinds of model changes that can be inferred.

If you use an inferred model and you use the SQLite store, then Core Data can perform the migration solely by issuing SQL statements. This can represent a significant performance benefit as Core Data doesn’t have to load any of your data.

When using the Automatic option for Lightweight migration, Core Data must be able to locate Models in standard locations. Core Data looks for models in the bundles returned by NSBundle’s allBundles and allFrameworks methods.

If models are stored in other locations, you must use initialize a NSMigrationManager. This is Custom Migration.

Limits of Lightweight Migration

Some model change mappings cannot be inferred.

- Splitting or combining entities that do not share an existing hierarchy in the store model.
- Select type changes. ex. Number -> String
- Business logic fixes.
- Internal format changes. ex. minutes -> seconds

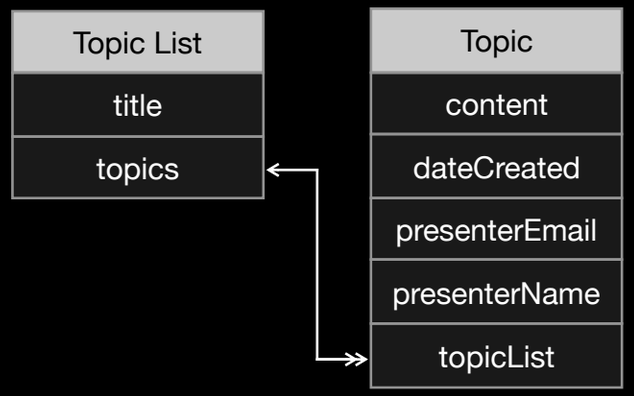
There are some changes that the Lightweight cannot handle very well.

Integer to Double and Double To Integer type changes can be inferred. Be careful that data loss is possible with narrowing range of numeric type. Ex: double -> Integer 16.

Number to string changes on attributes will actually fail when attempting to create an inferred mapping model.

Limits of Lightweight Migration

Moving, splitting, joining model elements.

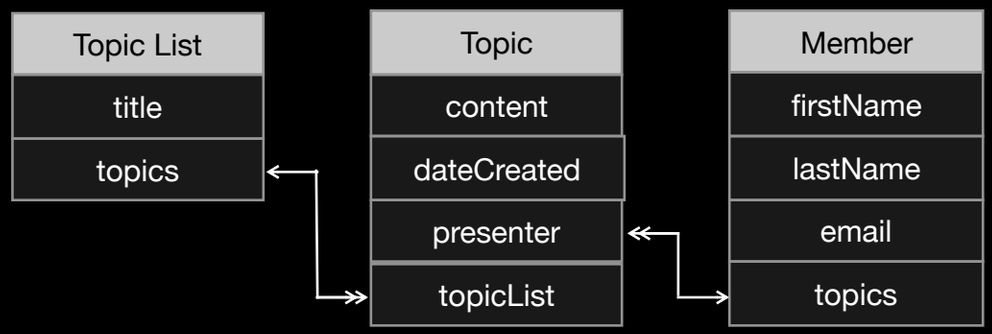


Discussing weakness of Lightweight migration, splitting or joining entities that are not part of an existing entity hierarchy.

Consider the diagram of our sample app model that we want to refactor.

Limits of Lightweight Migration

Moving, splitting, joining model elements.



Discussing weakness of Lightweight migration, splitting or joining entities that are not part of an existing entity hierarchy.

We want to demo how to refactor the Topic and Member entities from the previous model. Lightweight can handle the schema change, but it will not handle the actual data transform we want. We need custom code to do that.

Demo

Automatic Lightweight Migrations

Demo 2 Lightweight Migration
Delete app and data in simulator and run with Model1 to get baseline database populated
Show that we have 5 models in our Versioned Model file now.
Show the options in RootViewController to enable Automatic Lightweight Migration.
Set current model to Model2. Show that it contains new timeBudget attribute with default of 5min.
Run the app and then open the store. You will see a 5 min default timeBudget in each Topic.
Quit app and set current model to Model3. Renamed content to title on Topic using Renaming Identifier.
Run app and open store. Show that we've lost our Topic content. Need to uncomment code in CDMTopicsViewController.m in cellForRowAtIndexPath to use new title property to set cell content.
Run and show topic content now showing.
Quit app and set Model4 as current model. This is the minutes to seconds conversion…using Hash Modifier.
Run app and open store. Migration occurs, but data is still in minutes. This is limit of Lightweight migration.
Quit app and set model to Model5. Run app and open store.
Migration occurs and schema is changed, but we've lost the presenter information in our new database. This is another limit of Lightweight migration.
Demo ends.

Default & Custom Migration

Default vs Custom

Both offer customization using mappings and custom code.

- Default Migration
  - Automatic on store open using internal NSMigrationManager.
  - Model files must be discoverable.
  - Single pass using one mapping model.
- Custom Migration
  - Developer initiates with NSMigrationManager directly.
  - Model files can exist outside default locations.
  - Multiple passes using multiple mapping models.
  - Can use inferred mapping models like Lightweight Migration.

Default and Custom migration handle more complex change scenarios such as splitting or joining entities and attributes.

Both offer developer provided mapping models and custom code options.

Use custom migration when model files are not in standard locations, you have multiple mapping models, or you want more control of the compatibility testing and migration invocation.

Mapping Models

We’ve been mentioning mapping models. What are they?

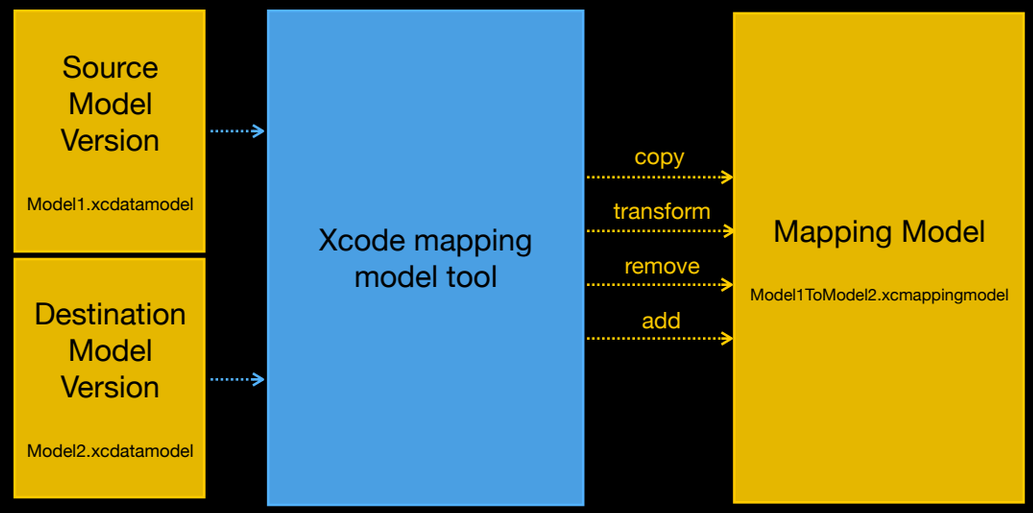
Mapping Models

A guide for selecting and transforming store data during migration.

- Encoded for a specific source -> destination migration path.
- Fetch criteria to select store data to migrate.
- Defines expressions for copying or transforming data.

Creating a Mapping Model

Xcode provides a tool for generating a mapping you can then customize.

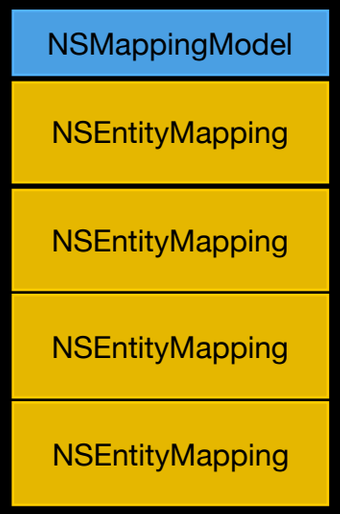


Mapping model defines what is copied, transformed, added, and removed when migrating from source->destination.

Using Xcode, a mapping model is created using File->New. It is often a project file. A resource to be added to your application bundle.

NSMappingModel

A collection of NSEntityMappings.

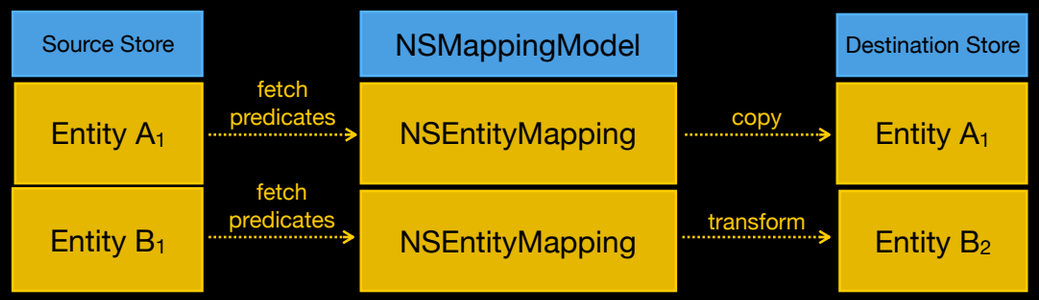


A mapping model resource is loaded using the NSMappingModel class.

A mapping model manages a collection of NSEntityMappings.

Entity Mappings

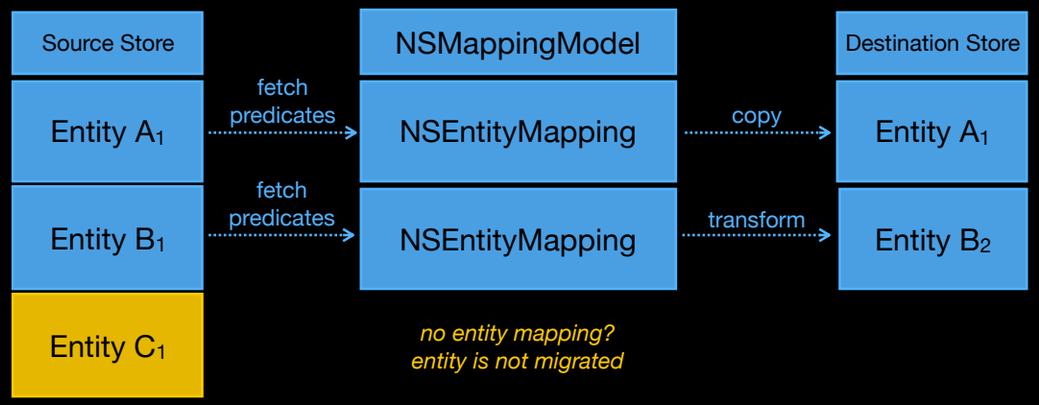
Each NSEntityMapping specifies an entity transform and optional fetch rule.



Each Entity Mapping can specify fetch criteria for selecting source instances to be mapped to the destination.

Entity Mapping

Each NSEntityMapping specifies an entity transform and fetch rule.

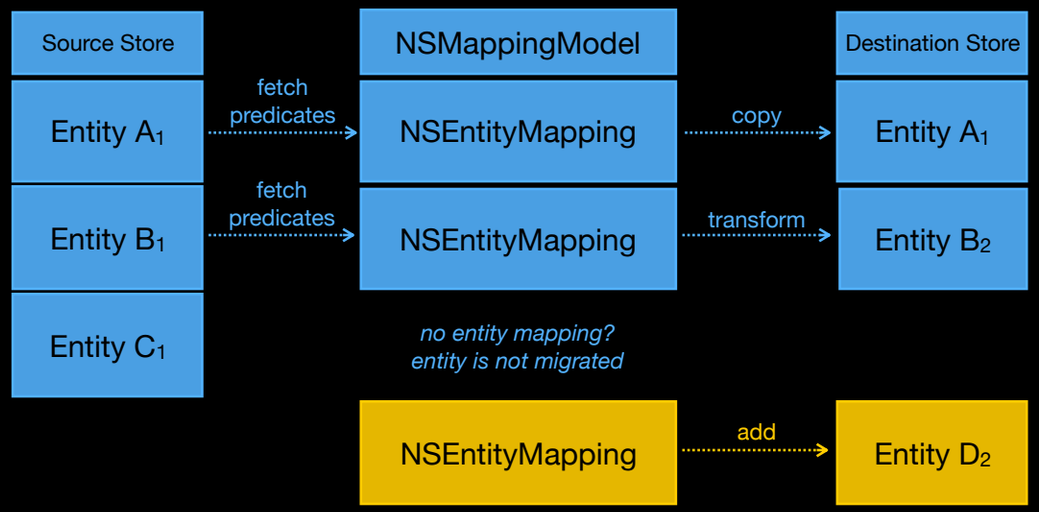


The lack of an EntityMapping for a source entity translates into the entity being dropped in the destination.

Renaming an Entity and forgetting to set the Renaming Identifier can result in this situation as well, which may be an error.

Entity Mapping

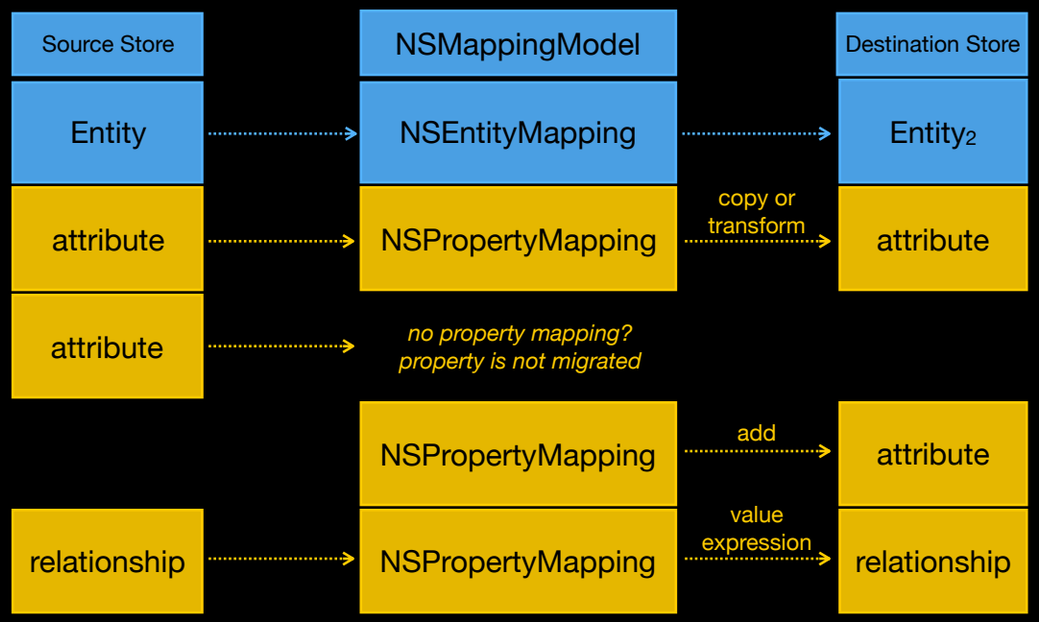
Each NSEntityMapping specifies an entity transform and fetch rule.



For new entities, you may still want to have a entity mapping to setup values and relationships rather than rely on model defaults.

Property Mapping

NSEntityMapping is composed of a collection of NSPropertyMappings.



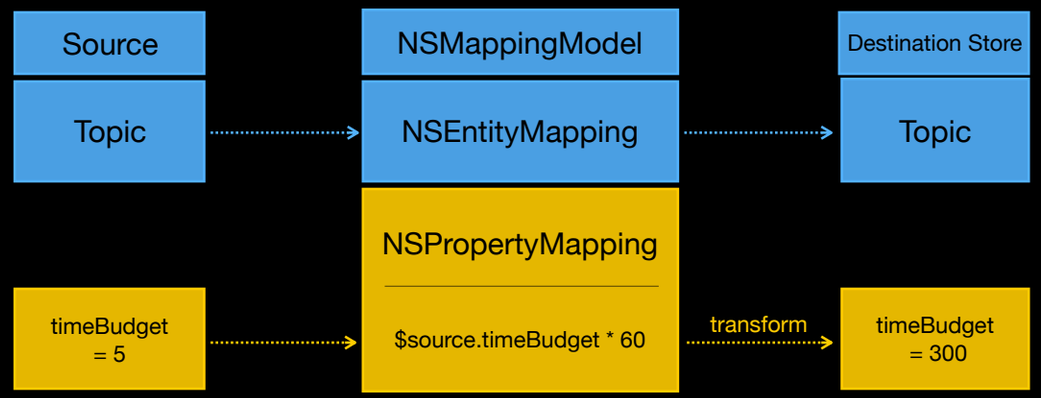
NSEntityMapping manages a collection of NSPropertyMappings. One for each persistent attribute and relationship of the destination model.

Property mappings often define a value expression to copy an attribute or relationship as is from the source entity.

If a transform is needed, this expression can be customized.

Transform Expression

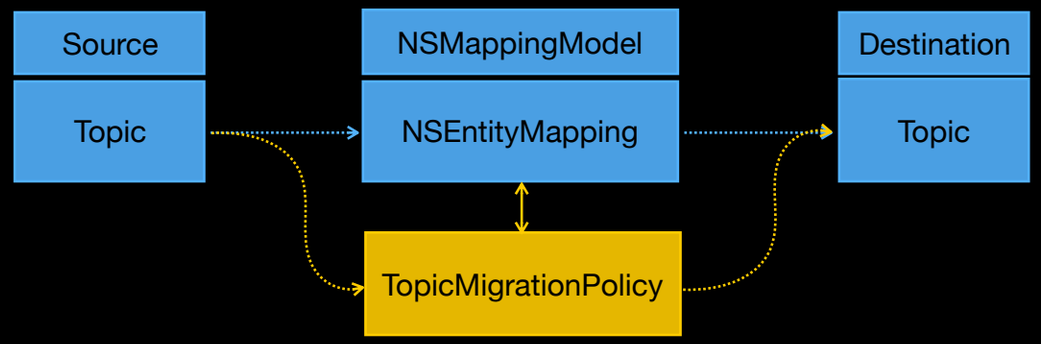
Property mapping value expression example.



Here is an example of a value expression the developer customized to handle a business logic model change.
timeBudget is transformed from minutes in the source to seconds in the destination.

Entity Migration Policy

NSEntityMigrationPolicy is subclassed to enhance mapping with your code.



The final feature to note about Mapping Models is the ability to customize with developer code.

This typically done by subclassing the NSEntityMigrationPolicy and specifying your custom class name in the Entity Mapping. This is done using Xcode's mapping model editor.

Custom migration policies provide the mechanism to hook into the underlying stages of the migration process.

We will discuss the migration process next, then show an example of using custom policies.

Migration Process

Initialization

A NSMigrationManager drives the process. Init with Store and App models.

NSMigrationManager

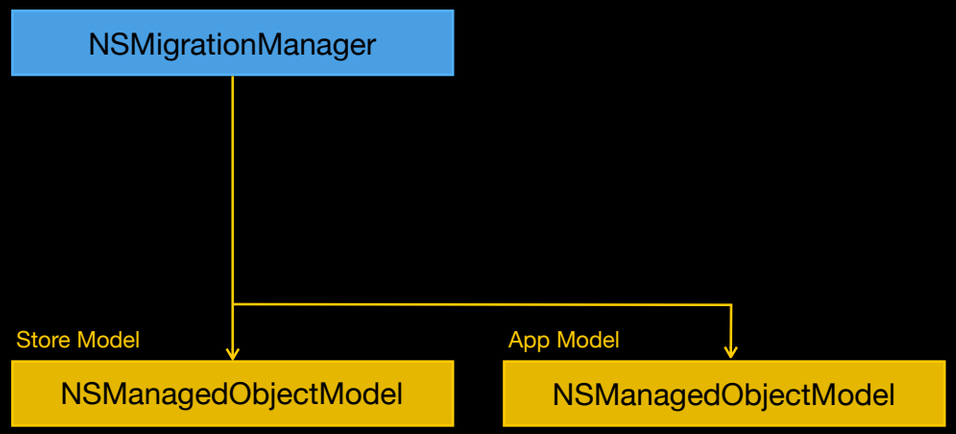
All migrations are managed by NSMigrationManager.

Automatic migrations create one for you.

Custom migration is handled by the developer allocating and initializing a NSMigrationManager instance.

Initialization

A NSMigrationManager drives the process. Init with Store and App models.



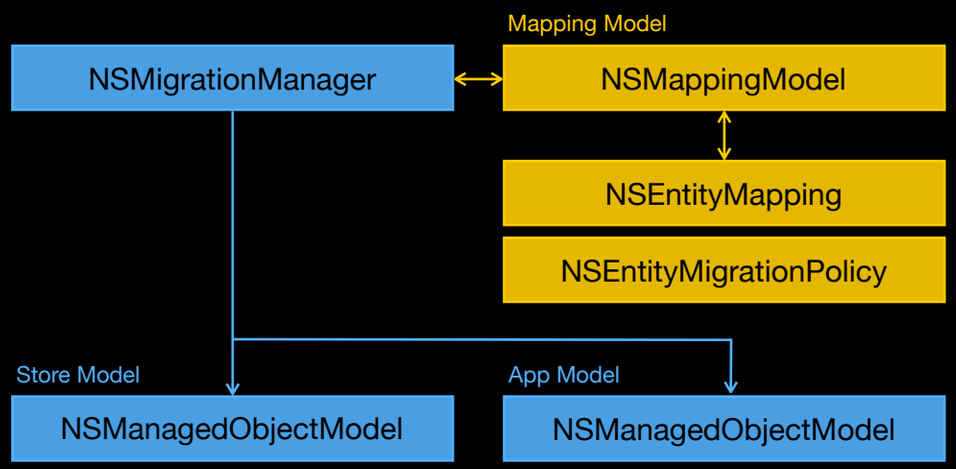

```
// NSMigrationManager
- (id)initWithSourceModel:(NSManagedObjectModel *)sourceModel
         destinationModel:(NSManagedObjectModel *)destinationModel;
```

If you’re not using an Automatic option, you initialize the migration manager yourself with the store and app models. For Automatic migrations, Core Data will do this for you.

The store model is your source, the current model in your app is the destination.

Started with a Mapping Model

And the source & destination store locations and options you want to use.



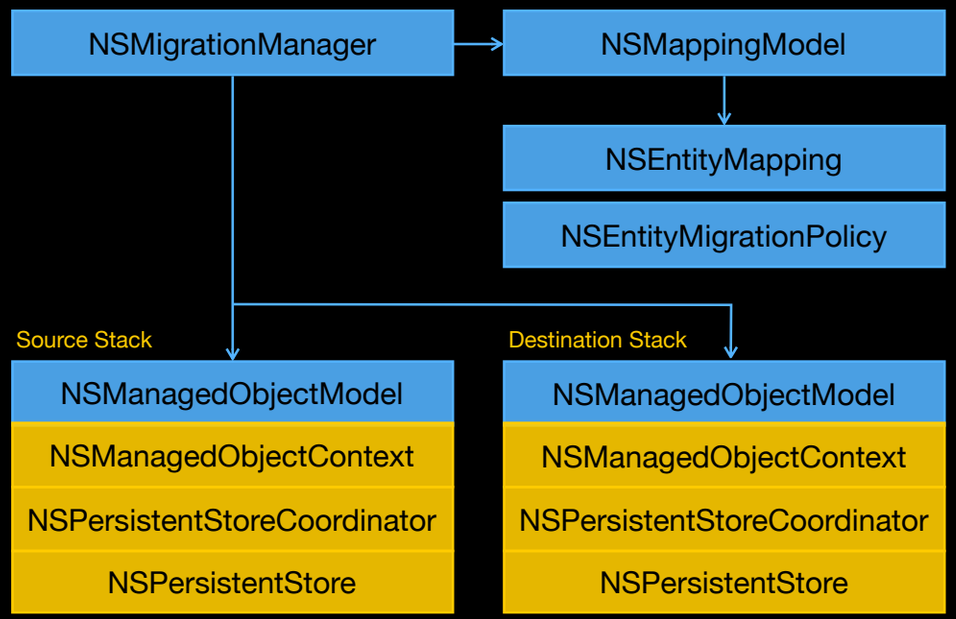

```
// NSMigrationManager
- (BOOL)migrateStoreFromURL:(NSURL *)                sourceURL
                       type:(NSString *)             sStoreType
                    options:(NSDictionary *)         sOptions
           withMappingModel:(NSMappingModel *)       mappings
           toDestinationURL:(NSURL *)                dURL
            destinationType:(NSString *)             dStoreType
         destinationOptions:(NSDictionary *)         dOptions
                      error:(NSError **)             error;
```

Migration is started by invoking migrateStoreFromURL and specifying the store locations and a mapping model.

For Automatic migrations, Core Data starts the migration if needed when adding the store to the store coordinator.

Stacks Initialization

Two Core Data Stacks. One for the source and one for the destination.



The migration manager creates two Core Data stacks. One for each store.

During the migration, source and destination contexts are accessible to your code for fetching, inserting, deleting as needed.

For lightweight migration of SQLite stores, Core Data does not utilize the two stacks, instead it migrates the store by issuing SQL statements, effectively migrating the store in place within SQLite.

Stage 1 - Entity Creation

Destination entities are added, copied, or transformed from source store.


```
// NSEntityMigrationPolicy
- (BOOL) createDestinationInstancesForSourceInstance:(NSManagedObject *)sInstance
                                        entityMapping:(NSEntityMapping *)mapping
                                              manager:(NSMigrationManager *)manager
                                                error:(NSError **)error;
```

During the migration, Core Data does not instantiate your custom NSManagedObject classes you may specified in your Models. Migration uses only standard NSManagedObject instances.

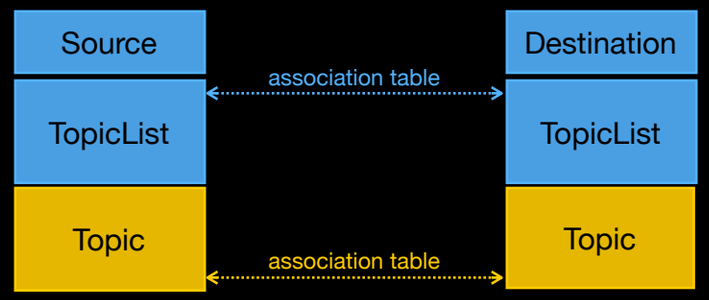
In this stage, Core Data populates only the attributes of destination objects.

As entities are created in the Destination store, the migration manager keeps a mapping of source entities instances and the destination entity instance that was created. These “association tables” are used in stage 2 to help in recreating relationships.

If you provided a custom migration policy class, you can participate in this stage by implementing the method shown here. If you handle destination instance creation on your own, you’ll need to tell the migration manager about this using the associateSourceInstance method.

Stage 1 - Entity Creation

Destination entities are added, copied, or transformed from source store.



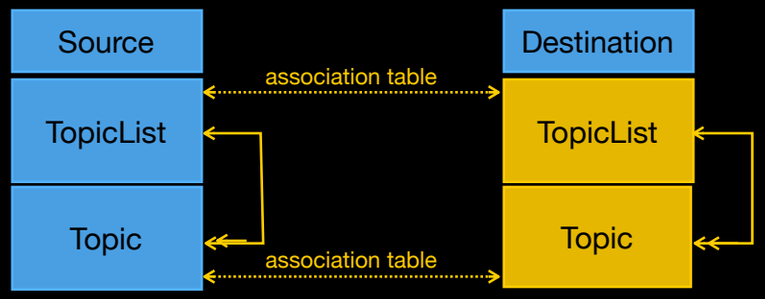

```
// NSEntityMigrationPolicy
- (BOOL) createDestinationInstancesForSourceInstance:(NSManagedObject *)sInstance
                                        entityMapping:(NSEntityMapping *)mapping
                                              manager:(NSMigrationManager *)manager
                                                error:(NSError **)error;
```

The stages are completed in a breadth first sequence. Each custom policy is tasked to create the destination entity.

The order of the sequence can be set by the developer in Xcode by dragging the Entity mapping entries in the order you need.

Stage 2 - Recreate Relationships

Source relationships and association tables guide destination rebuild.



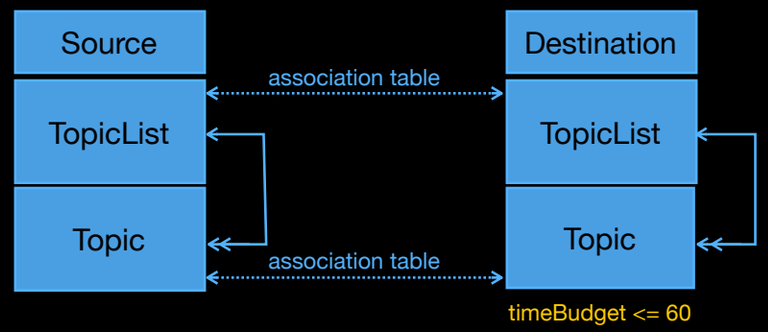

```
// NSEntityMigrationPolicy
- (BOOL) createRelationshipsForDestinationInstance:(NSManagedObject *)dInstance
                                      entityMapping:(NSEntityMapping *)mapping
                                            manager:(NSMigrationManager *)manager
                                              error:(NSError **)error;
```

The migration manager utilizes the association tables to match source objects to the new destination objects and recreate relationships in the Destination store.

If you provided a custom migration policy class, you can participate in this stage by implementing the method shown here.

Stage 3 - Validation & Save

Custom policy validation methods + validation rules of destination entities.




```
// NSEntityMigrationPolicy
- (BOOL) performCustomValidationForEntityMapping:(NSEntityMapping *)mapping
                                         manager:(NSMigrationManager *)manager
                                           error:(NSError **)error;
```

Validation includes invoking any validation your custom policy classes have implemented. Finally, the validation rules of the Destination Model are evaluated.

Additional Migration Policy APIs

Handy for setup and cleanup activities you may need to perform.

```
// NSEntityMigrationPolicy

- (BOOL) beginEntityMapping:(NSEntityMapping *)mapping
                    manager:(NSMigrationManager *)manager
                      error:(NSError **)error;

- (BOOL) endEntityMapping:(NSEntityMapping *)mapping
                  manager:(NSMigrationManager *)manager
                    error:(NSError **)error;


- (BOOL) endInstanceCreationForEntityMapping:(NSEntityMapping *)mapping
                     manager:(NSMigrationManager *)manager
                       error:(NSError **)error;

- (BOOL) endRelationshipCreationForEntityMapping:(NSEntityMapping *)mapping
                     manager:(NSMigrationManager *)manager
                       error:(NSError **)error;
```

Additional APIs you can override in your custom migration policy class.

NSMigrationManager APIs

Helpful APIs for handling custom migration of relationships.

```
// NSMigrationManager

- (void) associateSourceInstance:(NSManagedObject *)sourceInstance
            withDestinationInstance:(NSManagedObject *)destinationInstance
            forEntityMapping:(NSEntityMapping *)entityMapping;

- (NSArray *) destinationInstancesForEntityMappingNamed:(NSString *)mappingName
            sourceInstances:(NSArray*)sourceInstances;

- (NSArray *) sourceInstancesForEntityMappingNamed:(NSString *)mappingName
            destinationInstances:(NSArray *)destinationInstances;
```

Helpful APIs for handling custom relationship migrations.

Demo

Default Migrations using Mapping Models and Custom Policies

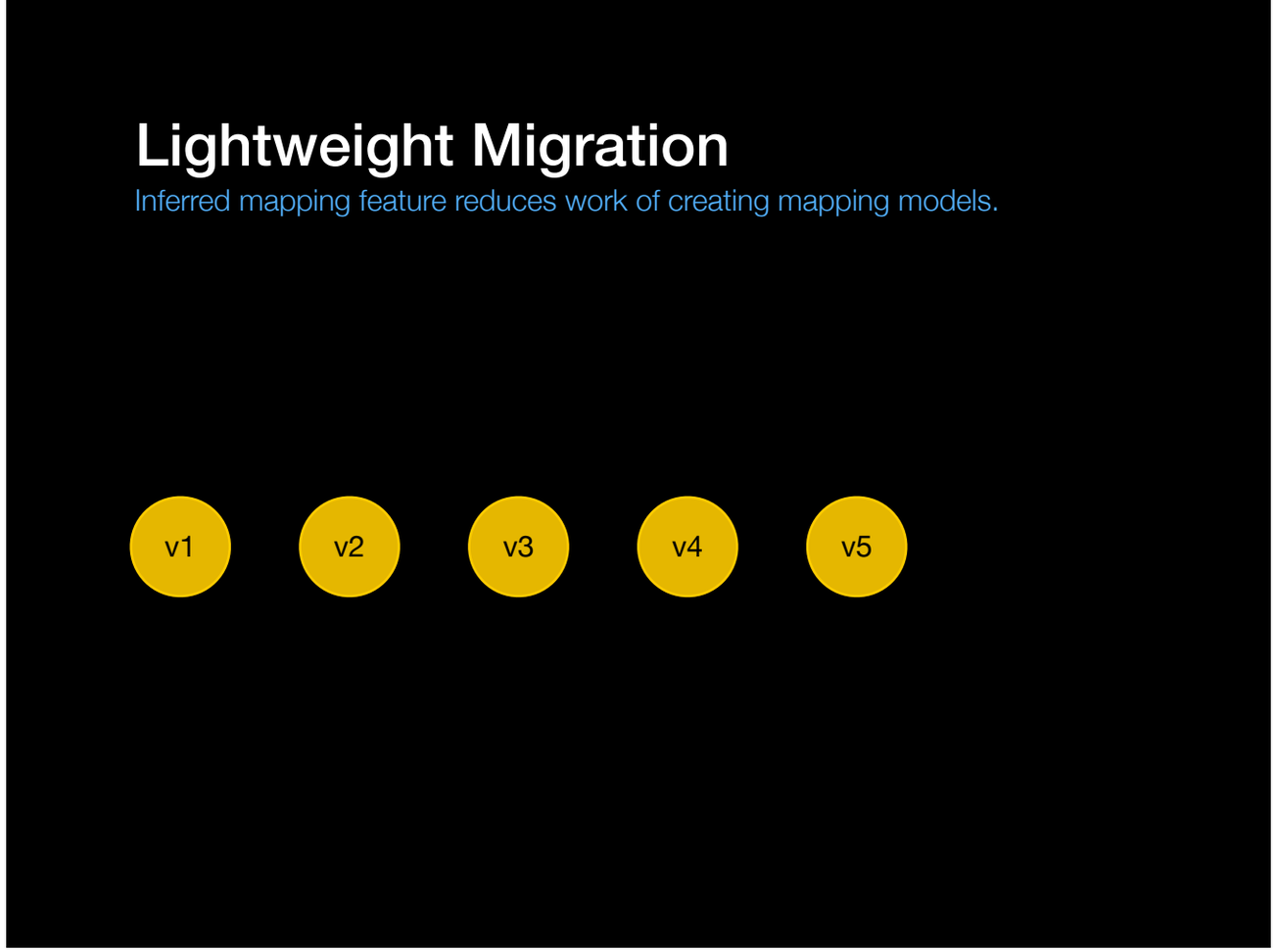
Demo 4 Default Migration
The demo is run the same as Demo 2.
Differences include we now have mapping models and a custom TopicToTopic migration policy.

In this demo, the use of these mappings and policy code accomplish the following:
Model1->Model2. We set the timeBudget to be 20% of the topic content length…vs defaulting to 5 min. This is done using a value expression in the mapping model.
Model3->Model4. The conversion of timeBudget to seconds from minutes works using a value expression in the mapping model.
Model4->Model5. Presenter data is successfully split from the Topic and a Member entity populated as needed. Relationships between Topic and Member are created correctly.

Progressive Migrations

Progressive migration is a solution approach that developers in the Core Data community have written and blogged about.

These slides review the strengths of Lightweight, Default, and Custom Migration…and their weaknesses in an effort to build an understanding of why progressive migration is a useful approach for real world problems.



Lightweight migration is great to use when it can infer mappings.

If your customers or development team have older stores of varying versions, lightweight migration can handle any of the migration paths that might exist out in the world.

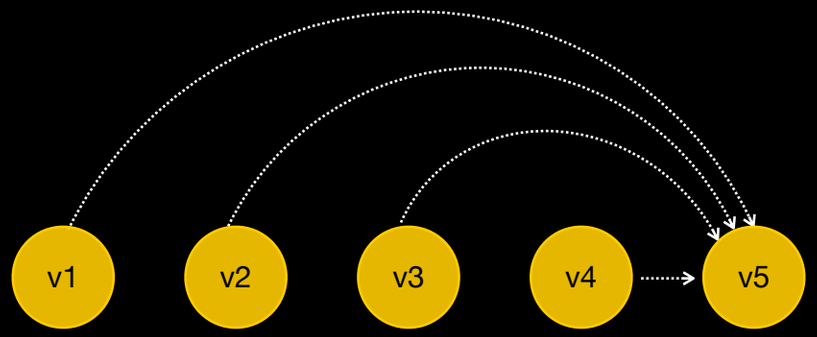
Lightweight Migration

Migrating from each previous release to the next is straight forward.

Lightweight handles the expected path of migrating from the last release to the next release. All without the need for creating mapping models or custom code.

Lightweight Migration

We can migrate across versions in a single hop without custom mappings.

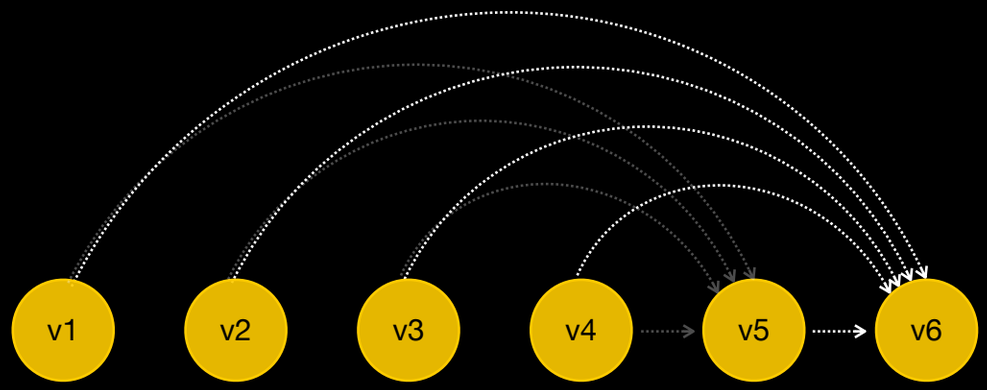


The inferred mapping helps with all the permutations of stores stuck in the past.

That’s 4 additional mapping models you didn’t have to create.

Lightweight Migration

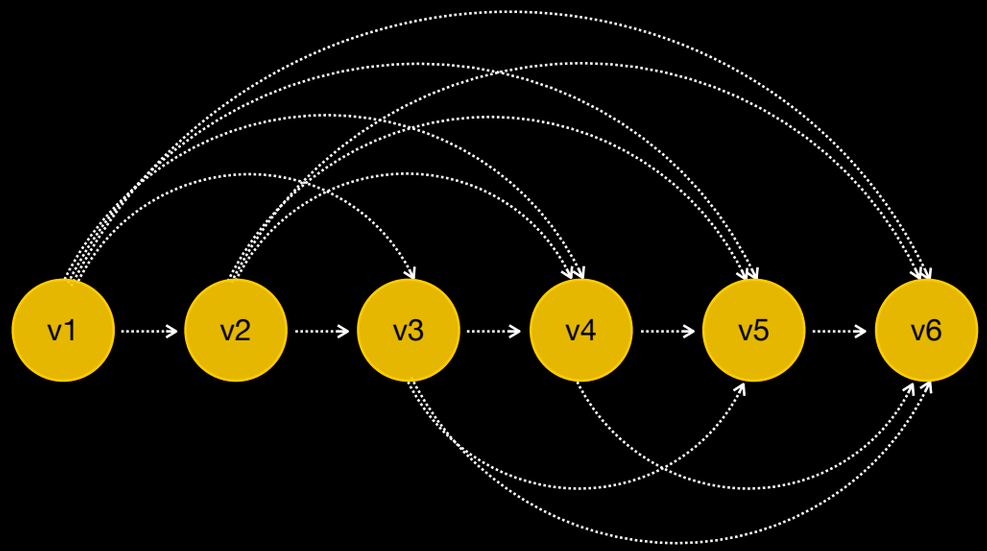
All that hopping could become a challenge with regression testing.



Some concerns:
Testing all the migration hop permutations could be prohibitive. Each new model release requires regression testing migrations from the same older store models to the latest app model.

Lightweight Migration

Over time, that adds up to quite a bit of regression testing.

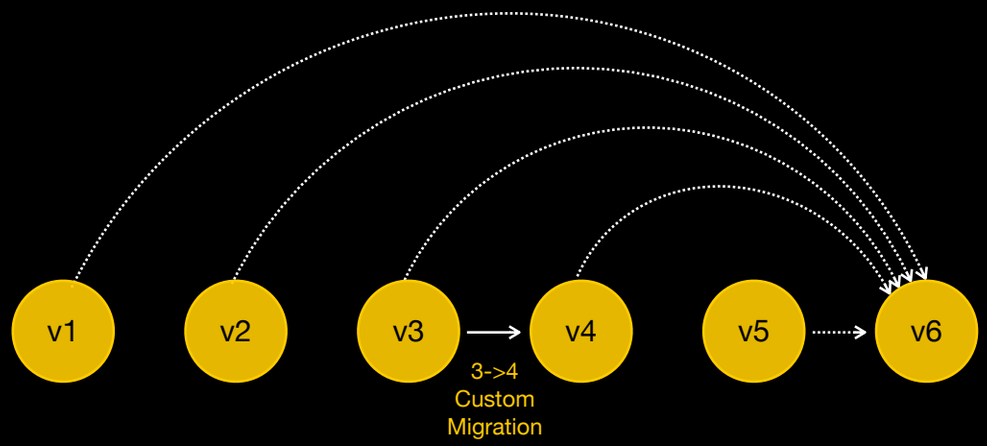


Over time, you've tested migration from v1 to a new version 5 times.

All this regression testing could be a concern for you.

Lightweight Migration

A required custom migration can create a problem for this design pattern.



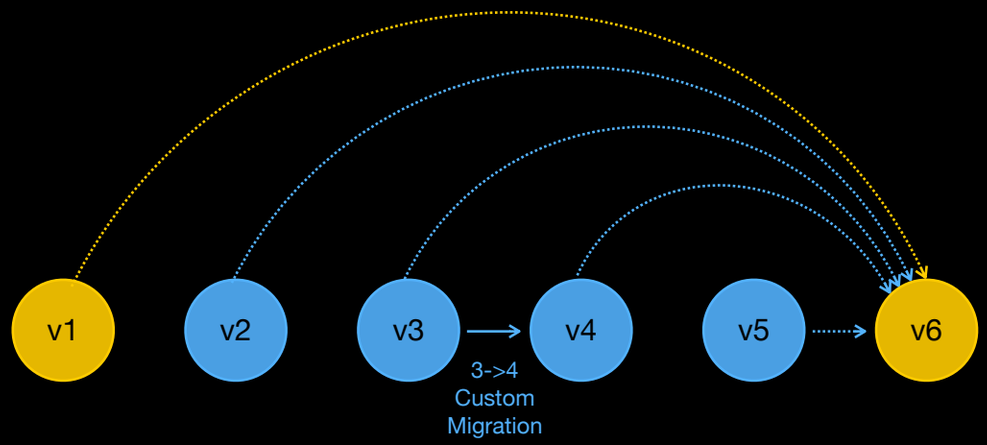
Another concern:
If a custom mapping is needed one day, a lightweight migration only solution becomes a problem. If lightweight is your only approach, this scenario of overstepping a needed custom migration could be a problem for you.

Your custom mapping and code is only used when the store is at version 3 and the app is version 4.

The custom migration may fix a bug in the data or represent a model change lightweight can’t handle very well or at all.

Lightweight Migration

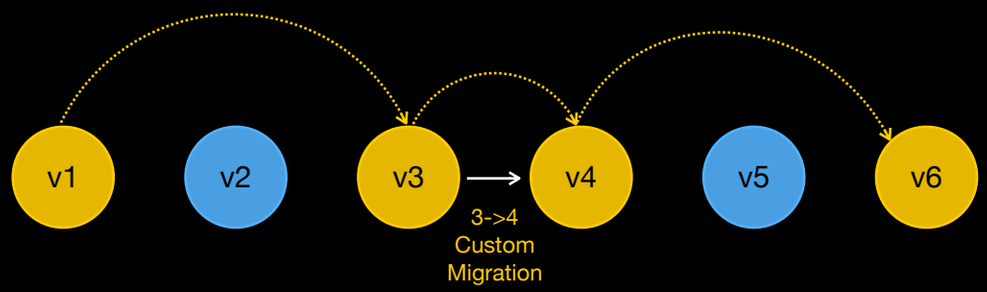
A required custom migration can create a problem for this design pattern.



Problem illustration. Store is at v1. App is at v6. Lightweight migration will attempt inferred mapping and migration from v1 to v6, skipping some potentially important data fix implemented in your v3->v4 code.

Lightweight Migration

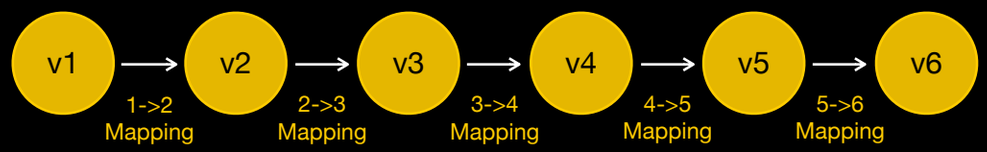
A required custom migration can create a problem for this design pattern.



What we'd like is for the 3->4 migration to be a way point that is visited in the migration from v1 to v6.

Default & Custom Migration

Each hop is well defined with a mapping model.



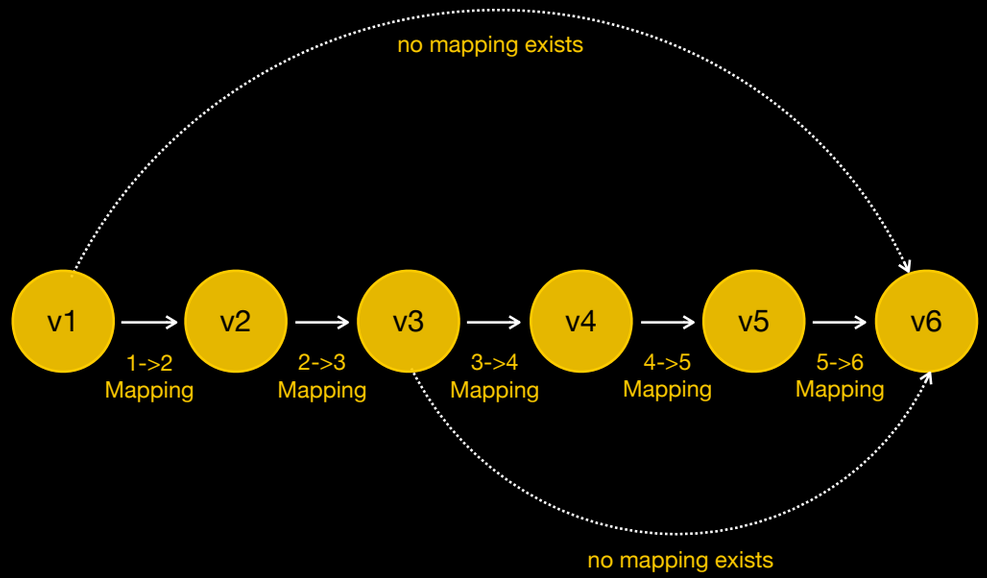
Now let’s look at Default and Custom Migration.

For default and custom migration, you would typically build a mapping from the one model version to the next model version.

One advantage to this setup is there are only 5 migration paths to test in this scenario. Once we test 1->2, we don’t have to retest when we ship newer versions.

Default & Custom Migration

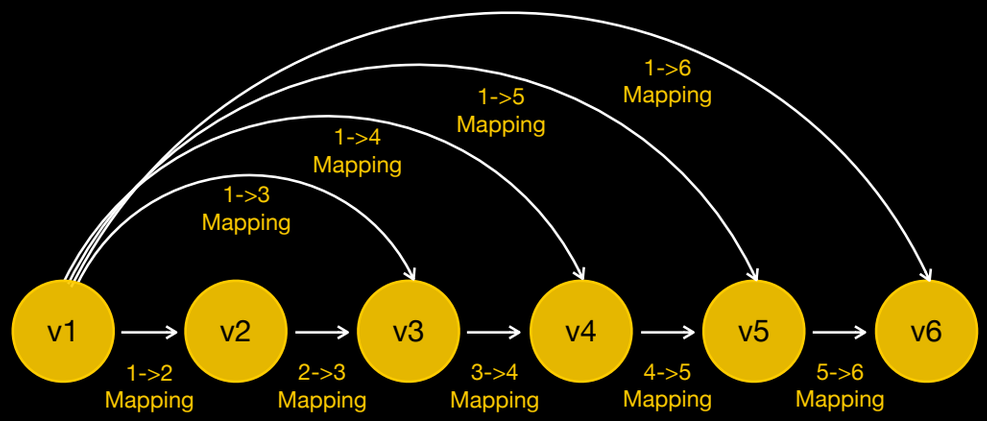
Multiple version hops don't work unless we have a mapping model.



The disadvantage: We cannot hop from v1 or v3 to v6 directly without a 1->6 or 3->6 mapping model.

Default & Custom Migration

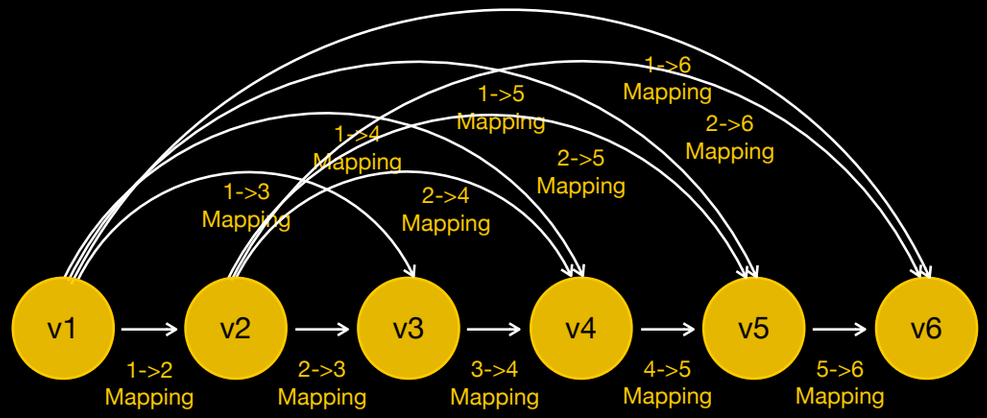
To overcome this, add more mappings.



We can overcome this by authoring more mapping models and bundling them in our app.

Default & Custom Migration

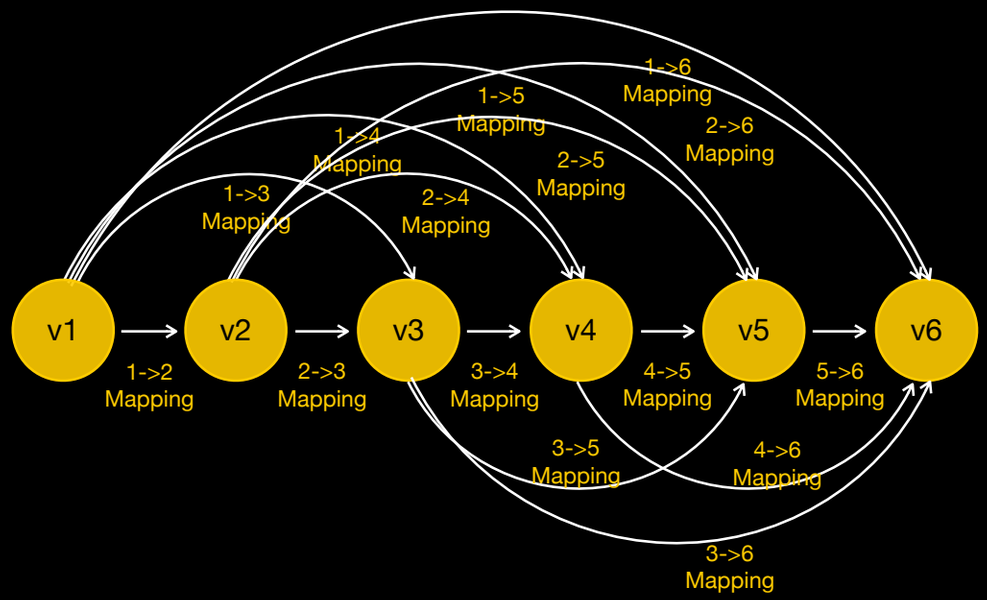
Adding more mappings becomes a burden.



This can get grow the number of mappings we have to create, test, maintain.

Default & Custom Migration

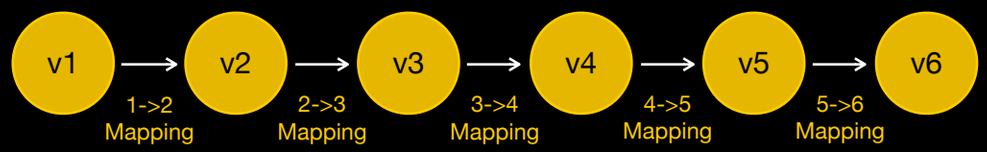
This can become a mess very quickly.



Eventually, it is not sustainable.

Progressive Migration

Iterative steps from source to final target using Custom Migration.

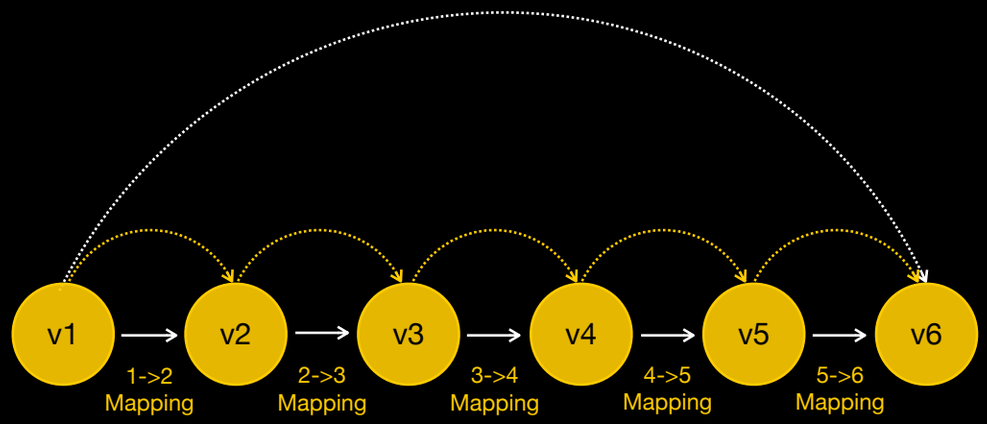


To avoid the proliferation of mapping models, we consider progressive migration.

Progressive migration is simply a recursive application of custom migrations we've built with our mapping models to move the user from source version toward the final destination version.

Progressive Migration

Iterative steps from source to final target using Custom Migration.

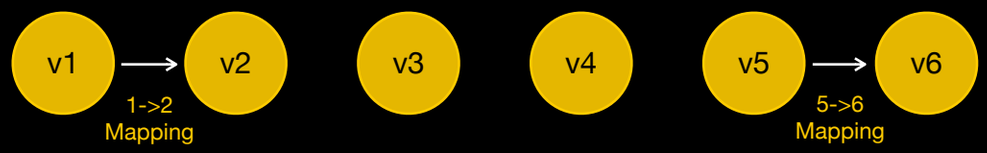


Here’s an example migrating progressively from v3 to v6. We progressively migrate the store 5 times from v1->2->3->4->5->6.

This works nicely. But can we take advantage of Lightweight Migration’s flexibility, inferred mapping, and in situ migration of SQLite stores to extend this design pattern and avoid creating mapping models we don’t necessarily need?

Hybrid Migration

Custom Migration as needed. Lightweight Migration for everything else.

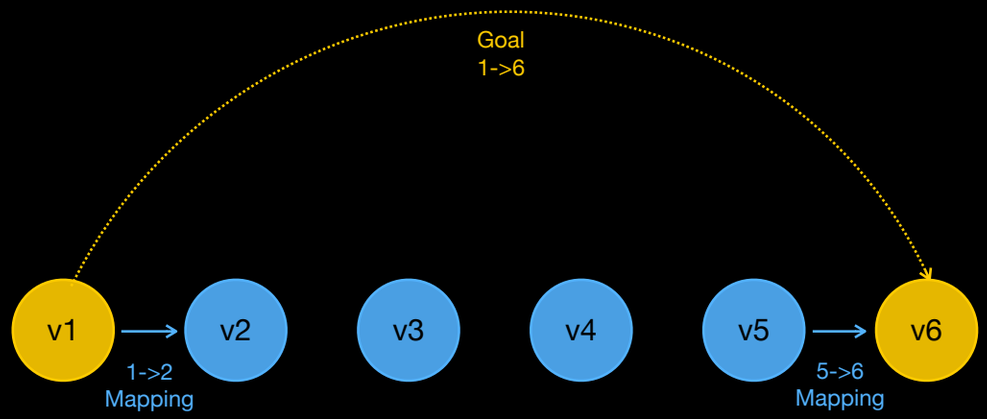


A Hybrid Migration solution allows the developer to supply mappings for migrations that need developer intervention (mapping and policy code). For all other migrations, lightweight will do the work.

Consider this scenario. The developer has mappings for 1->2 and 5->6. These mappings may be due to fixes needed with code or limits of inferred mapping.

Hybrid Migration

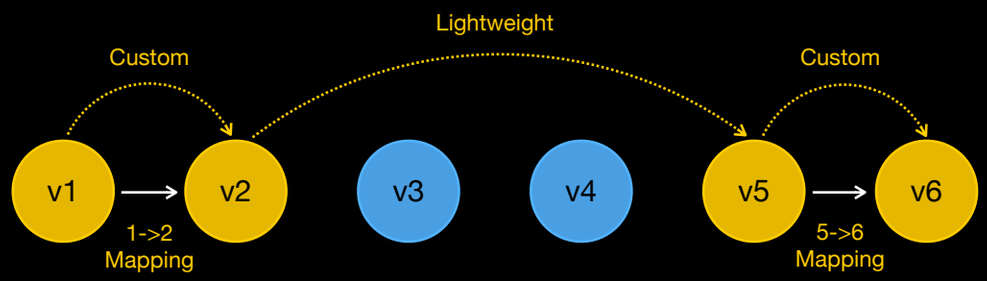
Custom Migration as needed. Lightweight Migration for everything else.



We have a v1 store we need to migrate to the latest v6 app.

Hybrid Migration

Custom Migration as needed. Lightweight Migration for everything else.



The Hybrid Migration approach uses a Custom Migration for developer supplied mappings 1->2 and 5->6, but takes a single leap from v2->v5 using Lightweight migration.

The advantages:
Only required mapping models migrations need to be built by the developer.
Lightweight is used to provide inferred mappings where developer mappings don't exist, taking multiple-version hop as far as it can. We also gain speed and memory efficiency advantage if using SQLite stores.

With this approach, we may have fewer migration paths to test. More than progressive, but possibly less than a complete lightweight solution.

Demo
Hybrid Migration

Demo 5 Hybrid Migration
This demo uses the same configuration of Demo 4 but replaces the code in CDMRootViewController in openPersistentStore to instantiate a CDMMigrationManager class which handles the hybrid migration.

Script
Delete app and data from the simulator.
Set current model to Model1. Run app to generate baseline database.
Quit app.
Set current model to Model5.
Run app and open the store.
The console log will show evidence of 4 migrations. 3 custom and one lightweight.
Review the Topics tableView. We show valid topics with timeBudgets in seconds, set by the length of the topic itself, and properly factored Topic/Member entities.
Demo ends.

Hybrid Migration Setup

Based on Convention vs Configuration

- Model file naming convention:
  `<modelName><version>.xcdatamodel`

  ```
  Model2.xcdatamodel
  Model3.xcdatamodel
  ```

- Mapping model file naming convention:
  `<modelName><version>To<modelName><version>`

  ```
  Model2ToModel3.xcmappingmodel
  ```

- Multi-part mapping model naming convention:
  `<modelName><version>To<modelName><version>-<order>`

  ```
  Model4ToModel5-1.xcmappingmodel
  Model4ToModel5-2.xcmappingmodel
  Model4ToModel5-3.xcmappingmodel
  ```

Informing the hybrid migration of the migration path uses a convention based scheme. By naming model files and mapping model files using the convention above, the manager can build the plan for migration from the store to the app.

An improvement to the naming convention would be support for the common pattern of naming development versions of models and mappings using the app bundle version followed by a build number. For example, 1.2.1 {1}, 1.2.1 {2}, 1.2.1 {2}, 1.2.1 {4} or similar.

Wrap Up

Migration Insights

- Automatic Lightweight infers a mapping model unless a developer mapping model is found.
- Lightweight migration can fail due to model validation rules.
- Using inferred mapping will migrate SQLite stores in-place.
- Your entity subclasses are not instantiated during migration.
- Hash Modifiers expose invisible model changes.

Some recaps and insights I learned during the research into Core Data Migration.

Migration Insights

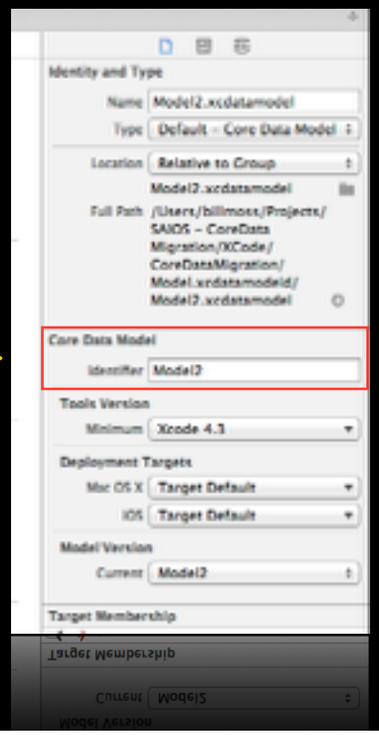
- Migration Policy classes are alloc’d and dealloc’d for each stage of the migration.
- Migration concatenates data if the destination store exits.
- Address memory pressure using multi-pass migration.
- Multi-pass migration can be done with fetch predicates or entity mapping distribution.

Some recaps and insights I learned during the research into Core Data Migration.

Set the Core Data Model Identifier

Identifying the <modelName> and <version> is helpful during migrations.

- This is added to the Store Meta Data. Helpful for debugging and audit trail.
- You can inspect a NSManagedObjectModel and find this in the versionIdentifiers property.



The Core Data model Identifier can be used as a simple mechanism for switching logic in your custom migration policy code. Other developers have used userInfo dictionaries for this.

This Identifier is added to the Store Meta Data which could be helpful for audit trails and debugging.

UserInfo Dictionaries

Useful for passing meta data to Custom Policy classes during migration.

Accessible from your custom policy code ⟶

There are many User Info dictionaries in the model file and mapping model file that can be used to supply meta data for migration code use.

Use Core Data debug flags

Launch arguments for helpful console logging of Core Data activities.

- -com.apple.CoreData.MigrationDebug 1
- -com.apple.CoreData.SQLDebug 1

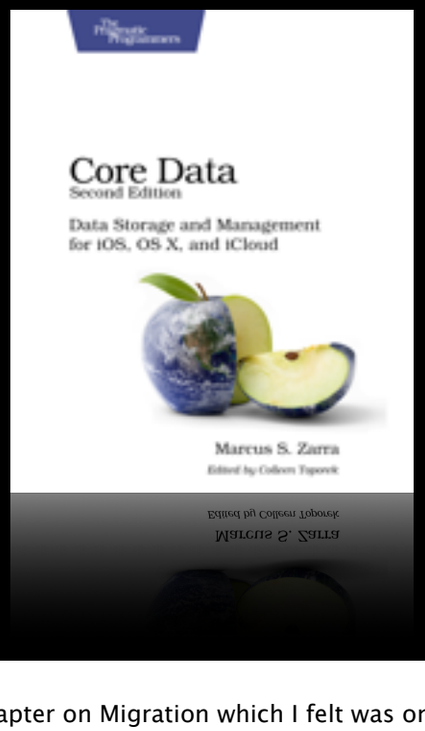
Tool Tips & Challenges

- Removing unwanted Models in a Versioned Model file is not available through Xcode. Edit the package with Finder.
- Ordering Model files in a Versioned Model in Xcode doesn’t work.
- Update Default Mappings caused my mapping file to become unusable.
- Regularly use Clean, Clean Build Folder, and *delete the app* to work around model file edits that don’t always get deployed in a build.

Some tips and problems I had to work around when experimenting with model versions and mapping models. Core Data model editor works great. Main issues seem to be with dealing with Mapping Model and Model updates. Most of these issues are minor and can be overcome with some hand editing.

Resources

Books



Marcus Zarra's book is a recent edition. It has a really good chapter on Migration which I felt was one of the most comprehensive of sources I've found on the topic.

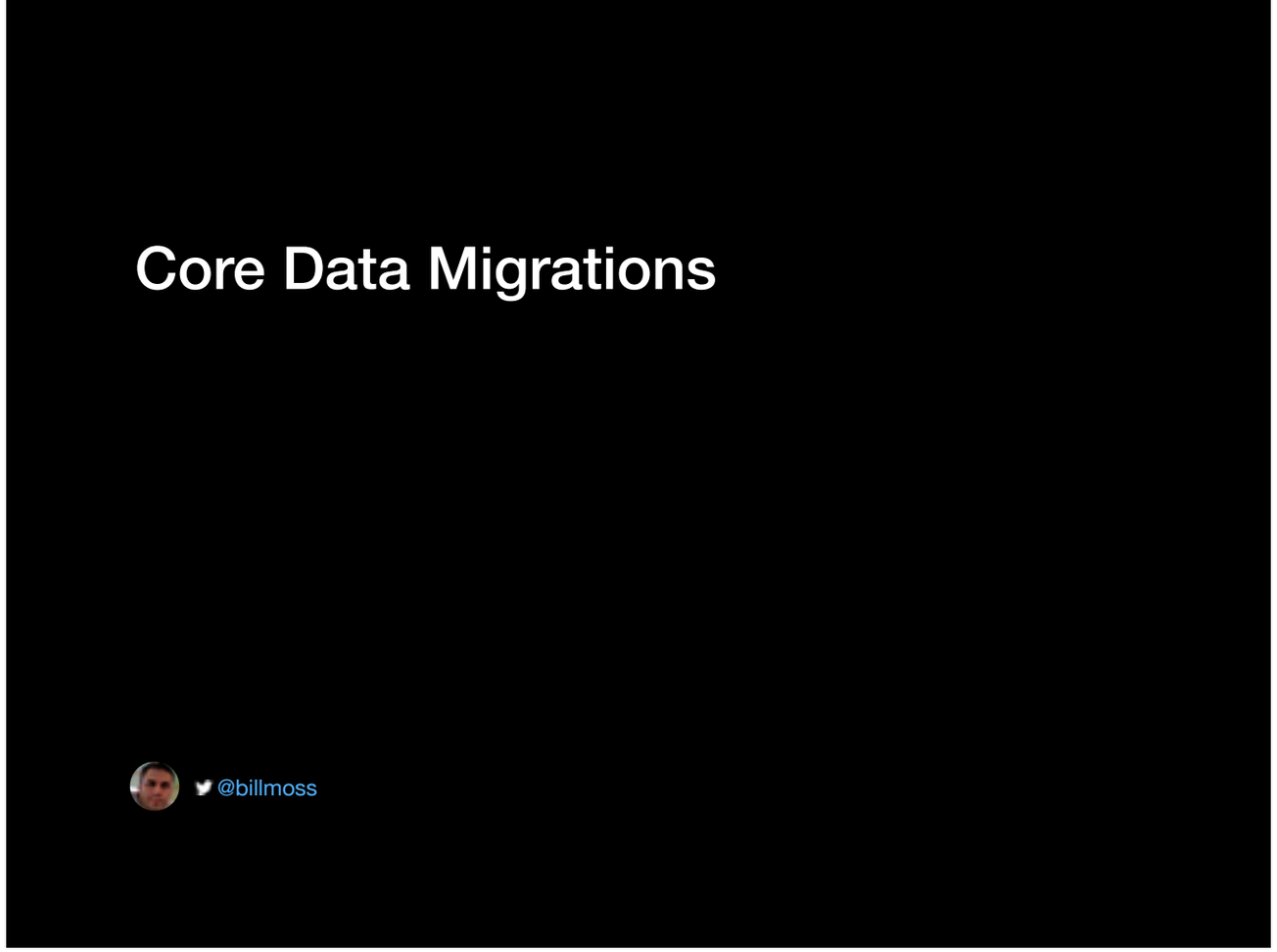
Both books are good for learning Core Data. Both have good chapters on Migration.

There is some overlap between the two, but each offer some unique content.

Links

- Apple Developer Docs on Core Data
  http://developer.apple.com/library/ios/documentation/cocoa/conceptual/coredata/articles/cdTechnologyOverview.html
- Apple Developer Docs on Core Data Migration
  http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreDataVersioning
- objc.io #4 - Core Data & Migration
  http://www.objc.io/issue-4/core-data-migration.html
- Art & Logic - Iterative Migration
  http://www.artandlogic.com/blog/2013/07/iterative-core-data-migrations/

Some links to web resources I thought were good for learning migration and Core Data.

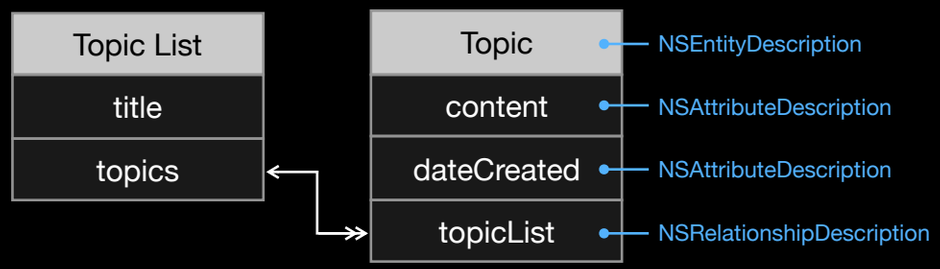


Thank you.

These are some slides I cut out of the presentation. They were more detailed slides on the Entity Version Hash scheme Core Data uses for Model Compatibility testing.

Entity Version

Core Data uses version hashes to uniquely identify an entity and its' attributes, and relationships.



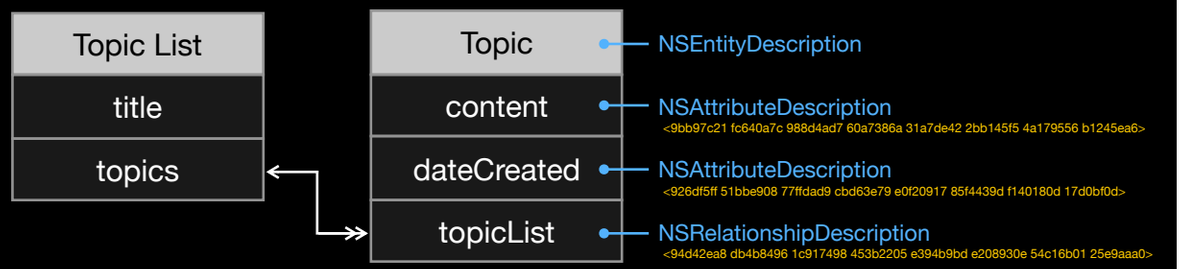
Core Data assigns to each Entity a version hash to uniquely identify its' description.

From the NSEntityDescription header file:
The version hash is used to uniquely identify an entity based on the collection and configuration of properties for the entity.

The version hash uses only values which affect the persistence of data and the user-defined versionHashModifier value. (The values which affect persistence are the name of the entity, the version hash of the superentity (if present), if the entity is abstract, and all of the version hashes for the properties.) This value is stored as part of the version information in the metadata for stores which use this entity, as well as a definition of an entity involved in an NSEntityMapping.

Entity Version

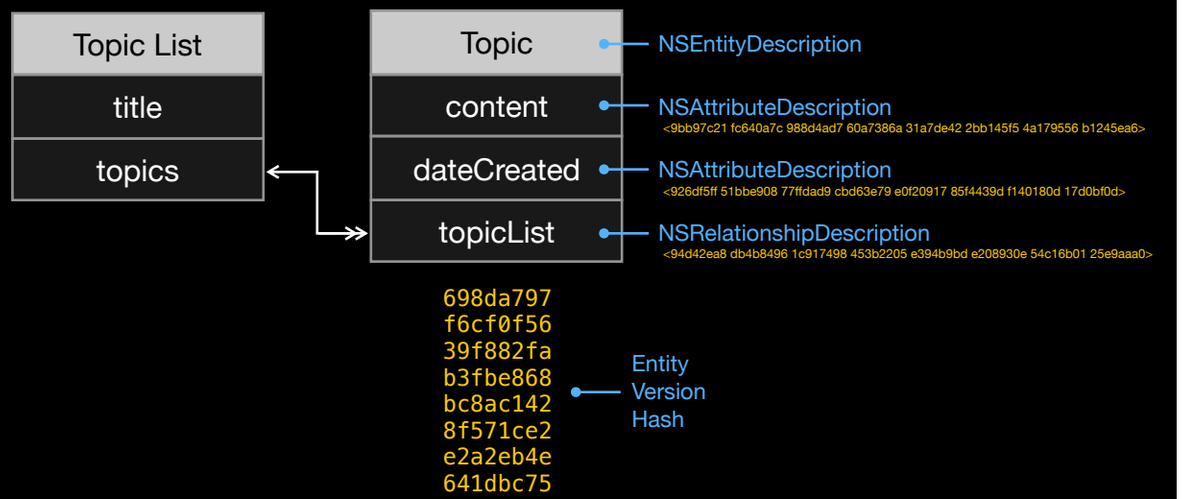
Core Data uses version hashes to uniquely identify an entity and its' attributes, and relationships.



Attributes and relationships are assigned a version hash of their own.

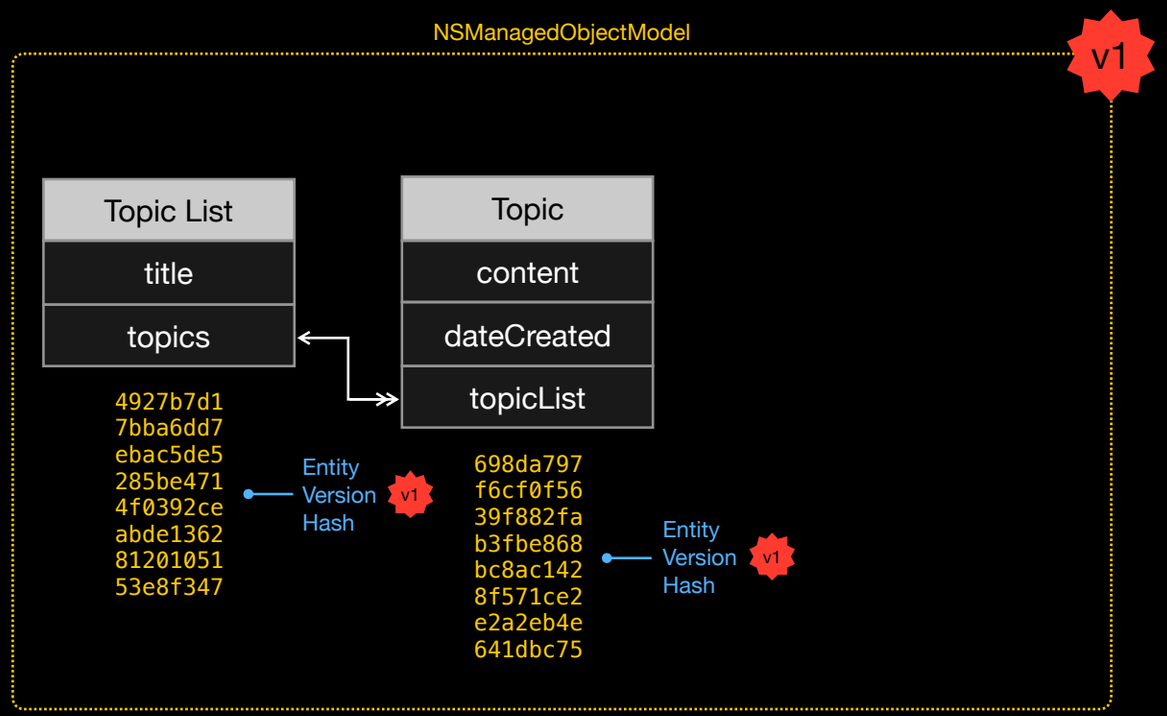
Entity Version

Core Data uses version hashes to uniquely identify an entity and its' attributes, and relationships.



The entity hash is built using its' own properties along with the version hashes of its' attributes and relationships.

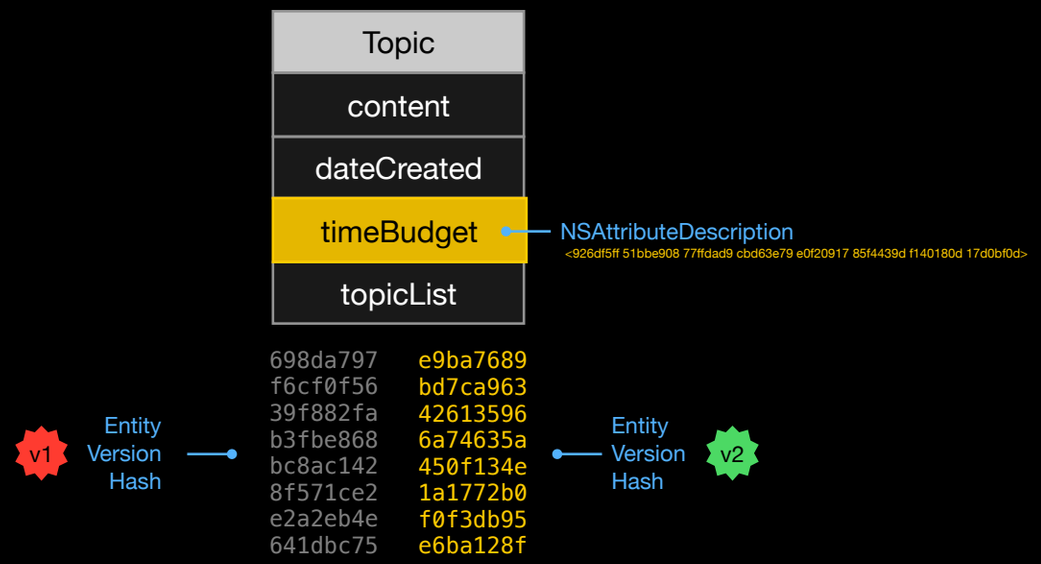
Model Version



Now, let's go back to our Version 1 Model. These two entities establish the model version with their respective entity version hashes. These hashes are stored within the store meta data.

Entity Version Changes

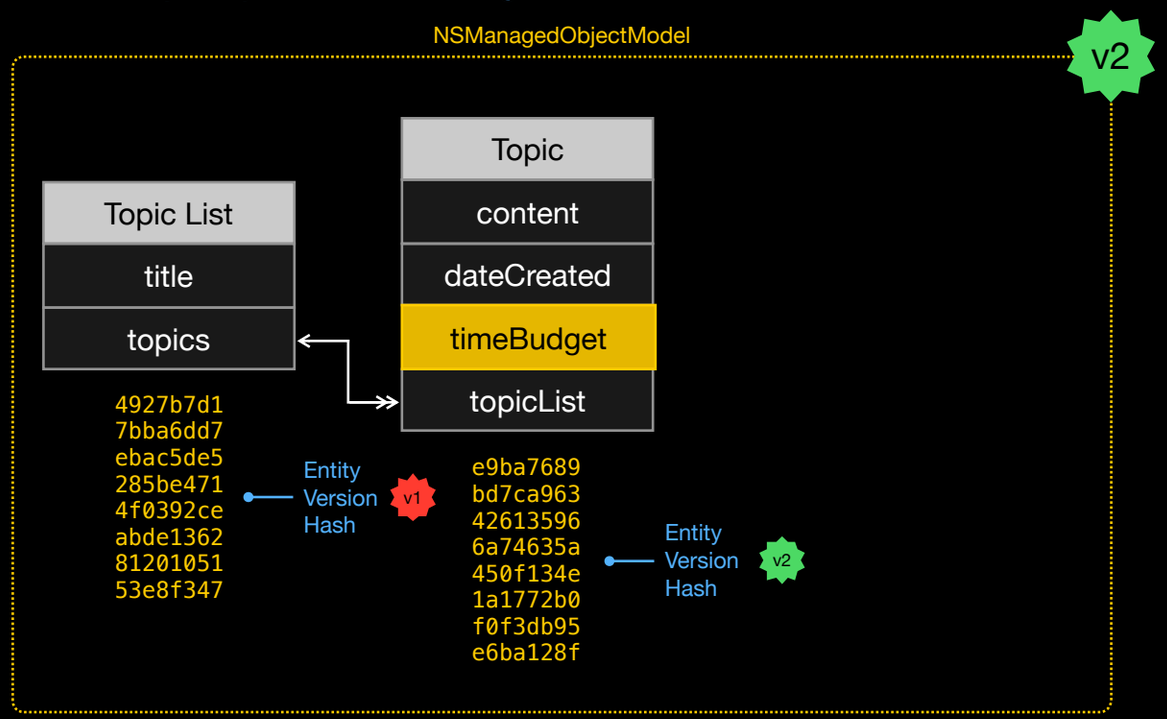
When a persistence element changes.



When we make certain model changes, the version hashes change. This effectively established a different version of our model schema.

Model Version Changes

When any entity version hash changes.



Version 2 is what we might call this schema.

Model Version

Inspecting the version information in an object model.

```
NSDictionary *entityHashes = [managedObjectModel entityVersionHashesByName];
```

```
(lldb) po entityHashes
{
    Topic = <698da797 f6cf0f56 39f882fa b3fbe868 bc8ac142 8f571ce2 e2a2eb4e 641dbc75>;
    TopicList = <4927b7d1 7bba6dd7 ebac5de5 285be471 4f0392ce abde1362 81201051 53e8f347>;
}
```

You can access the entity hashes at runtime using NSManagedObjectModel APIs.

Model Version

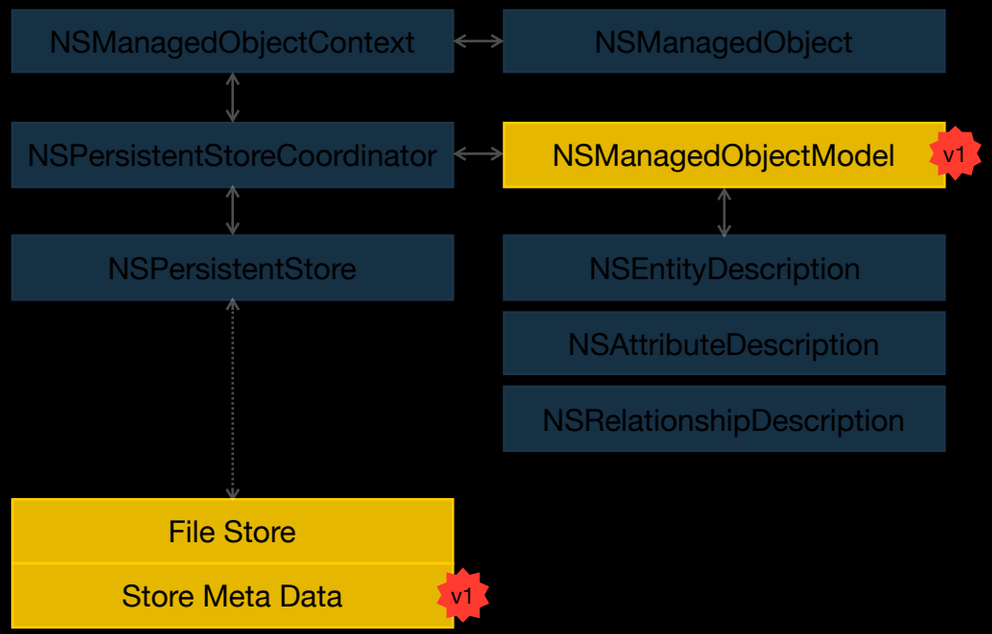
Inspecting the version information in an object model.

```
NSDictionary *entityHashes = [managedObjectModel entityVersionHashesByName];
```

```
(lldb) po entityHashes
{
    Topic = <698da797 f6cf0f56 39f882fa b3fbe868 bc8ac142 8f571ce2 e2a2eb4e 641dbc75>;
    TopicList = <4927b7d1 7bba6dd7 ebac5de5 285be471 4f0392ce abde1362 81201051 53e8f347>;
}
```

Store Meta Data

Captures version that identifies the model that created the store.



Store Meta Data is contains version information that is compared to similar version information from the managed object model. If they're not equal, they're not compatible.

Model versions are just a collection of the Entity Versions it manages.

Store Meta Data

Inspecting the version information in the store.

```
NSDictionary *storeMetaData = [NSPersistentStoreCoordinator
                                    metadataForPersistentStoreOfType:storeType
                                    URL:persistentStoreURL
                                    error:&error];
```

You can fetch the store meta data using a NSPersistentStoreCoordinator class method.
It returns an NSDictionary which includes the entity version hashes from the model that created the store.

Store Meta Data

Inspecting the version information in the store.

```
NSDictionary *storeMetaData = [NSPersistentStoreCoordinator
                                metadataForPersistentStoreOfType:storeType
                                URL:persistentStoreURL
                                error:&error];


(lldb) po storeMetaData
{
    NSPersistenceFrameworkVersion = 479;
    NSStoreModelVersionHashes =     {

    Topic = <698da797 f6cf0f56 39f882fa b3fbe868 bc8ac142 8f571ce2 e2a2eb4e 641dbc75>;

    TopicList = <4927b7d1 7bba6dd7 ebac5de5 285be471 4f0392ce abde1362 81201051 53e8f347>;
    };

  NSStoreModelVersionHashesVersion = 3;
    NSStoreModelVersionIdentifiers =     (
        Model1
    );
    NSStoreType = SQLite;
    NSStoreUUID = "8DF5B9BC-D14F-419C-97C8-391BF3B3B27A";
    "_NSAutoVacuumLevel" = 2;
}
```

Inspecting the store meta data. Note the model's version hashes. This is the version hashes of the model used to create the store.

Test for Compatibility

Core Data does this on adding the store. You can do it in code yourself.

```
// Test if our app model is compatible with the store.
- (BOOL) isStoreCompatible
{
    NSError *error = nil;

    NSDictionary *storeMetaData = [NSPersistentStoreCoordinator
                                       metadataForPersistentStoreOfType:self.storeType
                                       URL:self.persistentStoreURL
                                       error:&error];

    if (nil != error) {
       return NO;
    }

    if (nil != storeMetaData)
    {
       return [self.managedObjectModel isConfiguration:nil
                   compatibleWithStoreMetadata:storeMetaData];
    }

    return NO;
}
```

You can test for store/app model compatibility yourself.

Testing for Compatibility

Core Data does this on adding the store. You can do it in code yourself.

```
// Test if our app model is compatible with the store.
- (BOOL) isStoreCompatible
{
    NSError *error = nil;

    NSDictionary *storeMetaData = [NSPersistentStoreCoordinator
                                       metadataForPersistentStoreOfType:self.storeType
                                       URL:self.persistentStoreURL
                                       error:&error];

    if (nil != error) {
       return NO;
    }

    if (nil != storeMetaData)
    {
       return [self.managedObjectModel isConfiguration:nil
                   compatibleWithStoreMetadata:storeMetaData];
    }

    return NO;
}
```

You can test for store/app model compatibility yourself.