Atomic Store Programming Topics

Data Management: File Management

2008-02-08

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction to Atomic Store Programming Topics

This document describes how to use the atomic store API to create custom persistent stores for Core Data applications.

You should read this document if you want to create a Core Data persistent store for which you manage the file format.

This is not an entry-level subject. You must be familiar with the Core Data architecture described in Core Data Basics and Persistent Stores, and with the design and implementation of Objective-C classes. You must also understand your own file format, and how to read, parse, and write the files you want to support.

Organization of This Document

This document contains the following articles:

- "Atomic Store Fundamentals" (page 7) describes the fundamental concepts that underly the atomic store.
- "Atomic Store Life-cycle" (page 9) describes the life-cycle of an atomic store. You should read this article to learn what methods are invoked on a store when, and what the store is expected to do as a result. This will help you to understand how to implement a custom store.
- "Registering a Custom Store Type" (page 13) describes how to register a custom atomic store type.
- "Initializing a Store and Loading Data" (page 15) describes how initialize and load data into a custom atomic store.

Atomic Store Fundamentals

This article describes the fundamental concepts that underly the atomic store.

Introduction

Core Data provides four native types of persistent store:

- SQLite
- Binary
- XML
- In-Memory

These store types each offer different benefits and tradeoffs, and are described in Persistent Stores. The Binary and XML stores are "atomic" stores—they must be read and written in their entirety, unlike the SQLite store which can be modified piecemeal, one record at a time if you wish.

Core Data manages all interaction with these stores, translating insertions, deletions, and updates of managed object into, from, and in, a managed object context. Core Data also manages the file formats.

The atomic store API allows you to create a custom store format for your data, and interact with the data using standard Core Data APIs. This in turn allows you to leverage Core Data in your application but use either an existing—perhaps legacy—file format, or to define a new file format that may be used by external applications that do not use Core Data. For example, you might create a "generic" store type such as for HTML or comma-separated values, or some other intermediary type for integration with a third-party. It is important though to note that the atomic store API does not provide support for integration with client-server relational databases or similar SQL-based stores.

If you implement a custom store type, it is typically tied to a specific managed object model that describes the schema encoded in the store.

Atomic Store Classes

There are three classes you use when implementing a custom store:

- `NSPersistentStore`

  This is the abstract base class for all Core Data stores.

- `NSAtomicStore`

  This is a subclass of `NSPersistentStore`.

- `NSAtomicStoreCacheNode`

  This is a concrete class that provides a basic representation of a "node" in your store.

Of these, the only one you need to subclass is `NSAtomicStore`. It is an abstract superclass for you to subclass to create atomic stores. It provides default implementations of some utility methods. You cannot directly subclass `NSPersistentStore`. You can subclass `NSAtomicStoreCacheNode` to provide specialized behavior if necessary.

Implementation Requirements

`NSAtomicStore` is a subclass of `NSPersistentStore`. Both classes require you to implement various methods.

In a subclass of `NSPersistentStore`, you must override the following methods:

`type`	Returns the type string of the receiver.
`identifier`	Returns the unique identifier for the receiver.
`metadata`	Returns the metadata for the receiver.
`metadataForPersistentStoreWithURL:error:`	Returns the metadata from the persistent store at the given URL.

In a subclass of `NSAtomicStore`, you must override the following methods:

`load:`	Loads the cache nodes for the receiver.
`newCacheNodeForManagedObject:`	Returns a new cache node for a given managed object.
`newReferenceObjectForManagedObject:`	Returns a new reference object for a given managed object.
`save:`	Saves the cache nodes.
`updateCacheNode:fromManagedObject:`	Updates the given cache node using the values in a given managed object.

Atomic Store Life-cycle

This article describes the life-cycle of an atomic store. You should read this article to learn what methods are invoked on a store when, and what the store is expected to do as a result. This will help you to understand how to implement a custom store.

Registration

To use a custom store type in an application, you must register the store type with the `NSPersistentStoreCoordinator` class using `registerStoreClass:forStoreType:`. For more details, see "Registering a Custom Store Type" (page 13).

Initialization and Loading Data

You create an instance of store of a given type using the `NSPersistentStoreCoordinator` method `addPersistentStoreWithType:configuration:URL:options:error:`. You must implement the appropriate methods to properly initialize your store and load its contents.

Initializing the Store

The coordinator allocates a new store and initializes it using with `initWithCoordinator:configurationName:url:options:`. This method is invoked both for new stores and for existing stores; your implementation of `initWithCoordinator:configurationName:url:options:` should therefore check whether a file exists at the given URL—if it does not, you must create it.

You must also initialize the metadata for the store. After `initWithCoordinator:configurationName:url:options:`, the coordinator invokes `metadata` on the new store; at this point the metadata must be correct.

After the store is initialized, coordinator instructs the store to load its data by sending it a `load:` message.

Loading a File

The `load:` method should retrieve the store's data from the URL specified for the store in `initWithCoordinator:configurationName:url:options:`. The `load:` method should parse the contents of the store to extract the persistent data and create a reference data object—such as a dictionary—for each element.

For each object to be represented, your store must do the following:

1. Create a managed object ID with the reference data and entity type for the node, using `objectIDForEntity:referenceObject:`.

   The object ID for a cache node is also the object ID for a managed object representing the cache node.

2. Create a cache node (an instance of `NSAtomicStoreCacheNode`), using `initWithObjectID:`.

   The store must provide a mutable property cache object for each node.

3. (Optional) Push the corresponding persisted data into the node.

   This step may be omitted if you implement lazy loading or other behavior.

   The default implementation of `NSAtomicStoreCacheNode` provides key-value coding compliant access for property storage, so for the common case you do not need to create a custom subclass of `NSAtomicStoreCacheNode`.

Once all of the nodes have been created, you register them with the store using `addCacheNodes:`.

At this point, the cache now has registered nodes and the framework is ready. Core Data will handle all requests for fetches, and create managed objects from the underlying cache node information.

This process is described in greater detail, with examples, in "Initialization and Loading Data" (page 9).

Zero-Length Files

Any subclass of `NSAtomicStore` must be able to handle being initialized with a URL pointing to a zero-length file. This serves as an indicator that a new store is to be constructed at the specified location and allows you to securely create reservation files in known locations which can then be passed to Core Data to construct stores.

You may choose to create zero-length reservation files during an `initWithCoordinator:configurationName:url:options:` or `load:` method. If you do so, you must remove the reservation file if the store is removed from the coordinator before it is saved.

Processing and Saving Changes

When a managed object context is sent a `save:` message, any associated store must (in this order) process all of the inserted, updated, and deleted objects that are assigned to it (the store), and then commit any changes to the external repository.

1. *New objects*

   For each newly-inserted object in the context, the store receives a `newReferenceObjectForManagedObject:` message. Your store must provide a unique—and unchanging—value for this instance and entity type for the store (see "New Reference Objects" (page 11)).

After the reference data is returned, the store requests a new cache node be created for each of these objects using `newCacheNodeForManagedObject:`.

`NSAtomicStore` provides a default implementation that should suffice in most cases, However, if you wish to customize the creation of the cache nodes or use custom cache nodes, you can override `newCacheNodeForManagedObject:`. In your implementation, you should create the cache node with the managed object ID from included managed object, and the values to be persisted should be pushed from the managed object into the cache node. You then register the new nodes with the store, using `addCacheNodes:`

2. *Updated objects*

   For all objects which are changed in the context, the store receives a `updateCacheNode:fromManagedObject:` message. The arguments for the method are the cache node and managed object pair (which share the same object ID.) The store must push the persisted values from the managed object into the cache node.

3. *Deleted objects*

   Your store receives a `willRemoveCacheNodes:` callback; if necessary you can override this method to break any retain cycles to ensure that deleted objects are reclaimed.

4. *Saving changes*

   Finally, the store receives a `save:` message to commit the changes to the external repository.

   Your store must write the data (the cache nodes) and the metadata to the URL specified for the store in whatever format it uses.

New Reference Objects

Your implementation of `newReferenceObjectForManagedObject:` must return a stable (unchanging) value for any given object. If you don't do this, then Save As and migration operations will not work correctly.

This means that you can only use arbitrary numbers, UUIDs, or other random values if you save the value together with the rest of the data for a given object. If you cannot save the originally-assigned reference object, then `newReferenceObjectForManagedObject:` must derive the reference object from the managed object's values. As this constraint exists for Save As and store migration operations, you could also use a side table or in-memory structure to ensure that a newly-migrated store loads its atomic store cache nodes with the same objectIDs that the current managed object context is using for the migrated managed objects.

Removal and Deallocation

If a store is to be removed from the coordinator, it receives the message `willRemoveFromPersistentStoreCoordinator:`. In a custom atomic store class, you can override this method to implement clean-up or other behavior. If you do, your implementation of `willRemoveFromPersistentStoreCoordinator:` should invoke super's implementation.

Note that the coordinator parameter may be `nil` if:

- There was an error during the load: method, and the store cannot be added to the coordinator
- The coordinator is in the process of being deallocated or finalized.

In these cases, `willRemoveFromPersistentStoreCoordinator:` is invoked to allow you to perform any necessary resource cleanup (such as removal of reservation files—see "Zero-Length Files" (page 10)).

Registering a Custom Store Type

To use a custom store type in an application, you must register the store type with the `NSPersistentStoreCoordinator` class using `registerStoreClass:forStoreType:`. The type name must be a unique string.

Typically you define the store type as a string constant:

```
// in MyAtomicStore.h
extern NSString *MY_ATOMIC_STORE_TYPE;

// in MyAtomicStore.m
NSString *MY_ATOMIC_STORE_TYPE = @"MyAtomicStore";
```

You should ensure that a type is registered before you try to add a store of that type to a persistent store coordinator. For example, in an application delegate you might register in `applicationWillFinishLaunching:`.

```
- (void)applicationWillFinishLaunching:(NSNotification *)aNotification
{
    [NSPersistentStoreCoordinator registerStoreClass:[MyAtomicStore class]
                                        forStoreType:MY_ATOMIC_STORE_TYPE];
}
```

In a document-based application, the type is the identifier `NSPersistentDocument` uses to associate a persistent store type with a document type (see `persistentStoreTypeForFileType:`).

Initializing a Store and Loading Data

This article provides details of how to initialize a persistent store and to load data from an URL.

Initializing a Store

When you add a store to a persistent store coordinator, the store is initialized with `initWithCoordinator:configurationName:url:options:`. After a store is initialized, it receives a request from the persistent store coordinator to load its data through the `load:` method.

You are not required to implement `initWithCoordinator:configurationName:url:options:`—in some situations the default implementation will suffice. If you do need to provide additional initialization, however, you need to consider that `initWithCoordinator:configurationName:url:options:` is invoked both for existing stores and for new stores.

Your implementation of `initWithCoordinator:configurationName:url:options:` must be able to handle being initialized with a `nil` URL and an URL that points to a zero-length file. The latter serves as an indicator that a new store is to be constructed at the specified location and allows you to securely create reservation files in known locations which can then be passed to Core Data to construct stores.

Loading a File

The `load:` method is responsible for retrieving the data from the URL specified for the store and creating the necessary objects. For each element in the store, you must create:

- A reference data object that contains the data from the element
- A managed object ID
- An atomic store cache node

Once all of the nodes have been created, you register them with the store using `addCacheNodes:`.

As an example, consider a store with the following contents:

```
Person,Melissa,Turner,86349382003
Person,Jesus H.,Keenan,5987749473
Person,Ben,Trumbull,7987082467
```

Reference Data

The reference data object is simply an object representation of the data in the store. For example, you might create an `NSMutableDictionary` object for each Person:

```
{ firstName = @"Melissa", lastName = @"Turner", id = 86349382003 }
{ firstName = @"Jesus H.", lastName = @"Keenan" id = 5987749473 }
{ firstName = @"Ben", lastName = @"Trumbull" id = 7987082467 }
```

Managed Object ID

You create a managed object ID using `objectIDForEntity:referenceObject:`, passing entity and reference data for the node. You can get the entity from the managed object model associated with the store's persistent store coordinator, for example:

```
NSEntityDescription *personEntity =
    [[[[self persistentStoreCoordinator] managedObjectModel]
        entitiesByName] objectForKey:@"Person"];
NSManagedObjectID *moID =
    [self objectIDForEntity:personEntity referenceObject:personDictionary];
```

The object ID for a cache node is also the object ID for a managed object representing the cache node.

Cache node

You create a cache node using the `NSAtomicStoreCacheNode` method `initWithObjectID:`.

```
NSAtomicStoreCacheNode *personNode =
                [[NSAtomicStoreCacheNode alloc] initWithObjectID:moID];
```

After creating each node, you usually push the corresponding persisted data into the node (although you can implement lazy loading or other behavior).

The cache node uses a mutable dictionary as a backing store. If when you parse the data you create instances of `NSMutableDictionary` to represent each element, and the keys in the dictionary are the property names of the corresponding entity, then you can set the cache node directly:

```
[personNode setPropertyCache:personDictionary];
```

The values in cache node must be:

- For an attribute value, instance of an attribute type supported by Core Data (see `NSAttributeDescription`);
- For a to-one relationship, another cache node instance;
- For a to-many relationship, a collection of the related cache nodes.

Example

Consider a store with the following contents:

```
Person::first:Melissa:last:Turner:id:86349382003
Person::first:Jesus H.:last:Keenan:id:5987749473
Person::first:Ben:last:Trumbull:id:7987082467
```

The `load:` method might first parse the file to create an `NSMutableDictionary` object for each Person:

```
{ firstName = @"Melissa", lastName = @"Turner", id = 86349382003 }
{ firstName = @"Jesus H.", lastName = @"Keenan" id = 5987749473 }
{ firstName = @"Ben", lastName = @"Trumbull" id = 7987082467 }
```

and collect them in an array, `personDictionaries`. In this case, the keys used in the dictionary are the same as the property names for the Person entity.

You then create the Core Data objects as follows:

```
NSEntityDescription *personEntity =
    [[[[self persistentStoreCoordinator] managedObjectModel]
        entitiesByName] objectForKey:@"Person"];

NSMutableSet *cacheNodes = [NSMutableSet set];

NSDictionary *referenceObject;
NSManagedObjectID *moID;
NSAtomicStoreCacheNode *personNode;

for (NSMutableDictionary *personDictionary in personDictionaries) {

    // not sure if you need to make a copy?
    referenceObject = [personDictionary copy];
    moID = [self objectIDForEntity:personEntity
                           referenceObject:referenceObject];
    [referenceObject release];

    personNode = [[NSAtomicStoreCacheNode alloc] initWithObjectID:moID];
    [personNode setPropertyCache:personDictionary];
    [cacheNodes addObject:personNode];
    [personNode release];
}

[self addCacheNodes:cacheNodes];
```

Document Revision History

This table describes the changes to *Atomic Store Programming Topics*.

Date	Notes
2008-02-08	Corrected typographical errors.
2007-07-24	New document that describes how to use the Core Data atomic store API.