Sync Services Programming Guide

Data Management: Syncing

2009-08-06

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Sync Services Programming Guide

Sync Services is a framework containing all the components you need to sync your applications and devices. If your application uses Sync Services, user data can be synced with other applications and devices on the same computer, or other computers over the network via MobileMe. Ideally, all Mac OS X applications should sync user data quickly and quietly in the background. Consequently, user data is available when and where the user wants it.

Important: If your application uses the Sync Services and Address Book frameworks together, then you should not use Sync Services to sync data shared with the Address Book Framework. The Address Book Framework already syncs its records with Sync Services, so applications sharing the Address Book data do not have to (and should not) sync those records. The results are unpredictable and may result in data loss, if you attempt to sync the same data as the Address Book Framework.

Concurrency Note: The shared `ISyncManager` object in the Sync Services framework is thread safe. No additional locking or other synchronization is required when using `ISyncManager` methods. However, other objects vended by Sync Services should be used in the thread in which they were created. Typically an `ISyncSession` object is used by the thread in which it is created.

Who Should Read This Document?

You should read this document if you want to sync your application's data. Types of applications include end-user applications, tools, or servers. For example, a tool may be an interface to a device (such as a phone) or a framework that maintains persistent data. You can use existing schemas for contacts, calendars, and bookmarks. You can also extend a schema or create your own schema to sync custom objects. You can use the Sync Services API from both Objective-C and C programs.

Organization of This Document

Before using Sync Services you should understand what it is used for, what the system architecture is, and what the core classes are. Because sync sessions are implemented as finite state machines, you also need an in-depth understanding of the states and transactions within a sync session regardless of which approach you choose. The consequences of syncing incorrectly are severe—users may lose their data—so read the following conceptual articles before using this framework. In particular, you need to read "Managing Your Sync Session" (page 25) if you are writing your own sync methods.

- "Why Use Sync Services?" (page 13) explains why syncing is an important feature to end users and may become commonplace on Mac OS X.
- "Sync Services Overview" (page 17) describes the components of the sync architecture and defines syncing terms such as those used for the different sync modes.

- "Managing Your Sync Session" (page 25) describes the anatomy of an `ISyncSession` object. It contains details about each state in the finite state machine and the methods that can be used in each state.

A simple approach to syncing is to use a delegation model where a driver sends messages to a data source and delegate while syncing. You can also use Core Data as the persistent store for local records and to create your schema. Read the following articles to learn more about these approaches:

- "Using a Session Driver" (page 77) describes how to use an `ISyncSessionDriver` object to control a sync session.
- "Syncing Core Data Applications" (page 85) describes how to create a Core Data application that syncs managed objects.

If you need more control, read the following articles that cover the low-level classes and methods you use to manage sync sessions:

- "Creating a Sync Schema" (page 39) describes how to create your own custom schemas and contains a description of the sync schema property list.
- "Registering Schemas" (page 55) describes how to register and unregister schemas.
- "Registering Clients" (page 57) describes how to register and unregister clients, as well as provides a description of the client description property list.
- "Syncing Relationships" (page 61) describes how to add relationships to your object model and schema. It also contains tips on pushing and pulling relationships.
- "Syncing with Other Clients" (page 65) describes how to configure your client to sync with other clients.
- "Using Sync Anchors" (page 67) describes how to use sync anchors to improve performance and reliability.
- "Filtering Records" (page 71) describes how to filter the types of records your application syncs.
- "Formatting Records" (page 75) describes how to change the format of a record without the sync engine interpreting the change as a change to the property values.

Read this article if you want to exclude your application preferences from syncing:

- "Syncing Preferences" (page 91) describes how to exclude selected preferences or all preferences of an application from syncing when the user turns this feature on.

See Also

For an in-depth description of the Sync Services API, read:

- *Sync Services Framework Reference*

If you are using or extending an Apple Applications schema, such as contacts, bookmarks or calendars, read:

- *Apple Applications Schema Reference*

If you are accessing Sync Services from a C application, refer to the *SeeMyFriends* sample code and this document for more information about mixing C and Objective-C:

- *Carbon-Cocoa Integration Guide*

Why Use Sync Services?

Applications that sync allow users to access their data when and where they want it. Applications can sync with other applications and devices on the same computer, or other computers over the network via MobileMe. Ideally, user data is synced automatically without the user even thinking about it.

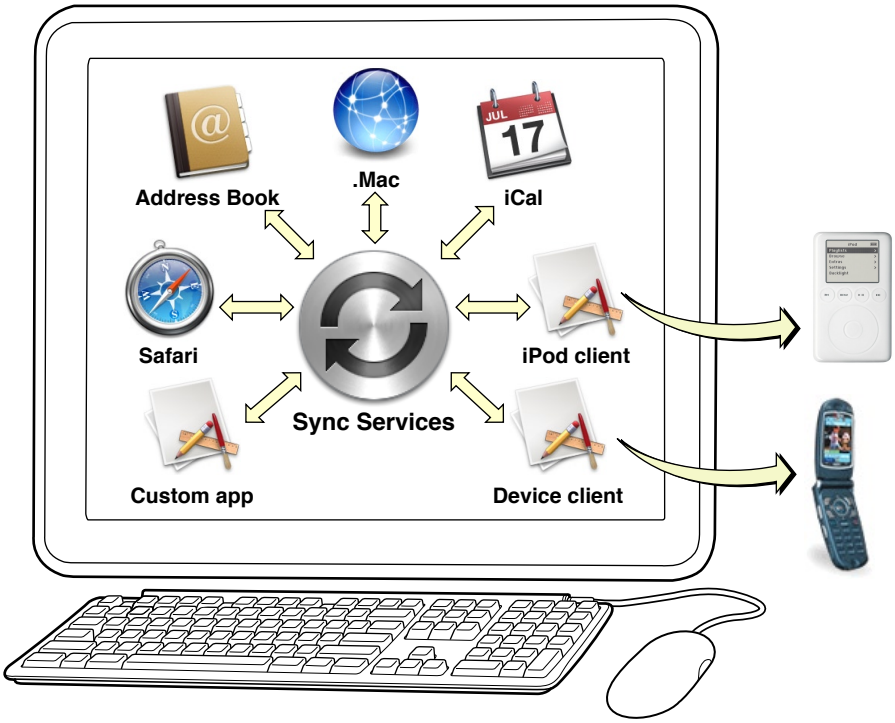
Benefits to Users

Users might have several computers, at home and at work, an iPhone, an iPod and other cell phones. Users will want to automatically sync data on all their computers and devices—especially, contacts and calendars which are supported on most devices.

For example, users may want to sync their phone and iPod devices with Address Book and iCal (see Figure 1 (page 14)). If they have a MobileMe account, they might want to sync their contacts, calendars, and bookmarks across multiple computers. But why stop there? They might want to sync their music, photos, some arbitrary folders, and Mail rules, too. Even large enterprises such as companies and universities might want to sync their custom objects across the network. Syncing is a service available to all applications, not just Apple applications.

The goal is for syncing to be ubiquitous, available to all as just another service on Mac OS X, so that users don't have to think about syncing; all applications, devices, and computers should sync quickly and quietly in the background.

Figure 1 Syncing your data



Benefits to Developers

Sync Services is a framework for Mac OS X developers who want to sync user data. Users can sync new devices using existing schemas, extend existing schemas, or sync custom objects. All user data can be synced via MobileMe, too—custom schemas automatically appear in the Sync pane of MobileMe preferences in System Preferences. Sync Services has the following features:

- It's a system service that's available everywhere.
- It's under the control of the application to sync.
- It's reliable—when used properly it doesn't lose user data.
- It's lightweight enough for an application to sync frequently.
- It allows multiple applications and devices to sync simultaneously.
- It can filter both entities and properties (records and fields) defined in a schema.
- It's not limited to existing schemas. You may extend existing schemas or create your own.

Ideally all applications and devices sync small sets of changes frequently. This process is called **trickle syncing**. For example, Address Book and iCal might initiate a sync each time they save their records to disk in order to push recent changes. Applications such as the MobileMe client might sync every 5 minutes to pull local changes and push remote changes made by other MobileMe clients running on other computers. The order in which applications sync, and the frequency of that syncing, doesn't matter as long as the syncs are fast and efficient.

Future User Environment

When Sync Services is used by most Mac OS X applications, a user's computer will behave something like this. Imagine that the user adds a new contact to her phone and independently adds a new event to iCal and a new contact to Address Book.

The user connects her iPod to her computer, which syncs. The new iCal event is added to the iPod. The user connects her phone and it syncs, too. The new iCal event and Address Book contact are added to the phone.

Syncing the phone may alert MobileMe and other applications that sync the same types of objects—for example, objects such as calendars and contacts. Address Book then syncs and adds the new phone contact. MobileMe is also alerted: It adds the new phone contact, Address Book contact, and iCal event.

Syncing MobileMe may apply changes to contacts and calendars made on other computers. Since Address Book, iCal, and the phone are all syncing together, they may apply these changes when MobileMe syncs. Sync Services merges all the changes and computes which changes need to be applied by which applications.

Joining a sync session is optional—for example, an application may decline an invitation if the user is busy making other changes. If so, it can just sync later.

If applications periodically sync and join related sync sessions, the user is unaware that their data is being synced—the data just appears in the right place when the user wants it. For example, the iPod was not actually part of the simultaneous sync session described above but will get the MobileMe changes if it periodically syncs.

Sync Services Overview

The primary goals of Sync Services are for syncing to be efficient and unobtrusive to the user. Most applications are expected to sync often, with small sets of changes. Applications may sync simultaneously with other applications. A sync operation can be interrupted or postponed, without data loss, to respond to user requests.

To achieve this, the Sync Services uses a finite state machine to manage a sync session. The API reveals this state machine, so developers have fine control over each state. Hence, the API is more flexible to meet the needs of diverse applications.

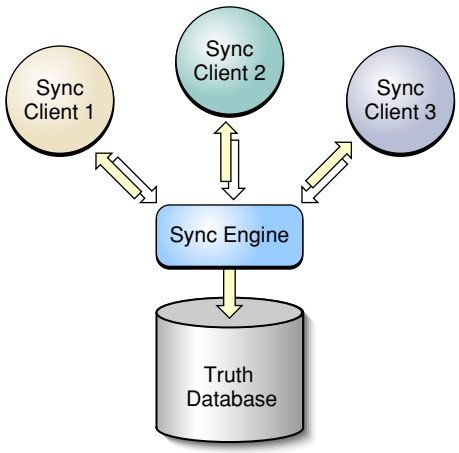
Because of the state machine, the core classes are more complex than typical Cocoa classes. The order in which methods are invoked, and the timing, are important. In fact, the consequences of using this API incorrectly are dire—the users may lose their data. Therefore, using Sync Services requires a deeper understanding of syncing concepts.

Syncing is also a coordinated effort between a number of client processes and a server running on a single computer. To use Sync Services, you first need to understand its architecture and the role of each component in that architecture. Then you need to understand sync modes and how to manage a sync session (covered in detail in "Managing Your Sync Session" (page 25)).

Sync Services Architecture

The Sync Services architecture is depicted in Figure 1 (page 17). All the processes and the truth database reside on a single computer. For each user account, there is one sync engine and one truth database. The **sync engine** is responsible for coordinating and synchronizing data between multiple clients. The sync engine stores the aggregate of all client records in the **truth database**. The sync engine is the only process that accesses the truth database directly.

Figure 1 Sync Services architecture



This architecture is primarily a **client server model** in which **clients** initiate syncs by pushing changes to the sync engine and pulling changes—possibly made by other clients—from the sync engine which is the **server**. Clients sync when they want to by communicating directly with the sync engine. For example, if the user made local changes, a client might batch the changes and sync them immediately. If the user is busy with some operation, the client might postpone syncing until later when the client is idle. Clients should never sync directly with another client or through another process.

There are multiple types of clients—for example, your custom **applications**, iCal and Address Book, **tools** for syncing devices such as phones, iPods, and other PDAs, and **servers** for syncing data over the network such as MobileMe. Clients must be able to transform sync engine record formats into formats used by devices which might have limited storage capacity. Sync Services can also support extensions to existing schemas and custom schemas, not just those data records used by Apple products.

Sync Engine

The **sync engine** does the bulk of the work merging changes and computing changes to be pulled by different clients. It is lightweight enough for clients to synchronize frequently—as often as once a minute if need be—and coordinate the requests of multiple clients simultaneously. It can notify a dependent client that an observed client is syncing, and allow that client to join the sync session. How does the sync engine achieve all this without the user having to think about syncing?

The sync engine selects the appropriate **sync mode** for each client, depending on the situation. Selecting a sync mode is typically a negotiation between the client and the sync engine. This is because syncing depends on the state of all the processes involved in the sync, not just on a single client. And because multiple clients may join the same sync session and have different requirements, a client is not necessarily given the sync mode it requests. Instead, the sync engine may force another mode.

The first time a client syncs, it pushes all its records to the sync engine and pulls changes computed by the sync engine. This sync mode is called **slow syncing**. While a client is pushing and pulling records, the sync engine keeps track of the client's state using a snapshot so that subsequent syncs can be more efficient. The next time a client syncs, only changes are pushed and pulled. This sync mode is called **fast syncing.**

Typically, the sync engine assumes the client is fast syncing unless the client negotiates another sync mode or some state has changed in the server or client that requires a different mode.

For example, if a device is reset, it may lose all its records. When this happens, a client can request a **refresh sync**. A refresh sync tells the sync engine to forget everything it knew about a client—the engine will remove the client snapshot. Typically, the client then pulls all the records and pushes none. A refresh sync is a slow process that should be performed only after a catastrophic event, such as the user manually resetting a device or deleting an application's data files.

Sometimes, a client might want to replace all the truth database records with its own records. This mode is called **push the truth**. In contrast, a client might want to replace all its records with the records in the truth database. This mode, called **pull the truth**, can be initiated by the client or the sync engine. If a client initiates a sync and then pushes the truth, the sync engine may force all other clients participating in the sync to pull the truth.

Ideally, it's transparent to the user whether or not a client is slow syncing or fast syncing, or how often it syncs. Most clients should **trickle sync**—that is, fast sync frequently and simultaneously with related clients. Dependent clients should just sync automatically without requiring human intervention.

Intelligence is built into the sync engine to resolve most conflicts and duplicates without requiring user input. The bulk of the sync engine's job is to merge changes from multiple clients and, upon request, to give your client only the changes it needs. The sync engine is a field-differencing engine that processes changes to individual fields in a record, not just changes to records. If two clients modify different fields in the same record, the engine can merge the changes successfully. But if two clients modify the same field on a record, the engine generates a conflict.

The sync engine may also reduce a complicated sequence of changes into simpler ones if changes from multiple clients are redundant. For this reason, clients should not rely on the order in which changes are applied.

The Truth Database

The **truth database** contains an aggregate of all the client's records. Consequently, the truth database uses a canonical schema that is an aggregate of all the schemas used by all the clients.

A sync schema is based on an entity-relationship model similar to that used by other Cocoa technologies. Read Cocoa Design Patterns in *Cocoa Fundamentals Guide* to learn more about entity-relationship models and terms such as entity, property, attribute, relationship, to-one, and to-many.

You can use one of the existing sync schemas—for example, for contacts, calendars, and bookmarks— extend one of these schemas or create your own. If you extend a schema or create your own, then you need to create an entity model for your custom objects and save it in a schema format that Sync Services understands.

This format, called **sync schema**, is a property list that specifies details about the entities in your model. For entities, you might specify the name of the entity and names of its attributes and relationships (collectively referred to as its **properties**). For attributes, you might specify the name of the attribute and its data type. For relationships, you might specify the name, destination entity name, cardinality, and delete rule. See "Creating a Sync Schema" (page 39) for a complete description of the sync schema property list.

A sync schema defines a template for records stored in a database whose records are of a particular type (records belong to an entity) and may have relationships to other records. Records stored in the truth database are dictionary objects with key-value pairs, one for each property defined in the entity. Each record dictionary also has an entity name property and an associated unique record identifier. The record identifier is not stored with the record but is instead used by the client and sync engine when referring to a record. The truth database can also store custom fields in a record that are not defined in the schema. For example, these fields can be used to store client information added by a device.

The truth database doesn't store arbitrary key-value coding-compliant objects—it stores record dictionaries. Therefore, unless all your entities are dictionaries, you typically transform records back and forth between the sync engine's record representation and your client's object representation. However, when fast syncing, you can apply changes only to properties—you don't need to push and pull entire records when only a few property values changed.

Because the truth database is an aggregate of all the client schemas, it can contain a lot of information that your application doesn't care about. Your client can filter the records that it pushes and pulls in several different ways.

Clients

Typically, a **client** is an end-user applications (like your Cocoa application, iCal, or Address Book), a server applications (like MobileMe), or a tool that syncs a device (such as a phone). A **device client** is a liaison between the sync engine and a device. A server client is a liaison for a remote server that stores user data. Actually, a client can be any application or tool running on Mac OS X that uses an existing schema, extends an existing schema or defines its own. Sync Services can be used to sync your custom objects. Clients may vary dramatically in their capacity to store records and in the complexity of their object models. For example:

- Some clients cannot support all entities and properties defined in a schema.
- Some device clients may restrict the format or limit the lengths of attributes (for example, truncate string values).
- During any given sync session, a client might want to push or pull only a subset of the entities it supports. For example, a phone device client might pull only calendar events that occur during the next two weeks.
- Some clients may not be capable of modifying records such as device clients for an iPod or a phone. Other clients might have full read and write access to the same records.

The sync engine is flexible enough to support a diverse set of clients as long as the clients do their part. Clients need to specify their capabilities, filter records, format records, and resolve conflicts when they arise. Any client that wants to sync records using Sync Services has several responsibilities, described next.

Registering Schemas

If a client extends an existing schema or defines its own schema, it needs to register that schema with the sync engine. Ideally, a client should register a schema once and thereafter reregister it only if it changed. (Reregistering the same schema is harmless if the schema is unchanged.)

Changing schemas should be done cautiously and infrequently. Changing schemas can cause data loss if entities and properties are removed. It can also require that all clients using the schema slow sync during the next sync.

See "Creating a Sync Schema" (page 39) for more information on schema formats, and "Registering Schemas" (page 55) for information on registering schemas.

Describing Client Capabilities

A client must provide the sync engine with a description of its capabilities. At a minimum, the client must specify the schemas it uses, and the entities and properties in those schemas that it supports. You can also specify which entities are read-only and which entities are read-write allowing the sync engine to skip the pushing and mingling states when it can. The client description is one way a client can filter records.

A client description is stored in a property list file specified when registering the client with the sync engine. The sync engine periodically checks this file for changes and forces the client to slow sync on the next sync if the property list changes.

Note that the client description specifies the entities and properties that a client can support but not necessarily the entities and properties that a client can sync. The entities and properties that a client can sync must be a subset of the supported ones.

See "Registering Clients" (page 57) for more information on registering a client.

Syncing

Most often it is the responsibility of the client to initiate syncs. A client should trickle sync—periodically fast sync in the background—or allow the user to manually sync the application as needed.

In contrast, a client may not initiate a sync if it shares a schema with another application. If a client is an observer of another client, it may be alerted by the sync engine to sync when the other client syncs. For example, a client that uses calendar records might receive an alert when iCal syncs. When alerted, the client can optionally join the sync session.

You can also specify a tool to be launched if an observed client syncs, so that your application doesn't need to be running in order to sync. For example, an Address Book tool syncs, even if the Address Book application is not running, whenever MobileMe syncs.

The client is also responsible for managing the sync session and performing the sync operations in the expected sequence: that is, negotiate a sync mode, push changes to the sync engine, and pull changes from the sync engine. See "Managing Your Sync Session" (page 25) for more information on managing a sync session.

Recording Changes

If a client wants to fast sync when pushing records, the client needs to keep track of changes made to its local data. When fast syncing, clients need to inform the sync engine which records and, optionally, which properties changed—syncs will be faster if you tell the engine which properties changed. Clients also need to inform the sync engine of added and deleted records.

Device clients might be able to obtain this information from the device itself. Otherwise, clients that use their own data stores need to record this information when users change records. If your client cannot provide this information, then it should slow sync each time it syncs.

Filtering Records

Clients may specify filters that are applied to records pulled from the sync engine. A filter simply conforms to a filtering protocol that takes a record and either accepts or rejects that record. A client does not see rejected records. You can program business logic into the filters. You can also apply logical `AND` and `OR` binary operators to a set of filters creating composite filters. See "Filtering Records" (page 71) for more information on setting filters.

Formatting Records

It's the clients responsibility to inform the sync engine of records that are "reformatted" by a device client. For example, if a device has limited data storage and truncates all first and last names to 20 characters, then the sync engine needs to know what the new device format is. Otherwise, the sync engine assumes the records changed on subsequent pushes and issues false changes to other clients. See "Formatting Records" (page 75) for more information on formatting records.

Resolving Conflicts

Conflicts can occur if two records that are "logically" the same record are added by different clients. In this case, the sync engine should recognize that these are changes to the same record, not two different records. The sync engine can do this if clients specify the properties that are used to identify a record in the sync schema. The sync engine uses these identity properties to resolve conflicts without duplicating records. See "Identity Properties" (page 49) for more information on identity properties.

For example, a device pushes a "John Smith" contact but there's already a record for "John Smith" in the truth database. If the sync schema specifies that the `firstName`, `lastName`, and `dateOfBirth` properties be used to identify a record, then the sync engine correctly determines that these two records are the same record, and does not generate a duplicate.

Core Classes

The Sync Services API consists of a few classes and protocols that work together to perform syncing. Sync Services is a low-level API that offers great control and flexibility. Consequently, it requires a more in-depth understanding of the syncing process in order to use it. You need to write your own sync methods that sync your data the way you want. To do this, it's important to understand the purpose of each class and how you use it. You can see this information in the list that follows.

`ISyncManager`
: You use an `ISyncManager` object to communicate directly with the sync engine. You primarily use an `ISyncManager` to register schemas and clients. There's only one `ISyncManager` shared instance per process.

`ISyncClient`
: An `ISyncClient` object encapsulates information about your client that the sync engine uses to identify your client, determine its capabilities, and maintain its state.

`ISyncSession`
: An `ISyncSession` object manages a single sync operation. You create an `ISyncSession` object, use it to sync your records, and then throw it away. `ISyncSession` supports multiple sync modes that you negotiate before pushing and pulling records.

`ISyncChange`
: An `ISyncChange` object encapsulates a set of changes related to a single record such as adding a record, deleting a record, and modifying an existing record. You use `ISyncChange` objects to push and pull changes.

You always use the shared `ISyncManager` instance and register your schema with it. Sync Services provides three canonical schemas: Bookmarks, Contacts, and Calendars. You can extend one of these schemas or register your own (read "Creating a Sync Schema" (page 39) for how to design your own schema). Your schemas must be registered before beginning any sync operations (read "Registering Schemas" (page 55)).

Typically, you create one `ISyncClient` object per application or tool and then register it with the `ISyncManager` (read "Registering Clients" (page 57)). However, if your application syncs multiple devices or data files, you could have a client per device or data file.

You create an `ISyncSession` object each time you sync your records. `ISyncSession` is implemented as a finite state machine, so the order in which you invoke its methods is critical. Read "Managing Your Sync Session" (page 25) for an in-depth discussion of syncing.

Alternatively, you can use the `ISyncSessionDriver` class that uses a delegation model to control a sync session without using any of the core classes directly. Read "Using a Session Driver" (page 77) for more information on this approach.

Managing Your Sync Session

In the simplest form of syncing, the client pushes changes to the sync engine, the sync engine applies those changes to the truth database, and the client pulls changes from the sync engine that were made by other clients since the last sync. In reality, syncing is not that simple, because events can take place, on either the client or server, that change the sync mode. Understanding how to manage different sync modes is fundamental to using the Sync Services framework.

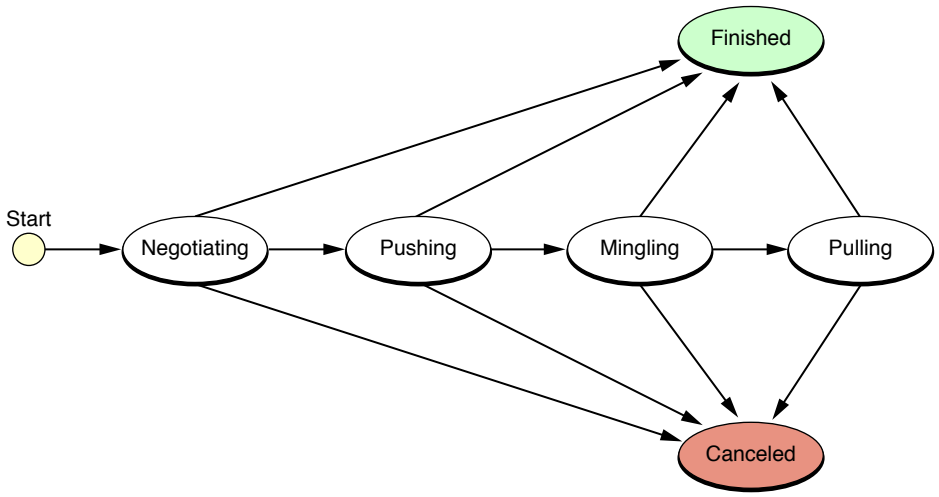
An `ISyncSession` object encapsulates the process of a single sync operation. Sync Services uses a client-server model analogous to a database model. The client sends read and write instructions within a transaction to the sync engine using an `ISyncSession` object. The `ISyncSession` object behaves as finite state machine. For example, pushing and pulling changes are different states, and you always push before you pull. There are implicit transactions within each state, too, and sometimes multiple transactions within a state. Invoking some `ISyncSession` methods may begin a transaction, end a transaction, or transition to another state. It's important to understand these states and transactions when managing a sync session.

This chapter describes how to use the core `ISyncSession` methods to manage your sync session. Refer to "Sync Services Overview" (page 17) if you are unfamiliar with the sync architecture and terminology.

Finite State Machine

In a sync session, you essentially create an `ISyncSession` object, use it to sync your records, and then release it. A session object behaves as a finite state machine illustrated in Figure 1 (page 25). The states always change in the direction depicted by the arrows in this figure.

Figure 1 ISyncSession finite state machine



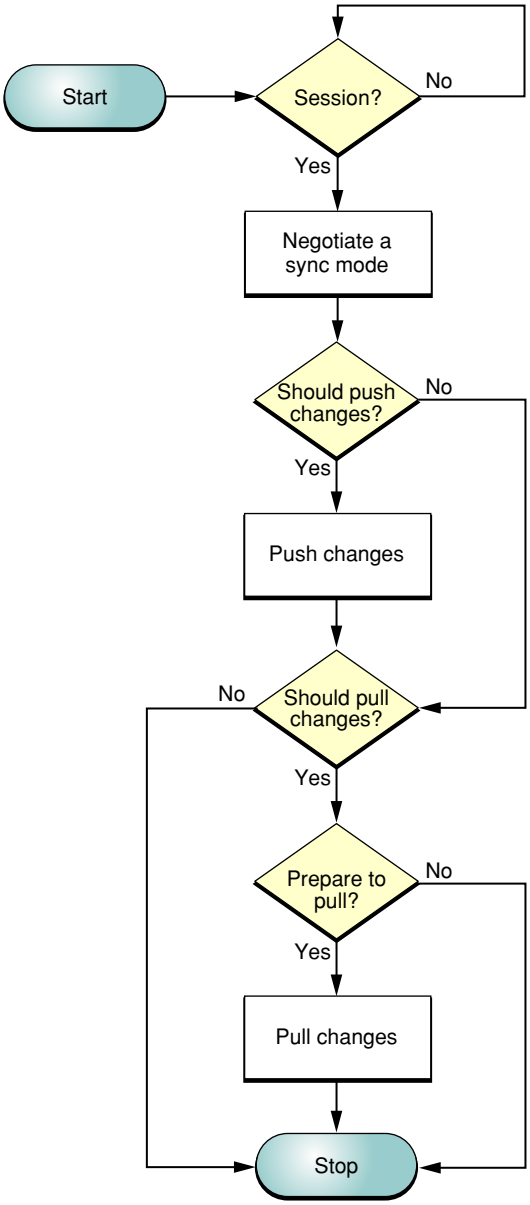
Only certain `ISyncSession` methods may be used within a state; invoking other methods in a state may raise an exception. You also transition from one state to another explicitly or implicitly depending on the methods you invoke. Negotiation and mingling are mandatory states, but pushing and pulling are optional as long as you push records before you pull records. Finishing or canceling terminates a session.

The exact sequence of messages sent to an `ISyncSession` object depends on your application and the way you store your records. This article outlines the typical sequence of messages for syncing your data and contains sample code to help you write your own custom sync methods to meet the needs of your application or device.

Implementing Your Sync Method

Figure 2 (page 27) shows the logic of a typical sync method implementation. After creating a sync session and negotiating a sync mode, you ask the sync engine if you should push changes for an entity. If the answer is yes, you push the changes; otherwise, you skip to the pulling state. Similarly, when pulling changes, you ask the sync engine if you should pull changes. If the answer is yes, you prepare to pull changes which runs the mingler. The **mingler** is a process in the sync engine that computes the changes to be pulled by each client that participates in a sync session. When the mingler completes this process, you pull the changes; otherwise, you can abort the sync session. The methods you use to implement your sync methods are described next.

Figure 2 Typical sync logic



Starting a Sync Session

The sync engine won't let your application sync when syncing is disabled by the user or some other event, and when the sync engine is waiting for other clients to join a sync session.

Is the Sync Engine Enabled?

Before beginning any sync operation, you first determine whether the sync engine is disabled as follows:

```
if ([[ISyncManager sharedManager] isEnabled] == YES) {
    // begin syncing
    ...
}
```

If the sync engine is disabled, you can optionally register for the `ISyncAvailabilityChangedNotification` notification to sync later as follows:

```
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(syncWasEnabled:) name:ISyncAvailabilityChangedNotification
object:@"YES"];
```

When registering for this notification, pass either "YES" or "NO" as the notification object. If you only want to be notified when the sync engine is enabled, pass “YES” as the notification object.

Note: If you do not instantiate the shared manager object by invoking the `sharedManager` class method, register this notification with the distributed notification instead.

Beginning a Sync Session

Typically, a client syncs its records independent of other clients. Sync Services also supports multiple clients syncing simultaneously. A client registers itself as an observer of another client and is alerted when that client syncs. For example, a device client for a phone might want to sync whenever Address Book syncs to update its contacts. Therefore, when you begin a sync session, the sync engine needs to alert all dependent clients and wait until those clients either join the sync session or don’t. Because this process may block your client momentarily, you have the option of specifying how long you are willing to wait for other clients to join the session.

There are two ways to create a sync session. You begin a `ISyncSession` object using one of two `ISyncSession` class methods:

```
beginSessionWithClient:entityNames:beforeDate:
beginSessionInBackgroundWithClient:entityNames:target:selector:
```

The `beginSessionWithClient:entityNames:beforeDate:` method is a blocking method that returns when all dependent clients have had the opportunity to join the sync session, as follows:

```
ISyncSession *session =
        [ISyncSession beginSessionWithClient:client
                entityNames:entityNames
                beforeDate:[NSDate dateWithTimeIntervalSinceNow:5]];
```

If you change the *`beforeDate`* argument to the current date or a past date, this method returns immediately with or without creating a sync session.

The `beginSessionInBackgroundWithClient:entityNames:target:selector:` method allows you to specify a target and selector to be invoked when all clients have joined the session. Use this method if you do not want to block the client while waiting for other clients.

Negotiating

All clients need to negotiate a sync mode before pushing and pulling records. The sync modes are slow sync, fast sync, refresh sync, pull the truth, and push the truth.

The first time a client syncs, it slow syncs—that is, it pushes all its records. Otherwise, the default mode (fast syncing) is used to push and pull changes only. However, certain events can prevent a fast sync. When this happens, the client might request a different sync mode, or the sync engine might force a mode. If multiple clients sync simultaneously, then all the client requests need to be considered. For example, if one client wants to push the truth, all other clients need to pull the truth.

Slow Syncing

Use the `clientWantsToPushAllRecordsForEntityNames:` method to request a slow sync. For example, if your client doesn't know what records changed since the last sync, it should slow sync. This method informs the sync engine that the client will push all its records:

```
// Request a slow sync
[session clientWantsToPushAllRecordsForEntityNames:entityNames];
```

If a client is slow syncing, the client may pull deletes for records that the engine knew were on the client during the last sync.

Refresh Syncing

The `clientDidResetEntityNames:` method informs the sync engine that the client was reset and wants to refresh sync. However, if a client is refresh syncing, the engine doesn't use the client snapshot to compare pushed records. Consequently, no delete changes are pulled.

Pull the Truth

If you want all client records replaced by the truth database records, send `setShouldReplaceClientRecords:forEntityNames:` to your `ISyncClient` object. This is how you request a pull the truth sync mode:

```
// Requests that client records be replaced by truth
[client setShouldReplaceClientRecords:YES
                        forEntityNames:entityNames];
```

Mixing Sync Modes

Typically, the sync modes for all the entities synced by a client are the same. However, in some cases the sync modes may be different. The most common case is when a client doesn't sync the same entities in every sync session or explicitly requests a different sync mode for one entity and not another. Because of such differences, the sync engine might allow a fast sync of one entity and force a slow sync of another entity. This is permissible except when relationships exist between these entities.

To avoid relationship inconsistencies, clients should always request the same sync mode for entities that have relationships between them. In other words, clients should use the same sync mode for records belonging to the same object graph; otherwise, inconsistencies between the records may result—especially if inverse relationships are changed.

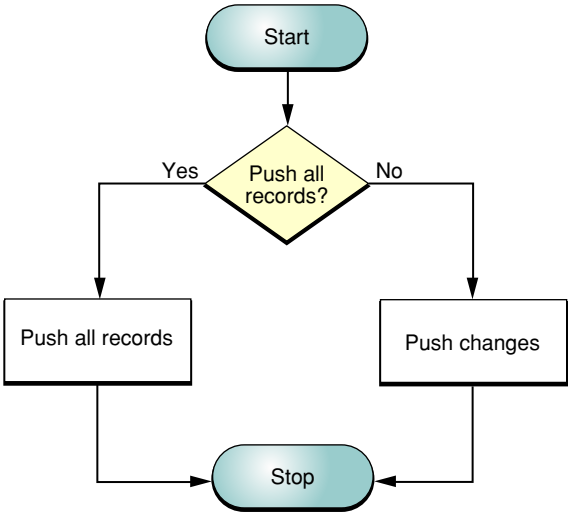
Warning: If the sync modes of entities that have relationships between them are different, then the client must request a refresh sync for those entities using the `clientDidResetEntityNames:` method. Otherwise, data may be corrupted.

Pushing

Assuming you negotiated a sync mode, you can now push your changes to the sync engine. Pushing changes is an optional step—it can be skipped if the client has no changes or is not ready to push them. However, if you decide to push changes, you must do so before pulling changes.

Figure 3 (page 30) shows the flow of a typical pushing method. You first ask the sync engine if you should push any changes for the specified entities. Then you ask if you should push all the records or just the changes. You do this because the sync engine might select a mode other than the one you requested.

Figure 3 Typical pushing logic



When slow syncing, a client pushes all its records to the sync engine and lets it determine which properties of those records changed. The sync engine does this by comparing each pushed record to a snapshot of the record taken at the end of the previous sync.

When fast syncing, a client may push only the records that changed. Similarly, the sync engine determines what individual properties of those records changed by comparing them to the snapshot.

Alternatively, a client can save time and effort by pushing just the changes to individual properties of records (assuming that the client keeps track of every property change to every record).

The sync session transitions to the pushing state the first time any of the methods discussed below are invoked. All the methods can be used only during this state, except `clientLostRecordWithIdentifier:shouldReplaceOnNextSync:`, which can also be invoked during the pulling state. An implicit transaction begins when entering the pushing state and ends when leaving this state.

Although the methods for requesting a sync mode and pushing and pulling records are entity-based, entities in the same data class usually have the same sync mode. Entities can have relationships to other entities in the same data class, but not relationships to entities in other data classes. Therefore, you should not request different sync modes for entities in the same data class, and enable or disable entities in the same data class after the application first syncs.

Push Changes?

To determine whether any records belonging to an entity should be pushed, use the `shouldPushChangesForEntityName:` method, as follows:

```
if ([session shouldPushChangesForEntityName:entityName]){
    // Push records for entityName
    ...
}
```

Push All Records?

To determine whether you should slow or fast sync, use the `shouldPushAllRecordsForEntityName:` method. If this method returns `YES`, you should push all records. If you do not push all the records, the sync engine assumes you deleted the records you did not push and deletes them from the truth database. If the `shouldPushAllRecordsForEntityName:` method returns `NO`, push just the changes since the last sync.

```
if ([session shouldPushAllRecordsForEntityName:entityName]) {
    // Slow sync entityName
    ...
} else {
    // Fast sync entityName
    ...
}
```

Pushing Records

You can push all records or just the changed records by using the `pushChangesFromRecord:withIdentifier:` method, as follows:

```
[session pushChangesFromRecord:record withIdentifier:id];
```

Typically, you transform your entity objects into a dictionary representation before pushing them. The `record` argument above is a dictionary that contains only the supported properties you want to sync. It should not contain any local properties used exclusively by your client.

For example, the dictionary representation of a Media object pushed by the MediaAssets sample application might look like this:

```
MediaAssets[789] pushing sync record={
    "com.mycompany.syncservices.RecordEntityName" =
"com.mycompany.syncexamples.Media";
    date = 2004-05-22 00:00:00 -0700;
    event = ("3571D5D0-C0A5-11D8-A57D-000A95BF2062");
    imageURL = "file://2004/05/22/IMG_1106.JPG";
    title = "IMG_1106.JPG";
}
```

The sync engine uses record identifiers to locate records. Record identifiers are unique across all sync records, not just within the scope of an entity. Therefore, the `record` dictionary also needs to contain a value for the `ISyncRecordEntityNameKey` key that identifies the record's entity name.

The client is expected to create the unique record identifiers, typically a UUID, when pushing new records. Conversely, clients can either accept the record identifiers assigned by the sync engine when pulling new records or reset it to something else (see "Pulling" (page 33)). For these reasons, your client needs to maintain the association between a record identifier and your entity object in order to communicate future changes to the sync engine.

How you generate record identifiers is dependent on your application but you should follow this rule:

> A local identifier for a given record must not change or collide with other identifiers for the life of the record—it can only be reused after the record is deleted and expunged from the truth database.

Pushing Changes to Properties

If you know what properties of a record changed, you can just push the properties that changed by creating an `ISyncChange` object and using the `pushChange:` method. You can improve efficiency by pushing just the changes.

Pushing Delete Changes

When fast syncing, you don't have to create an `ISyncChange` object for deleted records. You can use the `deleteRecordWithIdentifier:` method to push deleted record changes as follows:

```
[session deleteRecordWithIdentifier:recordID];
```

Handling Lost Records

If you lost a record and want to replace it or ignore it in a future sync, use the `clientLostRecordWithIdentifier:shouldReplaceOnNextSync:` method.

Mingling

The sync session enters the mingling state after all clients participating in the sync have pushed their changes. During mingling, the engine takes all the changes from all the participating clients, checks them for conflicts, and applies them to the truth database. At the end of the mingling state, all changes have been incorporated from multiple clients.

If there is a conflict, the engine first tries to resolve it using a set of rules specific to the entity in question—for example, the engine uses the identity properties of an entity to determine if two records are the same record.

The client informs the sync engine that it is ready to begin pulling changes by invoking one of the `prepareToPullChanges...` methods. Invoking them at any other time will raise an exception.

Using the `prepareToPullChangesForEntityNames:beforeDate:` method, you specify how long the client is willing to wait for the mingling state to complete as in:

```
if ([session prepareToPullChangesForEntityNames:entityNames beforeDate:[NSDate
 distantFuture]]) {
    // Pull changes for each entity
    ...
```

The mingling state ends when this method returns. If this method returns `NO`, you should skip pulling changes.

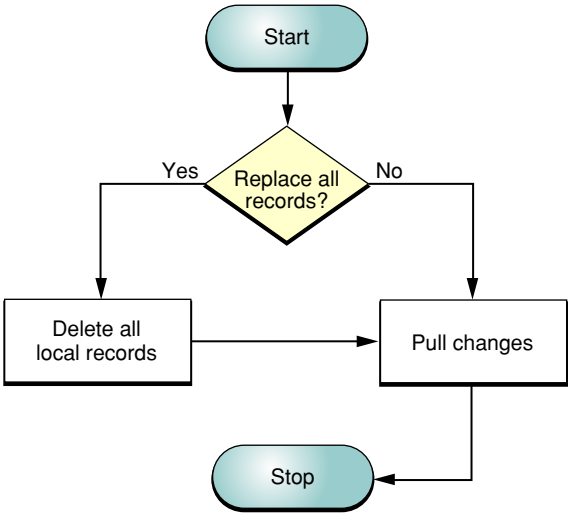
Using the `prepareToPullChangesInBackgroundForEntityNames:target:selector:` method, you specify a target and action to be invoked when the mingling state is complete. This method is nonblocking. The mingling state ends when this method invokes the action.

Note that the mingler considers the changes made between syncs to be transitive. The mingler optimizes a set of consecutive changes to a record by discarding all changes to a property except the last. Consequently, clients should not assume that changes are pulled in the same sequence in which they are pushed—only the final value in a sequence of changes is relevant.

Pulling

Pulling changes from the sync engine is the final state in the sync session. The sync session transitions to this state at the end of the mingling state. Typically, you iterate through the entity names passed to one of the `prepareToPullChanges...` methods, and for each entity, you pull and apply the changes. But first you need to see whether you should try to pull changes. Figure 4 shows the logic of a typical pulling implementation. Details of pulling changes are covered in "Pulling Changes" (page 34).

Figure 4 Typical pulling logic



Pull Changes?

You should first check with the sync engine, using the `shouldPullChangesForEntityName:` method, to determine whether you should try to pull changes for an entity before pulling changes. It is useful to create a filtered list of entities from the supported entities, as follows:

```
NSEnumerator *entityEnumerator = [entityNames objectEnumerator];
NSMutableArray *filteredEntityNames = [NSMutableArray array];
while (entityName = [entityEnumerator nextObject]){
    if ([session shouldPullChangesForEntityName:entityName])
        [filteredEntityNames addObject:entityName];
}
```

Replace All Records?

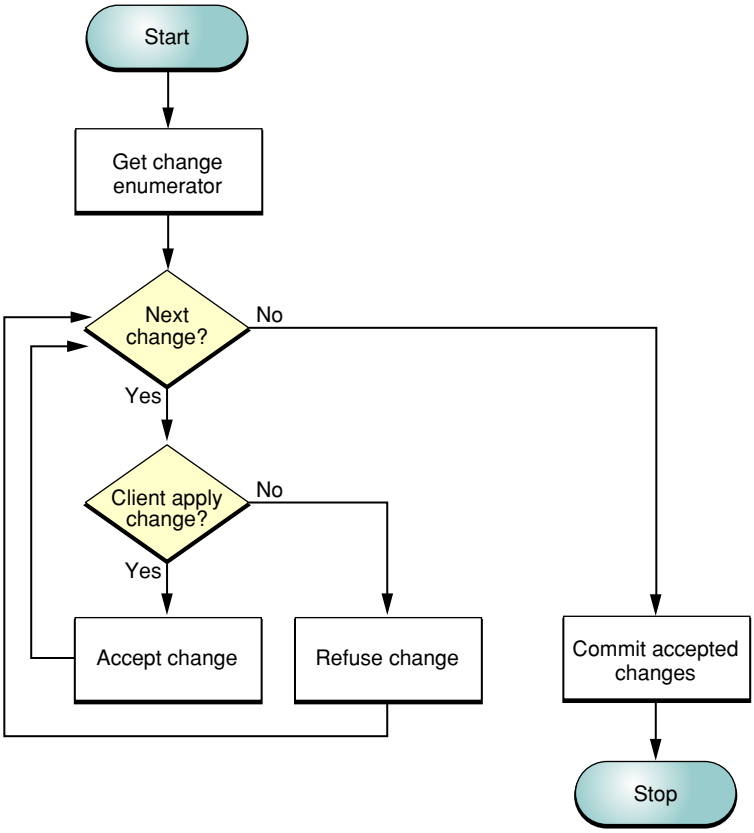
For each entity, you need to check with the sync engine, using the `shouldReplaceAllRecordsOnClientForEntityName:` method, to determine whether you should replace all its records. For example, you need to replace all the records if the sync mode is pull the truth. If this method returns `YES`, delete your copies of the records before pulling changes, as in this example:

```
if ([session shouldReplaceAllRecordsOnClientForEntityName:entityName]) {
    // Remove the records for this entity from the client data source
    ...
}
```

Pulling Changes

Now that you have a collection of entity names to pull changes for, you are ready to pull and apply the changes. Figure 5 shows the flow of a typical pulling implementation.

Figure 5 Typical pulling-changes logic



Getting the Changes

You use the `changeEnumeratorForEntityNames:` method to get the changes from the sync engine for the specified entity names. Because this method returns an object enumerator, you need to iterate through the changes and apply them one by one. Figure 5 (page 35) shows how you might process each change.

Accepting and Refusing Changes

For each change, you need to decide whether you are going to apply it to your local data source. If you apply it, you need to invoke `clientAcceptedChangesForRecordWithIdentifier:formattedRecord:newRecordIdentifier:` to inform the sync engine.

Otherwise, you invoke `clientRefusedChangesForRecordWithIdentifier:` to reject a change. This method applies only to add and modify changes, not deletes. On subsequent fast syncs, the rejected changes do not appear in the change enumerator returned by the `changeEnumeratorForEntityNames:` method. However, they do appear in the change enumerator during the next slow sync.

When you invoke the `clientAcceptedChangesForRecordWithIdentifier:formattedRecord:newRecordIdentifier:` method, you have the opportunity to change the record identifier supplied by the sync engine to a record identifier supplied by the client. The identifiers for new records are chosen by the sync engine. (When refresh

syncing, every record not pushed by the client is considered new.) Optionally, you can change all the record identifiers for new records at the end of applying changes using the `clientChangedRecordIdentifiers:` method.

How you generate record identifiers is dependent on your application but you should follow this rule:

> A local identifier for a given record must not change or collide with other identifiers for the life of the record—it can only be reused after the record is deleted and expunged from the truth database.

Handling Lost Records

Lost records are handled in a way that is similar to the pushing state. To inform the sync engine about a record that the client lost and may want to replace, use the `clientLostRecordWithIdentifier:shouldReplaceOnNextSync:` method.

Committing Changes

When you are done processing all the changes, you invoke `clientCommittedAcceptedChanges` to commit the accepted changes to the sync engine. This method closes the transaction that was opened by the first invocation of one of the following methods:

```
clientAcceptedChangesForRecordWithIdentifier:formattedRecord:newRecordIdentifier:
clientRefusedChangesForRecordWithIdentifier:
clientChangedRecordIdentifiers:
clientLostRecordWithIdentifier:shouldReplaceOnNextSync:
```

The pulling state supports multiple transactions so that clients can batch changes and apply them more efficiently. A new transaction begins by invoking one of the above methods.

Finishing

Sending `finishSyncing` to an `ISyncSession` object closes the last open transaction and terminates the session. The `ISyncSession` object cannot be reused after entering this state. This method can be invoked at any time during an active sync session.

If you invoke `finishSyncing` during the negotiating state, the sync engine assumes that the client has no records to push.

If you invoke `finishSyncing` during either the pushing or pulling states, the last transaction is closed and any changes in that transaction are applied by the sync engine. For example, if you are in the pulling state, then any changes you accepted using the `clientAcceptedChanges...` method are applied even if you did not invoke `clientCommittedAcceptedChanges` before finishing the session.

If you invoke `finishSyncing` during the mingling state (for example, while waiting for the target method specified in `prepareToPullChangesInBackgroundForEntityNames:target:selector:` to be invoked), the session is terminated and all changes made during the previous pushing state are applied.

An `ISyncSession` object cannot be reused, so release it after invoking `finishSyncing`.

Canceling

You can invoke the `cancelSyncing` method at any time to terminate a sync session. However, when canceling a sync session, the sync engine rolls back to the state of the last closed transaction. It does not roll back all the transactions made within the session. For this reason, you need to know when transactions begin and end in the various states.

If you invoke `cancelSyncing` during the negotiating state, then any negotiation methods invoked during this state are ignored. For example, if you request a refresh sync, then you will have to request it again the next time you sync.

If you invoke `cancelSyncing` during the pushing state, all pushed changes, deletions, and lost records are discarded, and the state is rolled back to the end of the negotiation state. If a client is fast syncing, it must remember which records changed so that it can push those changes again during the next sync. If the client cannot do so, it should slow sync the next time.

If you invoke `cancelSyncing` during the mingling state, the session is terminated and all changes made during the previous pushing state are applied.

If you invoke `cancelSyncing` during the pulling state, all changes that were not committed using `clientCommittedAcceptedChanges` are pulled again during the next sync.

An `ISyncSession` object cannot be reused, so release it after invoking `cancelSyncing`.

Creating a Sync Schema

A sync schema consists of a property list, image files, and localization files contained in a bundle that you specify when registering a schema. If you are using Core Data, then it might contain one or more managed object models, too. You can create a schema to extend an existing schema or to define your own schema. The schema property list contains a description of the entities and properties used by the client as well as other attributes used by the sync engine.

See Cocoa Design Patterns in *Cocoa Fundamentals Guide* for a description of entity-relationship models and terminology. See "Registering Schemas" (page 55) for how to register your own schemas.

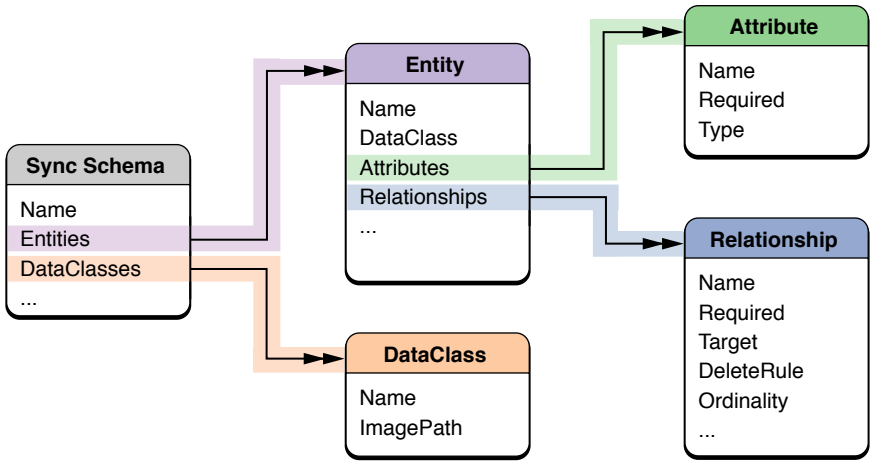
Note: Core Data syncing and the managed object models schema property are available in Mac OS X v10.5 and later.

Sync Schema Components

A sync schema is a property list stored in a `.syncschema` bundle along with other related files such as localization and image files. A property list is made up of primitive types such as arrays, dictionaries, and string values, not arbitrary types. Nevertheless, it's helpful to view a property list as a pseudo object model.Figure 1 (page 39) illustrates the components in a sync schema: the schema itself, entities, data classes, attributes, and relationships. The double arrowheads in the diagram imply a to-many relationship (implemented as an array in the property list) between components.

Note: All line endings in a string attribute should use UNIX-style line endings, not DOS-style endings.

Figure 1 Sync schema components



Creating a Sync Schema Bundle

You can create a schema bundle using Xcode. Just select New Project from the File menu and select the Sync Schema template from the Assistant. Then edit the `Schema.plist` file to define your entities and properties as described in "Creating a Sync Schema Property List" (page 40). You can also localize the names of entities and properties using the `Schema.strings` file as described in "Localizing Property Names" (page 50).

Creating a Sync Schema Property List

After you create a schema bundle, you edit the `Schema.plist` file to define your entities and properties. Unless otherwise stated, the following high-level sync schema properties are optional:

Property	Description
`BaseName`	A string that is the name of the schema that this schema is extending, if applicable. Available in Mac OS X v10.6 and later.
`BaseMajorVersion`	An integer that is the major version number of the schema that this schema is extending, if applicable. For example, if the version number is `10.4`, the major version is `10` and the minor version is `4`. Available in Mac OS X v10.6 and later.
`BaseMinorVersion`	An integer that is the minor version number of the schema that this schema is extending, if applicable. For example, if the version number is `10.4`, the major version is `10` and the minor version is `4`. Available in Mac OS X v10.6 and later.
`Comment`	A string used to document the schema. Available in Mac OS X v10.4 and later.
`DataClasses`	An array containing one or more dictionaries that describe a data class. The properties of a data class are described in "Data Class Properties" (page 42). If this is not a schema extension, this key is required. Available in Mac OS X v10.4 and later.
`Entities`	An array of dictionaries describing new record types and record type extensions. The entity properties are described in "Entity Properties" (page 42). Available in Mac OS X v10.4 and later.
`MajorVersion`	An integer that is the major version of this schema, if applicable. For example, if the version number is `10.4`, the major version is `10` and the minor version is `4`. Available in Mac OS X v10.6 and later.

Property	Description
`ManagedObjectModels`	An array of strings containing the paths to the Core Data managed object model files (files with a `.mom` suffix) associated with this schema. Relative paths are relative to the location of the schema bundle. Use a relative path to load a model that lives outside the schema bundle—for example, use `../../../MyModel.mom` to load a model located in the application's `Resources` directory. Use this property if you want to sync Core Data managed objects. By default, all entities and properties defined in a managed object model are registered with the sync manager. Read "Syncing Core Data Applications" (page 85) for how to designate a subset of entities and properties to sync and set other required sync information in a managed object model. Available in Mac OS X v10.5 and later.
`MinorVersion`	An integer that is the minor version of this schema, if applicable. For example, if the version number is `10.4`, the major version is `10` and the minor version is `4`. Available in Mac OS X v10.6 and later.
`Name`	A string providing a unique name for this schema. Typically, a reverse DNS-style name such as `com.mycompany.SyncExamples`. You should never change the schema name. Doing so orphans the old schema definition, which you then must remove. This key is required. Available in Mac OS X v10.4 and later.
`StrictParsing`	Set to `true` if you want to use strict validation, `false` otherwise. If not specified, uses the validation corresponding to the binary registering the schema. If the binary was built on Mac OS X v10.5 and later, uses strict validation. Available in Mac OS X v10.5 and later.
`UIHelperClass`	Set to the class that implements the `ISyncUIHelper` informal protocol. Use this key if you want to customize the presentation of your records and schema in the user interface. When the sync schema bundle is loaded, an instance of this class is created to handle presentation requests from the data change alert and conflict resolver user interfaces. Therefore, if this key is set, the class must be included in this schema bundle. Available in Mac OS X v10.5 and later.

For example, the following property list fragment specifies the name of the sync schema as `com.mycompany.SyncExamples`. Typically, you use reverse DNS-style names for schemas, entities, and client identifiers.

```
<plist version="1.0">
<dict>
    <key>Name</key>
    <string>com.mycompany.SyncExamples</string>
    <key>DataClasses</key>
    <array>
     ...
```

```
    </array>
    <key>Entities</key>
    <array>
     ...
    </array>
</dict>
</plist>
```

Data Class Properties

A data class is a logical grouping of related entities. Typically, it is a grouping that makes sense to a user—for example, a group of entities that a user might want to sync over MobileMe such as contacts. You can add an entity to a data class by setting the entity's `DataClass` attribute. Any data class you create should be added to the schema's `DataClasses` property.

The properties of a data class are as follows:

Property	Description
`ImagePath`	Path to an image representing this data class. If it is a relative path, then it is assumed to be relative to the schema bundle. Typically, the image is an icon file that can be displayed at different resolutions. Otherwise, it should be at least 32 x 32 pixels. This key is required. Available in Mac OS X v10.4 and later.
`Name`	A unique name for the data class. Typically, a reverse DNS-style name such as `com.apple.Contacts`. Can be localized using the `Schema.strings` file located in the schema bundle. This key is required. Available in Mac OS X v10.4 and later.

For example, the following property list fragment specifies the data classes for an example application:

```
    <key>DataClasses</key>
    <array>
        <dict>
            <key>Name</key>
            <string>com.mycompany.SyncExamples</string>
            <key>ImagePath</key>
            <string>SyncExamples.icns</string>
        </dict>
    </array>
```

Entity Properties

The properties of an entity within a sync schema are described below and are optional unless explicitly stated otherwise:

Property	Description
`Attributes`	An array containing dictionaries describing the attributes in the entity or extension. Attribute properties are described in "Attribute Properties" (page 44). Available in Mac OS X v10.4 and later.
`CompoundIdentity-Properties`	An array of arrays of property names used to match a new record from a sync client with an existing record in the truth database. This key is similar to the `IdentityProperties` key except that it supports more flexible matches. If both this key and the `IdentityProperties` key are set, then this property key is used to match records instead. When this key is used to match records, the properties in each array are used as the identity properties in the order that the array appears. If none of the properties in an array are pushed by a client, then the properties in the next array are used as the identity properties. A property should not be in more than one array of a `CompoundIdentityProperties` array. For example, the contacts schema sets the `CompoundIdentityProperties` key to two arrays. The first array contains the `first name`, `middle name`, and `last name` property keys. The second array contains the `company name` property key. This allows a contact pushed without a company name to match a record that was previously pushed with the same first and last name but with a company name set. It also allows for company-only matches. Available in Mac OS X v10.6 and later.
`DataClass`	The name of the data class this entity belongs to. This key is required. Available in Mac OS X v10.4 and later.
`ExcludeFromData-ChangeAlert`	Set to `true` if you want to exclude changes to records of this entity type from the total count of changed records used in alert messages, `false` otherwise. The default value is `false`. Available in Mac OS X v10.4 and later.
`ExtensionName`	A unique name for this extension, if you are adding attributes to an existing entity. The extension name is used to scope the attribute and relationship names to prevent collisions. Typically, a reverse DNS-style name such as `com.apple.iCal.CalendarExtensions`. If this is a schema extension, this key is required. Available in Mac OS X v10.4 and later.
`IdentityProperties`	An array containing the names of the entity properties that are used to match a new record from a sync client with an existing record in the truth database. If the target of a to-one relationship is used, the name of the relationship is specified as the property. A client record matches a truth record if all the identity property values in the client record are equal to the values in the truth record. Properties with `null` values (values that are not set) also need to match. Available in Mac OS X v10.4 and later.

Property	Description
`Name`	A unique name for the entity. Typically, a reverse DNS-style name such as `com.apple.Contacts`. Can be localized using the `Schema.strings` file located in the schema bundle. This key is required. Available in Mac OS X v10.4 and later.
`Parent`	The name of the relationship that contains or encloses records belonging to this entity. Used to display a user friendly entity name when conflicts occur during syncing. For example, in the contacts schema, the target of the `contact` relationship is the parent of a Phone Number record. Available in Mac OS X v10.4 and later.
`PropertyDependencies`	An array containing arrays of groups of properties that are dependent on each other. A dependent property is one which must be pushed to a client if any of its dependencies are changed. Each entry in this array is itself an array of strings, specifying the names of the codependent properties. Entries in these arrays may be an attribute name or a relationship name. Available in Mac OS X v10.4 and later.
`Relationships`	An array containing dictionaries describing the relationships in the entity or extension. Relationship properties are described in "Relationship Properties" (page 47). Available in Mac OS X v10.4 and later.

For example, the following property list fragment describes the properties of the `com.mycompany.syncexamples.Media` entity. Sample values for the `Attributes` and `Relationships` properties appear in "Attribute Properties" (page 44) and "Relationship Properties" (page 47).

```
<dict>
    <key>Name</key>
    <string>com.mycompany.syncexamples.Media</string>
    <key>DataClass</key>
    <string>com.mycompany.SyncExamples</string>
    <key>Attributes</key>
    ...
    <key>Relationships</key>
    ...
    <key>IdentityProperties</key>
    ...
</dict>
```

Attribute Properties

The properties of an attribute belonging to an entity are as follows:

Property	Description
`AutomaticConflict-ResolutionPolicy`	A dictionary containing tips on how to automatically resolve conflicts to values of this attribute without generating change alert messages. See Table 1 (page 45) for a description of the optional keys you can use in this dictionary. Available in Mac OS X v10.6 and later.
`ExcludeFromData-ChangeAlert`	Set to `true` if you want to exclude changes to this attribute from the total count of changed attributes used in alert messages, `false` otherwise. Available in Mac OS X v10.4 and later.
`Name`	The name of the attribute. Can be localized using the `Schema.strings` file located in the schema bundle. The localization key is `$entity/$attribute_name`. For example, the localization key for the `title` attribute of the `com.mycompany.syncexamples.Media` entity would be `com.mycompany.syncexamples.Media/title`. This key is required. Available in Mac OS X v10.4 and later.
`Required`	Set to `true` if required, `false` otherwise. The sync engine will reject a record that doesn't have a required attribute set. The default value is `false`. Available in Mac OS X v10.4 and later.
`Type`	The type of the attribute. See Table 2 (page 46) for possible values. This key is required. Available in Mac OS X v10.4 and later.

Use the keys in Table 1 (page 45) in the `AutomaticConflictResolutionPolicy` dictionary to automatically resolve conflicts to attribute values.

Table 1 `AutomaticConflictResolutionPolicy` Keys

Key	Description
`PreferredClientTypes`	An array of client type strings specifying the preferred order of clients for resolving conflicts. The client types are described in Table 1 (page 58). For example, if `app` is the first item in the `PreferredClientTypes` array and the client is an application, then its record is selected instead of a record from a client of type `device` or `server`. Available in Mac OS X v10.6 and later.
`PreferredRecord`	A string specifying which record should be selected when there is a conflict. This string must be one of the following predefined values: `Truth`, `Client`, or `LastModified`. If the value for this key is `Truth`, the record in the truth database is selected. If the value for this key is `Client`, the client's record is selected. If the value for this key is `LastModified`, the record with the most recent changes is selected. Typically, you set this key to `LastModified`, in which case, you do not need to set the `PreferredClientTypes` key. Available in Mac OS X v10.6 and later.

Sync Services supports the types of attributes shown in Table 2 (page 46). Note that the types are case-sensitive and expected to appear in lowercase. The corresponding instance must be a member of the Objective-C class.

Table 2 Attribute types

Attribute type	Objective-C class	Description
`array`	NSArray	Available in Mac OS X v10.4 and later.
`boolean`	NSNumber	Available in Mac OS X v10.4 and later.
`calendar date`	NSCalendarDate	Available in Mac OS X v10.4 and later.
`color`	NSColor	Available in Mac OS X v10.4 and later.
`data`	NSData	Available in Mac OS X v10.4 and later.
`date`	NSDate	Available in Mac OS X v10.4 and later.
`dictionary`	NSDictionary	Available in Mac OS X v10.4 and later.
`enum`	NSString	Available in Mac OS X v10.4 and later.
`number`	NSNumber	Available in Mac OS X v10.4 and later.
`set`	NSSet	Available in Mac OS X v10.6 and later.
`string`	NSString	Available in Mac OS X v10.4 and later.
`url`	NSURL	Available in Mac OS X v10.4 and later.

For example, the following property list fragment describes the attributes of the `com.mycompany.syncexamples.Media` entity. Each dictionary below contains the name and type of each attribute. For example, the `date` attribute is specified as a `calendar date` type. Therefore, when pushing or pulling a `com.mycompany.syncexamples.Media` record, the value for the `date` key will be an instance of NSCalendarDate.

```
<key>Attributes</key>
<array>
    <dict>
        <key>Name</key>
        <string>date</string>
        <key>Type</key>
        <string>calendar date</string>
    </dict>
    <dict>
        <key>Name</key>
        <string>imageURL</string>
        <key>Type</key>
        <string>url</string>
    </dict>
    <dict>
        <key>Name</key>
        <string>title</string>
        <key>Type</key>
```

```
        <string>string</string>
    </dict>
</array>
```

If you use an `array` type, you can further limit the contents of the array to instances of a class and its subclasses using the `Subtype` key. For example, the following property list fragment describes a `dates` attribute as an array containing NSCalendarDate objects. The `Subtype` key is optional but if used, is enforced by the sync engine.

```
<key>Attributes</key>
<array>
    <dict>
        <key>Name</key>
        <string>dates</string>
        <key>Type</key>
        <string>array</string>
        <key>Subtype</key>
        <string>calendar date</string>
    </dict>
</array>
```

If you use an `enum` type, you must also specify the possible `enum` values using the `EnumValues` key. In this example, the `display order` attribute is an `enum` that can be either `firstNameFirst` or `lastNameFirst`:

```
<dict>
    <key>EnumValues</key>
    <array>
    <string>firstNameFirst</string>
    <string>lastNameFirst</string>
    </array>
    <key>Name</key>
    <string>display order</string>
    <key>Type</key>
    <string>enum</string>
</dict>
```

Relationship Properties

Relationships from one entity to another are allowed only if the entities belong to the same data class. If you attempt to define a relationship to an entity in another data class, validation fails at runtime and an exception is raised. Schema validation may fail as well.

The properties of a relationship belonging to an entity are:

Property	Description
`ExcludeFromData-ChangeAlert`	Set to `true` if you want to exclude changes to this relationship from the total count of changed relationships used in alert messages, `false` otherwise. Available in Mac OS X v10.4 and later.

Property	Description
`DeleteRule`	A value of `cascade` or `nullify`. If set to `cascade`, then when the source record of the relationship is deleted, all the target records are deleted. If set to `nullify`, then when the target record is deleted, the reference from the source's relationship is removed. If it is a to-one relationship, the relationship is set to `null`. If it is a to-many relationship, then it is removed from that relationship. Available in Mac OS X v10.4 and later.
`InverseRelationships`	An array of dictionaries specifying this relationship's inverse relationships. The keys in an inverse relationship dictionary are described in "Inverse Relationship Properties" (page 49). Available in Mac OS X v10.4 and later.
`Name`	The name of the relationship. Can be localized using the `Schema.strings` file located in the schema bundle. The localization key is `$entity/$relationship_name`. For example, the localization key for the `event` relationship of the `com.mycompany.syncexamples.Media` entity would be `com.mycompany.syncexamples.Media/event`. This key is required. Available in Mac OS X v10.4 and later.
`Ordinality`	Indicates a to-one or to-many relationship, value of `one` or `many`. The default value is `one`. Available in Mac OS X v10.4 and later.
`Ordering`	Value of `none`, `weak` or `strong`. If the value is `none`, then no order is maintained. If the value is `strong`, then the order is maintained and alert panels appear when conflicts occur. If the value is `weak`, then the order is maintained but alert panels do not appear when conflicts occur—the sync engine merges ordering conflicts. The default value is `none`. Available in Mac OS X v10.4 and later.
`Required`	Set to `true` if required, `false` otherwise. The sync engine will reject a record that doesn't have a required relationship set. The default value is `false`. Available in Mac OS X v10.4 and later.
`Target`	An array containing the names of the target entities belonging to the same data class. Generally, a relationship has a single target. However, you can specify multiple entities as the target if the target can be a number of different types. This key is required. Available in Mac OS X v10.4 and later.

For example, the following property list fragment describes the relationships of the `com.mycompany.syncexamples.Media` entity. The fragment defines a to-one relationship, called `event`, from `com.mycompany.syncexamples.Media` to `com.mycompany.syncexamples.Event`.

```
<key>Relationships</key>
<array>
    <dict>
```

```
        <key>Name</key>
        <string>event</string>
        <key>Ordinality</key>
        <string>one</string>
        <key>Target</key>
        <array>
            <string>com.mycompany.syncexamples.Event</string>
        </array>
    </dict>
</array>
```

Inverse Relationship Properties

The properties of an inverse relationship belonging to a relationship are as follows:

Property	Description
`EntityName`	The name of the entity belonging to the same data class. This key is required. Available in Mac OS X v10.4 and later.
`RelationshipName`	The name of the entity that has the inverse relationship. This key is required. Available in Mac OS X v10.4 and later.

For example, the following property list fragment describes the inverse relationship between the `event` relationship of the `com.mycompany.syncexamples.Media` entity and the `media` relationship of the `com.mycompany.syncexamples.Event` entity. The `media` relationship is a to-many relationship from `com.mycompany.syncexamples.Event` to `com.mycompany.syncexamples.Media`.

```
<key>InverseRelationships</key>
<array>
    <dict>
        <key>EntityName</key>
        <string>com.mycompany.syncexamples.Event</string>
        <key>RelationshipName</key>
        <string>media</string>
    </dict>
</array>
```

Identity Properties

The value of the `IdentityProperties` key is an array containing the names of the properties that are used to identify a record of this entity type if there is a conflict. For example, the identity properties for a `com.mycompany.syncexamples.Media` entity are specified as the `date` and `title` properties below. The `com.mycompany.syncexamples.Media` entity may have other properties, such as `imageURL`, which are not in this array.

```
<key>IdentityProperties</key>
<array>
    <string>date</string>
    <string>title</string>
</array>
```

Warning: You should always push the identity properties for a record. Otherwise, you might create duplicate records during the first sync or a refresh sync.

Property Dependencies

In some cases, if one property changes, you might want to push another related property, even if its value didn't change. For example, suppose you change a calendar event to an all day event in iCal and using your phone you change the start time of the same event. Even though each client changed a different property, the changes conflict and need to be resolved. You can inform the sync engine about dependent properties within your entities so that these types of conflicts are recognized when syncing.

The value of the `PropertyDependencies` key is an array of arrays containing the dependent properties. Each array contains the set of properties that are dependent on each other. For example, you might declare that the `startDate`, `endDate` and `allDayEvent` properties are dependent as follows:

```
<key>PropertyDependencies</key>
<array>
    <array>
        <string>startDate</string>
        <string>endDate</string>
        <string>allDayEvent</string>
    </array>
</array>
```

Localizing Property Names

You can define user-friendly names for data classes, entities, and properties by editing the `Schema.strings` schema bundle localization file. Localized names are used by Sync Services applications that present records to users.

For example, an alert message might appear using a data class name in a message—for example, a panel might appear asking the user if it's okay to syncronize with other applications. If no localization is provided, the DNS-style data class name is used in the alert message which might be confusing to the user. You should modify your `Schema.strings` file to provide both plural and singular forms of data class names.

Specifically, you can define a localized string for the DataClass, Entity, Relationship, and Attribute components illustrated in Figure 1 (page 39). The key for the DataClass and Entity localized string is simply the Name attribute of the DataClass component. The key for an Attribute or Relationship component is of the form:

```
$entity/$property
```

For example, the `Schema.strings` file for the MediaExample application is:

```
/* Data Class: The data class for the Sync Services examples.*/
"com.mycompany.SyncExamples" = "Media Example Data";

/* Singular Data Class Name: Used for any UI that needs a singular name.*/
"Singular:com.mycompany.SyncExamples" = "Media Example Data";

/* Entity: A media item such as a photo taken at an event. */
"com.mycompany.syncexamples.Media" = "Media";
```

```
/* Attribute: The date a photo was taken. */
"com.mycompany.syncexamples.Media/date" = "Creation Date";

/* Attribute: The file URL for the image. */
"com.mycompany.syncexamples.Media/imageURL" = "Image URL";

/* Attribute: A user-friendly title of the photo. */
"com.mycompany.syncexamples.Media/title" = "Image Title";

/* Relationship: The event where the photo was taken. */
"com.mycompany.syncexamples.Media/event" = "Event";
...
```

Creating a Schema Extension

Sync Services also allows you to extend an existing schema without changing it directly—for example, when you don't have permission to change it directly or you want to encapsulate your extensions from other applications. Sync Services ensures that only those applications that use the extension see the entities and properties defined by them. You can even extend existing Apple Applications schemas without adversely affecting other desktop applications. Read *Apple Applications Schema Reference* for details on these schemas.

Schema extensions reside in a bundle structure that is identical to sync schema bundles. The only difference is that the `Schema.plist` and `Schema.strings` files pertain only to entity and property extensions. Use unique reverse DNS-style names for the entities and properties you add to an existing schema.

For example, suppose you want to add an entity called Calendar to the MediaExample schema, which maintains a collection of Event records. That way, when you import an iCal file, you can group the events together. You can do this by creating a Calendar entity with an identity attribute, such as a name or title, and an inverse to-many relationship to Event.

Follow these steps to create a schema extension that adds a Calendar entity to `SyncExamples.syncschema` where the existing schema uses `com.mycompany.synceexamples` and the extension uses `com.anothercompany` as the prefix for names:

1. Define a unique name for the schema extension in `Schema.plist`:

```
    <key>Name</key>
    <string>SyncExamplesExtension</string>
```

2. Add new properties to existing entities in `Schema.plist` using a unique reverse DNS-style name for each. Specifically, add an inverse to-one relationship from Event to Calendar:

```
<dict>
    <key>ExtensionName</key>
    <string>com.anothercompany.EventExtensions</string>
    <key>Name</key>
    <string>com.mycompany.syncexamples.Event</string>
    <key>Relationships</key>
    <array>
        <dict>
            <key>InverseRelationships</key>
            <array>
                <dict>
```

```
                    <key>EntityName</key>
                    <string>com.anothercompany.Calendar</string>
                    <key>RelationshipName</key>
                    <string>events</string>
                </dict>
            </array>
            <key>Name</key>
            <string>com.anothercompany.calendar</string>
            <key>Ordinality</key>
            <string>one</string>
            <key>Target</key>
            <array>
                <string>com.anothercompany.Calendar</string>
            </array>
        </dict>
    </array>
</dict>
```

3. Add new entity definitions to `Schema.plist`. Specifically, add a Calendar entity and to-many inverse relationship from Calendar to Event. You can use the same data class, `com.mycompany.SyncExamples`, defined in `SyncExamples.syncschema` but use unique DNS-style names for the added class:

```
<dict>
    <key>Attributes</key>
    <array>
        <dict>
            <key>Name</key>
            <string>title</string>
            <key>Type</key>
            <string>string</string>
        </dict>
    </array>
    <key>DataClass</key>
    <string>com.mycompany.SyncExamples</string>
    <key>IdentityProperties</key>
    <array>
        <string>title</string>
    </array>
    <key>Name</key>
    <string>com.anothercompany.Calendar</string>
    <key>Relationships</key>
    <array>
        <dict>
            <key>InverseRelationships</key>
            <array>
                <dict>
                    <key>EntityName</key>
                    <string>com.mycompany.syncexamples.Event</string>
                    <key>RelationshipName</key>
                    <string>com.anothercompany.calendar</string>
                </dict>
            </array>
            <key>Name</key>
            <string>events</string>
            <key>Ordinality</key>
            <string>many</string>
            <key>Target</key>
            <array>
```

```
                <string>com.mycompany.syncexamples.Event</string>
            </array>
        </dict>
    </array>
</dict>
```

4. Finally, add localization strings for new entities and properties to `Schema.strings`:

```
/* Relationship: The calendar associated with an event. */
"com.mycompany.syncexamples.Event/com.anothercompany.calendar" = "Calendar";

/* Entity: A calendar, group of events.*/
"com.anothercompany.Calendar" = "Calendar";

/* Attribute: The calendar title. */
"com.anothercompany.Calendar/title" = "Title";

/* Relationship: The events associated with a calendar. */
"com.anothercompany.Calendar/events" = "Events";
```

Note: Make sure that the original schema is registered with the sync engine before registering a schema extension.

Registering Schemas

Typically, schemas are owned by one client and possibly used by multiple clients, although this approach is not enforced. Usually a schema owner registers the schema with the sync manager. Otherwise, multiple clients must coordinate registering a shared schema. It is highly recommended that you register the schema periodically even if it does not change—for example, register the schema each time your application launches. However, if a schema changes, update it with caution because changing a schema may cause records to be deleted and cause some clients to slow sync.

You register schemas with the shared `ISyncManager` using the `registerSchemaWithBundlePath:` method. Typically, a schema is stored in a bundle along with other related files such as images. The following code fragment gets the schema bundle and registers the schema property list:

```
[[ISyncManager sharedManager]
registerSchemaWithBundlePath:@"/Library/SyncServices/Schemas/SyncExamples.syncschema"];
```

You can register a schema multiple times. The sync engine compares the old and new schema and updates the schema only if it changed. If it changed, the new schema replaces the old one, records in the truth database may be deleted, and clients that use the schema may slow sync.

You can also unregister a schema, but doing so removes all records associated with the entities defined in that schema. You unregister a schema using the `unregisterSchemaWithName:` method.

Typically, you do not need to register Apple application schemas. See *Apple Applications Schema Reference* for more information on Apple application schemas.

Read "Creating a Sync Schema" (page 39) for more information on designing your own schemas.

Registering Clients

Before you can sync any records, you need to register your client with the sync engine. You do this by specifying a unique client identifier and client description property list.

When Do You Register a Client?

The lifetime of a sync client is dependent on the lifetime of the data you are syncing, not the lifetime of the application or tool that syncs. The sync engine keeps a snapshot of the client records that it uses when mingling in order to compute the changes that will be pulled by the client—changes that were possibly made by other clients. This information is retained until you unregister your client.

If you unregister a client, the snapshot is removed along with all other information the sync engine knew about your client. Consequently, if you unregister your client when the application terminates, you will not be able to fast sync the next time it launches.

Typically, you use one client identifier per data source or data file containing the records you want to sync, and you reuse that client identifier each time your application or tool launches. For this reason, you typically use a reverse DNS-style name as your client identifier. For example, the client identifier for the Events example application is `com.mycompany.syncexamples.events`.

You can see whether a client exists before registering it (see the sample code in "How to Register a Client" (page 57)). You unregister your client only if your data is corrupted or removed.

Registering a Client

You register a client using the `registerClientWithIdentifier:descriptionFilePath:` method of `ISyncManager`. When you register a client, you specify a unique client identifier. For example, you can use a reverse DNS-style name such as `com.mycompany.syncexamples.MediaAssets`.

You also specify a client description property list that describes the client's capabilities. You cannot sync any records without specifying the supported entities and properties in the client description. The client description may contain a subset of the entities and properties defined in a schema but *must* contain all the required properties. If a client does not support required properties it will fail to register. See "Client Description Properties" (page 58) for a description of this property list.

You use the `clientWithIdentifier:` method of `ISyncManager` to get an existing registered client. In this sample code, the `getSyncClient` method returns a client if it exists; otherwise, it registers it:

```
// Returns a sync client for this application
- (ISyncClient *)getSyncClient {
    // Get an existing client
    ISyncClient *client =
         [[ISyncManager sharedManager] clientWithIdentifier:clientID];
```

```
    if (client != nil) {
        return client;
    }

    client = [[ISyncManager sharedManager] registerClientWithIdentifier:clientID
        descriptionFilePath:[[NSBundle mainBundle]
pathForResource:@"ClientDescription" ofType:@"plist"]];

    return client;
}
```

Setting Client Description Properties

The client description is a property list stored in a file so that it can be updated independent of the client process. The sync engine periodically checks the client description file for changes, and if the file changed, the sync engine updates the client's capabilities.

The properties of a client description are listed in Table 1 (page 58).

Table 1 Client description properties

Property	Description
`Type`	A string identifying the type of client. This string must be one of the following predefined values: `app` (an application like iCal or Address Book), `device` (a phone, PDA or iPod), `server` (MobileMe) or `peer` (a peer-to-peer client). The default is `app`.
`DisplayName`	A string containing the display name for this client.
`ImagePath`	Path to an image of the client. This must be an absolute path except when the description is taken from a file. If it is a relative path, then it is expected to be relative to the folder containing the client description file.
`Entities`	A dictionary mapping entity names to an array. The array contains an array of property names (both attributes and relationships) identifying the properties supported by the client on each record type. The array must include the `com.apple.syncservices.RecordEntityName` property. This key is required.
`NeverFormats-Relationships`	A boolean value set to `true` if the client never reformats the destination records of pulled relationships, `false` otherwise. The default value is `false`. If this property is set to `true`, then the sync engine can perform some optimizations on behalf of the client. Read "Formatting Records" (page 75) for how to format records. Deprecated in Mac OS X v10.6.
`FormatsRelationships`	A Boolean value set to `true` if the client might format a relationship it pulls. In Mac OS X v10.6 and later, the default value is `false`. Available in Mac OS X v10.6 and later.

Property	Description
`PullOnlyEntities`	An array containing the names of the record types for which the client will pull changes from the engine but never push changes to the engine.
`PushOnlyEntities`	An array containing the names of the record types for which the client will only push changes to the engine but never pull changes from the engine.
`SyncsWith`	A dictionary specifying the kinds of clients this client wants to sync with. See the `setShouldSynchronize: withClientsOfType:` method of `ISyncClient` for details.

For example, the following client description property list is used by the MediaAssets and Events example applications to describe their client capabilities. It happens that the client descriptions for each application are the same—each application supports both entities, Event and Media, and all their properties.

```
<plist version="1.0">
<dict>
    <key>DisplayName</key>
    <string>Events</string>
    <key>ImagePath</key>
    <string>Events.icns</string>
    <key>Entities</key>
    <dict>
        <key>com.mycompany.syncexamples.Event</key>
        <array>
            <string>com.apple.syncservices.RecordEntityName</string>
            <string>date</string>
            <string>startDate</string>
            <string>endDate</string>
            <string>title</string>
            <string>media</string>
        </array>
        <key>com.mycompany.syncexamples.Media</key>
        <array>
            <string>com.apple.syncservices.RecordEntityName</string>
            <string>date</string>
            <string>imageURL</string>
            <string>title</string>
            <string>event</string>
        </array>
    </dict>
</dict>
</plist>
```

Syncs With Properties

The properties of the `SyncsWith` dictionaries are as follows:

Property	Description
`SyncAlertTypes`	An array of the client types that this client wants to sync with. This key is required.

Property	Description
`SyncAlertToolPath`	The path of a tool that the engine invokes when a client of the specified type starts syncing. If it is a relative path, then it is expected to be relative to the folder containing the client description file. The tool is passed at least two key-value pairs as arguments on the command line. The "--sync" key is followed by the client identifier. The “--entitynames” key is followed by a string containing the entity names to be synced separated by commas. The order of the key-value pairs is arbitrary. Any other command-line arguments should be ignored since they might be private. In general, options are always prefixed with “--” and followed by an optional value. The option and value are delimited by spaces.

See “Setting Alert Handlers Using the Client Description” (page 66) for an example of a client description property list that uses the `SyncsWith` property.

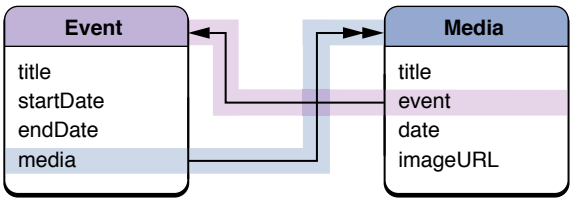
Syncing Relationships

To sync relationships, you need to describe the relationships in your schema and add code to your pushing and pulling methods to handle the relationships. Typically, you transform your representation of relationships (for example, a collection of objects) to Sync Services' representation of relationships (an array of record identifiers) and back.

Adding Relationships to Your Schema

You add relationships to your object model by adding a `Relationships` key-value pair to your entity descriptions in your schema. See "Creating a Sync Schema" (page 39) for a complete list of entity and relationship properties. This article contains an example of a to-one and inverse to-many relationship as shown in Figure 1 (page 61).

Figure 1 Object model for MediaExample



Describing To-One Relationships

For example, follow these steps to add a to-one relationship from Media to Event in the MediaExample schema.

1. Add a `Relationships` key and an array value to the `com.mycompany.syncexamples.Media` entity description.
2. Add a single dictionary to the array describing the to-one relationship.
3. Set the name of the relationship to `event`.
4. Optionally, set the display name to `Event`.
5. Set the ordinality to `one`.
6. Set the target to an array containing a single entity name: `com.mycompany.syncexamples.Event`.

The following property list fragment describes the to-one relationship, called `event`, from `com.mycompany.syncexamples.Media` to `com.mycompany.syncexamples.Event`:

```
<dict>
    <key>Name</key>
    <string>com.mycompany.syncexamples.Media</string>
    ...
    <key>Relationships</key>
    <array>
        <dict>
            <key>Name</key>
            <string>event</string>
            <key>Ordinality</key>
            <string>one</string>
            <key>Target</key>
            <array>
                <string>com.mycompany.syncexamples.Event</string>
            </array>
        </dict>
    </array>
</dict>
```

Describing To-Many Relationships

Similarly, you can define a to-many relationship from Media to Event by following the same steps in "Describing To-One Relationships" (page 61) except that you set the ordinality to `many`. The following property list fragment describes a to-many relationship, called `media`, from `com.mycompany.syncexamples.Event` to `com.mycompany.syncexamples.Media`.

```
<dict>
    <key>Name</key>
    <string>com.mycompany.syncexamples.Event</string>
    ...
    <key>Relationships</key>
    <array>
        <dict>
            <key>Name</key>
            <string>media</string>
            <key>Ordinality</key>
            <string>many</string>
            <key>Target</key>
            <array>
                <string>com.mycompany.syncexamples.Media</string>
            </array>
        </dict>
    </array>
</dict>
```

Describing Inverse Relationships

Sync Services supports inverse relationships and will maintain them in the sync engine even if you don't maintain them in your local data source. You are not required to push both sides of an inverse relationship.

For example, the to-one and to-many relationships between the Media and Event entities are inverse relationships (see Figure 1 (page 61)). Every time you set the `event` property of a Media object, you would expect that Media object to be added to the Event's collection of Media objects, called `media`. Similarly, if you add a Media object to an Event object's `media` property, you would expect the Media object's `event` property to be set to that Event object.

You specify an inverse relationship by adding an `InverseRelationship` property to the relationship description. For example, in the description of the `event` to-one relationship from Media to Event, you add the following `InverseRelationship` key-value pair. The `EntityName` property of an inverse relationship specifies the destination entity, and the `RelationshipName` property specifies the relationship in the destination entity that is inverse.

```
<key>InverseRelationships</key>
<array>
    <dict>
        <key>EntityName</key>
        <string>com.mycompany.syncexamples.Event</string>
        <key>RelationshipName</key>
        <string>media</string>
    </dict>
</array>
```

Similarly, in the description of the `media` to-many relationship from Media to Event, you add this `InverseRelationship` key-value pair:

```
<key>InverseRelationships</key>
<array>
    <dict>
        <key>EntityName</key>
        <string>com.mycompany.syncexamples.Media</string>
        <key>RelationshipName</key>
        <string>event</string>
    </dict>
</array>
```

Pushing Relationships

Typically, you need to transform your objects to sync records before pushing them to the sync engine. If you want to sync relationships in your object model, you need to convert your representation to the sync engine's representation.

The sync engine represents to-one and to-many relationships as arrays of record identifiers belonging to the destination objects. The value of a to-one relationship contains a single record identifier whereas the value of a to-many relationship may contain many record identifiers.

For example, the following output shows a Media record containing a to-one relationship called `event`:

```
MediaAssets[789] pushing sync record={
    "com.mycompany.syncservices.RecordEntityName" =
"com.mycompany.syncexamples.Media";
    date = 2004-05-22 00:00:00 -0700;
    event = ("3571D5D0-C0A5-11D8-A57D-000A95BF2062");
    imageURL = "file://2004/05/22/IMG_1106.JPG";
    title = "IMG_1106.JPG";
```

```
}
```

Conversely, the following output shows an Event record containing a to-many relationship called `media`:

```
MediaAssets[789] pushing sync record={
    "com.mycompany.syncservices.RecordEntityName" =
"com.mycompany.syncexamples.Event";
    media = (
        "3DD4A368-C0A5-11D8-B7DA-000A95BF2062",
        "3DD4EDAD-C0A5-11D8-B7DA-000A95BF2062",
        "3DD53833-C0A5-11D8-B7DA-000A95BF2062",
        "3DD5838E-C0A5-11D8-B7DA-000A95BF2062",
        "3DD5CDE3-C0A5-11D8-B7DA-000A95BF2062"
    );
    startDate = 2004-05-22 00:00:00 US/Pacific;
    title = "Burning Down the House";
}
```

Pulling Relationships

Typically, when pulling records, you need to convert the sync representation of relationships to your representation. You can transform relationships when pulling records or later when you go to use them. Be aware that some destination objects may be new—not pulled from the sync engine yet—so if you are resolving relationships when pulling records, you should resolve them *after* all records are pulled.

Be careful when pulling relationships and changing record identifiers at the same time. When you use the `changeEnumeratorForEntityNames:` method to pull changes, the targets for any pulled relationships use the old record identifiers, not new record identifiers that you can change using the `clientAcceptedChangesForRecordWithIdentifier:formattedRecord:newRecordIdentifier:` method. If you change record identifiers as you apply changes, then it is your responsibility to fix relationships pulled during the same sync session that may use the old identifier. For example, if you pull a new record that has an inverse to-one relationship to another record and change the record identifiers, then the targets of the relationships, stored in the `ISyncChange` object, still use the old identifiers. It's your responsibility to keep track of the new and old identifiers to resolve these relationships at the end of the pulling phase. Alternatively, you can change record identifiers after applying all changes using the `clientChangedRecordIdentifiers:` method.

Similar to "Pushing Relationships" (page 63), pulled relationships are represented by arrays of record identifiers. It's the application's responsibility to implement the code that takes a record identifier and returns an instance of the corresponding entity. Remember that record identifiers are unique across all entities.

Syncing with Other Clients

Your client **trickle syncs** if it syncs frequently and simultaneously when other related clients sync—clients that share the same entities as your client. This article explains how to configure your client to sync with other clients.

Your client syncs with other clients by setting alert handlers to be invoked when clients of a certain type or types sync. Clients can be one of these types: application, device, server or peer (see "Clients" (page 20)). Your client is assumed to be an application unless you specify otherwise in the client description property list (see "Client Description Properties" (page 58) for details).

Typically, a slow client, such as a device client, alerts a fast client, such as the MobileMe server, to sync. However, a fast client never invites slow clients to join its sync session. For example, if the user syncs the MobileMe server, the user expects it to sync quickly, not be delayed because every slow device hooked up to the computer suddenly wants to join the sync session (unless the user specifically requested that those devices be included). Conversely, if a phone client periodically syncs, then it might alert MobileMe to sync without any performance penalty.

You can specify alert handlers programmatically or by modifying the client description property list. However, delays can occur when syncing with other clients. Therefore, Sync Services offers a nonblocking API so that your applications can still be responsive to the user when syncing.

Setting Alert Handlers Programmatically

The first step is to specify the types of clients your client is interested in syncing with. You can send `setShouldSynchronize:withClientsOfType:` multiple times to an `ISyncClient` object to set multiple types. For example, the MediaAssets and Events example applications are configured to sync with any application and server clients that share the same entities as follows:

```
// Specify client types
[client setShouldSynchronize:YES
           withClientsOfType:ISyncClientTypeApplication];
[client setShouldSynchronize:YES
           withClientsOfType:ISyncClientTypeServer];
```

Next you set an alert handler to be invoked when related clients sync. For example, in this code fragment from the MediaAssets example application, `client:mightWantToSyncEntityNames:` is invoked when any application or server client syncs the Media or Event entities:

```
// Specify alert handler
[client setSyncAlertHandler:self selector:
    @selector(client:mightWantToSyncEntityNames:)];
```

Setting Alert Handlers Using the Client Description

Alternatively, you can specify the alert types and an alert tool using the client description property list described in "Client Description Properties" (page 58). For example, the following fragment of a client description property list specifies that an alert tool, called MySyncTool, should be launched whenever an application or server begins syncing entities used by your application.

```
<key>SyncsWith</key>
<dict>
    <key>SyncAlertToolPath</key>
    <string>MySyncTool</string>
    <key>SyncAlertTypes</key>
    <array>
        <string>app</string>
        <string>server</string>
    </array>
</dict>
```

Note that the sync tool path is typically relative to the location of the client description file. Also, if you register both an alert handler programmatically and an alert tool, as show above, then only the alert handler is invoked.

Maintaining User Responsiveness

Delays may occur while your client is waiting for related clients to join its sync session. Consequently, Sync Services provides a blocking and nonblocking API when beginning a sync session. See "Starting a Sync Session" (page 27) for code examples of the `beginSession...` class methods of ISyncSession.

If you use the blocking API, you can specify how long you are willing to wait for the session to begin. Because unknown events may occur between multiple processes, you should never wait indefinitely for a session to begin. For example, if you have a Cocoa application and a session doesn't begin after 5 seconds, you might return to the main run loop to process any user events and sync later.

Another delay can occur when the sync session enters the mingling state. Mingling begins when all clients have finished pushing their records and invoked one of the `prepareToPullChanges...` methods of `ISyncSession`. Your client may be delayed if a slower client is still pushing changes. Consequently, Sync Services provides a blocking and nonblocking API to begin the mingling state. See "Mingling" (page 33) for code examples of the different `ISyncSession prepareToPullChanges...` methods.

Using Sync Anchors

Sync anchors provide a mechanism for tracking the success or failure of phases within a sync session. Sync anchors mark when changes are successfully pushed and pulled. This information is used by the sync engine, in the next session, to select a better sync mode. A client may request a sync mode but the sync engine ultimately decides whether a client can fast sync or needs to refresh or slow sync. Using sync anchors can improve performance by fast syncing more often and avoids serious errors by forcing a refresh or slow sync when necessary.

You should use sync anchors unless you implement your own mechanism for tracking whether records are successfully pushed and pulled. A sync anchor is a string object, unique per client and per entity or data class, that is saved periodically throughout a sync session. During the next sync session, the sync engine compares the client's locally stored sync anchors with its copies to identify discrepancies and consequently select an appropriate sync mode. For example, if the client sync anchors are not the same as the sync engine sync anchors and the client is an application (of type `ISyncClientTypeApplication`), an alert panel appears asking the user to select an appropriate sync mode. If the client is not an application, the client slow syncs.

If you do not use sync anchors, then there may be some cases where the sync engine cannot determine accurately when a client can fast sync—that is, when it is not necessary to push every record. Sync anchors help determine when a fast sync is ok and therefore, significantly improve performance of your application by fast syncing more often, especially for large data sets.

There are also some cases when the sync engine might allow a fast sync when the client should slow sync. For example, if there is a communication failure between a client and its data store, then the sync engine might assume a fast sync is ok. The local data store could also be restored from backup (independent of the application) and both the application and the sync engine might mistakenly allow a fast sync. By using sync anchors, the sync engine can detect these types of errors and take the appropriate action.

This article describes the sync anchor methods you use when managing your sync session using either the `ISyncSession` or `ISyncSessionDriver` classes. You do not need to worry about sync anchors if you are syncing Core Data objects (see "Syncing Core Data Applications" (page 85) for more details).

The sync anchor methods for the `ISyncSession` and `ISyncSessionDriver` classes are similar. You are responsible for providing the sync anchors used in the previous sync session when beginning a sync session and providing new sync anchors for the current session after both the pushing and pulling phases.

Creating Sync Anchors

The sync anchor parameter to Sync Services methods is a dictionary where the keys are the entity names and the values are the sync anchors. A sync anchor is a unique string per entity or data class.

Typically, all entities in the same data class have the same sync mode. Therefore, you can optionally create a sync anchor per one representative entity in a data class. The sync engine then uses the same sync anchor for all entities in that data class. However, if you provide sync anchors for two or more entities per data class, then you need to specify sync anchors for all entities in that data class. Otherwise, a missing sync anchor for an entity causes that entity to refresh sync.

This code sample creates one sync anchor per a single data class that an application syncs:

```
NSString *representativeEntityName = [[self entityNames] objectAtIndex:0];
NSDictionary *syncAnchors = [NSDictionary dictionaryWithObject:[[NSDate date]
description] forKey:representativeEntityName];
```

Saving Sync Anchors

To use sync anchors, you need to provide the last sync anchors when beginning a sync session and save sync anchors at various times during a sync session. Therefore, you need to decide where and how to store your sync anchors. This section provides some tips on how and when to save sync anchors.

Typically, you store the sync anchors with your local records so they are synchronized. Keeping the sync anchors with your local records is important because if you provide the wrong sync anchors, it defeats the purpose of using sync anchors. You can avoid this by saving the sync anchors in the same file as your local records or saving the sync anchors in a separate file that is in the same bundle as your local records.

It's strongly recommended that you save your sync anchors whenever you create them during a sync session. It's also recommended that you save your sync anchors with your local records just before committing your changes at the end of the pulling phase. This synchronizes the state of the local records with the sync anchors and avoids problems should a session be canceled.

You also need to decide what to do if a sync session is canceled. If you save local records with sync anchors as recommended during the pushing and pulling phases, then you can safely revert to the last saved state when a session is canceled. Again, it's easier to do so if you store your local records and sync anchors together. Read "Canceling" (page 37) in "Managing Your Sync Session" (page 25) for complete details on the state of the sync engine when a session is canceled.

Read "Using Sync Anchors with ISyncSession" (page 68) or "Using Sync Anchors with ISyncSessionDriver" (page 69) for details on using sync anchors for each approach to managing your sync sessions.

Using Sync Anchors with ISyncSession

This section describes the sequence of steps you follow to use sync anchors when you are managing a sync session using an `ISyncSession` object. Read "Managing Your Sync Session" (page 25) for a complete descriptions of all the steps in managing a sync session.

Beginning a Sync Session

Invoke either the `beginSessionWithClient:entityNames:beforeDate:lastAnchors:` or `beginSessionInBackgroundWithClient:entityNames:target:selector:lastAnchors:` methods of `ISyncSession` to begin a sync session.

The `beginSessionWithClient:entityNames:beforeDate:lastAnchors:` method is synchronous—blocks until all other clients have joined the sync session—and the `beginSessionInBackgroundWithClient:entityNames:target:selector:lastAnchors:` method is asynchronous—sends a message to a target when a sync session is created.

You pass the sync anchors used in the previous successful sync as the `lastAnchors:` parameter to each of these methods. If it's the first time your application is syncing, then pass `nil` as the `lastAnchors:` parameter and your client will refresh sync.

Note: If you intend to use sync anchors in the next sync, use only the sync anchors versions of the `beginSession...` methods of `ISyncSession`. You cannot use sync anchor methods in combination with non-sync anchor methods.

Pushing

After pushing changes to the sync engine, create new sync anchors and save them locally just before invoking the `clientFinishedPushingChangesWithNextAnchors:` method. Pass the new sync anchors as the parameter of this method. Optionally, save your local records, too, so that the new sync anchors match. Read "Creating Sync Anchors" (page 67) for how to create sync anchors.

Pulling

After pulling changes from the sync engine and applying them to your local records, create new sync anchors and save them locally just before invoking the `clientCommittedAcceptedChangesWithNextAnchors:` method. Pass the new sync anchors as the parameter. You should also save your local records before invoking this method so that your last sync anchors match. Read "Creating Sync Anchors" (page 67) for how to create sync anchors.

Using Sync Anchors with ISyncSessionDriver

Using sync anchors is optional but recommended if you are using a sync session driver. All you need to do is implement the `nextAnchorForEntityName:` and `lastAnchorForEntityName:` methods of the `ISyncSessionDriverDataSource` protocol to use sync anchors.

Simply implement the `nextAnchorForEntityName:` method to return the next sync anchor for the specified entity name. Store the sync anchor locally and implement the `lastAnchorForEntityName:` method to return it the next time it is invoked during the next sync session. Note that if you implement one of these methods, you need to implement the other one, too.

In a way that is similar to using an `ISyncSession` object, you can create one sync anchor per representative entity in a data class. In this case, you implement the methods to return the same sync anchor for all entities in the same data class.

Filtering Records

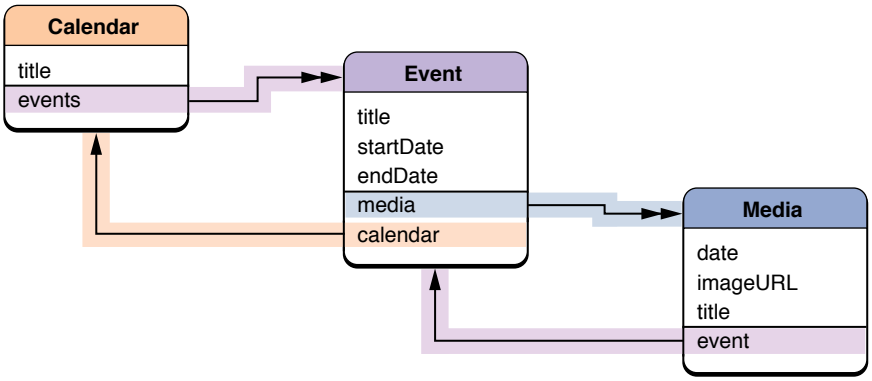
You can use the `ISyncFiltering` protocol and the `setFilters:` method of `ISyncClient` to filter the types of records your application syncs. Your filter can contain any custom logic you like as long as the sync engine can archive the filter and compare it with another filter. This article describes how to implement a filter using an example—extending the MediaExample application.

Creating a Filter Class

To implement a filter, you create a filter class that conforms to the `NSCoding` and `ISyncFiltering` protocols. The class needs to conform to `NSCoding` so that it can be archived. The `ISyncFiltering` protocol defines the methods that the sync engine will invoke to perform the actual filtering operation. The filter class also needs to override `isEqual:` so that the sync engine can compare two filters.

For example, suppose you want to add to the MediaExample schema an entity called Calendar, which maintains a collection of events. That way, when you import an iCal file, you can group the events together. You can do this by creating a Calendar entity with an identity attribute, such as a name or title, and an inverse to-many relationship to Event. Figure 1 (page 71) illustrates the modified schema.

Figure 1 Extended MediaExample schema



You can modify the Events application to group events by calendar—when you import events, you add Event records to a Calendar record. You can then use a filter to sync only those events that belong to your Calendar records—when the filter is set, no other events will be pushed or pulled. This section describes how to create a filter.

Follow these steps to create a filter that rejects all Event records that do not belong to a specified Calendar:

1. In this example, the filter class is called CalendarFilter and defines one instance variable, the record identifier of the Calendar record:

```
@interface CalendarFilter : NSObject <NSCoding, ISyncFiltering> {
```

```
    NSString *calendarIdentifier;
}

- (NSString *)calendarIdentifier;
- (void)setCalendarIdentifier:(NSString *)value;

@end
```

2. Next add implementations of the NSCoding protocol methods, `initWithCoder:` and `encodeWithCoder:`, to `CalendarFilter.m`:

```
- (id)initWithCoder:(NSCoder *)coder
{
    if ( [coder allowsKeyedCoding] ) {
        calendarIdentifier = [[coder decodeObjectForKey:@"calendarIdentifier"]
 retain];
    } else {
        calendarIdentifier = [[coder decodeObject] retain];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)coder
{
    if ([coder allowsKeyedCoding]) {
        [coder encodeObject:calendarIdentifier forKey:@"calendarIdentifier"];
    } else {
        [coder encodeObject:calendarIdentifier];
    }
    return;
}
```

3. Then add implementations for the `ISyncFiltering` methods, `supportedEntityNames` and `shouldApplyRecord:withRecordIdentifier:`, to `CalendarFilter.m`. The `supportedEntityNames` method simply returns the names of the entities that you are filtering.

```
- (NSArray *)supportedEntityNames
{
    return [NSArray arrayWithObject:@"com.mycompany.syncexamples.Event"];
}
```

The `shouldApplyRecord:withRecordIdentifier:` method performs the actual filter operation. In this example, the filter rejects all Event records that do not belong to the Calendar record specified by `calendarIdentifier`.

```
- (BOOL)shouldApplyRecord:(NSDictionary *)record withRecordIdentifier:(NSString
 *)recordIdentifier
{
    NSArray *relationship = [record valueForKey:@"calendar"];
    if ((relationship != nil) && ([relationship count] > 0) &&
        [[relationship objectAtIndex:0] isEqual:[self calendarIdentifier]])
        return YES;
    else
        return NO;
}
```

Setting Filters

You can set multiple filters. For example, you can create multiple instances of CalendarFilter—one for each Calendar instance in your application—or create an entirely different filter for other entities.

You create an instance of a filter and activate it by sending `setFilters:` to your sync client. The `setFilters:` method takes an array of filters as the argument. The sync engine compares the filters with existing filters (using `isEqual:`) and replaces only those filters that are different.

```
id filter = [[CalendarFilter alloc] init];
[filter setCalendarIdentifier:calendarIdentifier];
[mySyncClient setFilters:[NSArray arrayWithObject:filter]];
```

Note: You can also define a Calendar entity and a to-many inverse relationship to Event as extensions to the existing schema used by the MediaExample applications. For example, the MediaAssets application manipulates Media records but has no use for a Calendar entity. The MediaAssets application can use the existing schema that defines just the Event and Media entities. See "Creating a Schema Extension" (page 51) for details on how to extend an existing schema.

Using Filters

If you set a filter and push records that do not match that filter, the sync engine will issue deletes for those pushed records on the subsequent pull phase. This is because filters are applied to the pulled records only, not the pushed records. If this behavior is not desired, clients should implement their own filters when pushing records (for example, don't push records that you don't want to pull), or modify their filters to accept all pushed records.

You can also use the `ISyncFilter` class to create a combination of filters—that is, create a filter that is a logical `AND` or `OR` of multiple filters.

Filtering Using the Client Description

You can also filter the entities and properties your client syncs using the client description property list. The client description property list specifies the entities and properties you support and can be a subset of the entities and properties defined in your schemas. See "Registering Clients" (page 57) for more details on the format of client description property lists.

For example, if you want to use the existing iCal Calendars schema but only want to sync Calendar and Event entities, and a minimal set of their properties, then you could edit the client description property list as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>DisplayName</key>
```

```
    <string>FooApp</string>
    <key>ImagePath</key>
    <string>FooApp.icns</string>
    <key>Entities</key>
    <dict>
        <key>com.apple.calendars.Calendar</key>
        <array>
            <string>title</string>
            <string>notes</string>
            <string>events</string>
        </array>
        <key>com.apple.calendars.Event</key>
        <array>
            <string>summary</string>
            <string>start date</string>
            <string>calendar</string>
        </array>
    </dict>
</dict>
</plist>
```

Although the Calendar entity defines many more properties, when your client syncs this entity, it will only push and pull the `title`, `notes` and `events` properties. (The `events` property is an inverse to-many relationship to Event.) Similarly, it will only push and pull the `start date`, `summary` and `calendar` properties of Event records.

See *Apple Applications Schema Reference* for a complete description of the iCal Calendars schema.

Formatting Records

When pulling records from the sync engine, the client can change the format of property values without the sync engine interpreting the reformatting as changes to the values and mistakenly syncing the changes to other clients.

For example, a device that has limited capacity may truncate strings to save space—limit all strings to 12 characters or less—or reformat phone numbers to a preferred style. Records that are changed in this way are called **formatted records**. Typically, you do not want the sync engine to interpret format changes as changes to actual property values and push them to all other clients.

Therefore, if you change the format of a pulled record, you need to notify the sync engine when accepting the record as follows. First you create an NSDictionary containing the key-value pairs of the formatted properties and add the `ISyncRecordEntityNameKey` key to specify the record's entity name. Then you pass this dictionary as the *`formattedRecord`* argument to `clientAcceptedChangesForRecordWithIdentifier:formattedRecord:newRecordIdentifier:`. Thereafter, the sync engine remembers the client's formatted values and does not generate false changes during a slow sync.

```
NSDictionary *formattedRecord =
    [NSDictionary dictionaryWithObjectsAndKeys:
        entityName, ISyncRecordEntityNameKey,
        [myRecord valueForKey:@"title"], @"title", nil];
[session clientAcceptedChangesForRecordWithIdentifier:recordIdentifier
                                      formattedRecord:formattedRecord
                                    newRecordIdentifier:nil];
```

Using a Session Driver

`ISyncSessionDriver` is a class that encapsulates the entire syncing process including setting up clients, registering schemas, and managing sync sessions. Applications can optionally use an `ISyncSessionDriver` object to manage their sync sessions instead of creating `ISyncClient` and `ISyncSession` objects directly. `ISyncSessionDriver` works well for typical applications that have a local data store and that push and pull records during every sync. `ISyncSessionDriver` uses a delegation model where application-specific objects implement some required and some optional methods.

The application supplies two objects to the driver to help control a sync session. The responsibility of the **data source** is to provide client information and maintain local records—the data source conforms to the `ISyncSessionDriverDataSource` protocol. The responsibility of the **delegate** is to optionally intervene during the syncing process and perform application-specific tasks. By default, the data source and delegate are the same object. Explicitly set the delegate object if you want a separate object to control a sync session.

The following sections describe the data source and delegate methods in the sequence they are invoked in a typical application. The process involves creating a driver, setting a delegate, and starting a sync session. Some of the data source methods are required and some are optional. All of the delegate methods are optional. The driver sends a series of messages to the data source and delegate:

1. The driver requests information from the data source, such as the client identifier and client description property list before starting a sync session.
2. The driver requests changes to local records from the data source during the pushing phase.
3. The driver sends server-side changes to the data source during the pulling phase.
4. The driver can notify the delegate before and after each phase during the syncing process. For example, notify the delegate when the pulling phase is done or when a session is about to be canceled.

The data source and delegate are also responsible for identifying client-side errors and notifying the driver. They can cancel a sync session at any time. Read "Handling Errors" (page 83) for details.

An `ISyncSessionDriver` object can be used for multiple sync sessions—typically, there's one `ISyncSessionDriver` object per application.

This chapter uses code fragments from the *SimpleStickies* sample code project. Also read *ISyncSessionDriverDataSource Protocol Reference* and *ISyncSessionDriver Class Reference* for a complete description of the data source and delegate methods.

Creating a Driver

First you create an `ISyncSessionDriver` object and set its data source as follows:

```
// Create an ISyncSessionDriver object
```

```
    sessionDriver = [[ISyncSessionDriver sessionDriverWithDataSource:myDataSource]
retain];
```

If you do not set the delegate using the `setDelegate:` method, the data source and delegate are the same object.

Some data source methods are required. For example, a data source must provide a client identifier and client description property list in order for an `ISyncSessionDriver` object to create an `ISyncClient` object. The `sessionDriverWithDataSource:` method raises an exception if required methods are not implemented. The following sections cover the different data source methods in detail.

Starting a Sync Session

You start a sync session by sending either `sync` or `startAsynchronousSync:` to an `ISyncSessionDriver` object. The following code fragment uses `sync` as follows:

```
// Begin a sync session.
NSError *syncError;
BOOL syncSuccess = [sessionDriver sync];
if (syncSuccess == NO){
    syncError = [sessionDriver error];
    NSAlert *theAlert = [NSAlert alertWithError:syncError];
    [theAlert runModal];
}
```

The `sync` method sends a series of messages to the data source and delegate to create the client and session objects. The data source is required to implement some methods invoked during pushing and pulling, and the delegate can optionally implement other callback methods during the entire process.

The `sync` method returns `NO` if an error occurs in either the sync session or the application. How you handle the error is application dependent. Typically, you log or display the error message and set the next preferred sync mode accordingly. See "Handling Errors" (page 83) for more details.

Optionally, use the `startAsynchronousSync:` method if you need to sync asynchronously. In this case, you might implement the delegate methods to get notification when the sync session finishes or cancels.

Providing Client Information

The driver takes care of the details of registering schemas and clients with the sync engine so you don't have to. The driver sends the following messages to the data source to register the client and any schemas.

Providing a Client Identifier

A data source must provide a client identifier—a unique string that identifies your application—by implementing the `clientIdentifier` method. Typically, the client identifier is a reverse DNS-style string that identifies your organization and application as follows:

```
- (NSString *)clientIdentifier
```

```
{
    return @"com.mycompany.SimpleStickies";
}
```

Providing a Client Description

A data source must provide a client description property list by implementing the `clientDescriptionURL` method. In this sample code, the `ClientDescription.plist` file is located in the application `Resources` directory:

```
- (NSURL *)clientDescriptionURL
{
    return [NSURL fileURLWithPath:[[NSBundle mainBundle]
pathForResource:@"ClientDescription" ofType:@"plist"]];
}
```

The client description property list contains additional information about the client as well as the entities and properties that it syncs. The entities and properties may be a subset of the entities and properties defined in the schemas. See "Client Description Properties" (page 58) for a complete description of the client description property list.

Providing Schemas

The data source must implement the `schemaBundleURLs` method to return an array of NSURL objects containing the paths to the schemas. In this sample code, the schema called `Stickies.syncschema` is located in the application's `Resource` directory:

```
- (NSArray *)schemaBundleURLs {
    return [NSArray arrayWithObject:[NSURL fileURLWithPath:
        [[[NSBundle mainBundle] resourcePath]
stringByAppendingPathComponent:@"Stickies.syncschema"]]];
}
```

This method is invoked once before each sync session starts so it should return the paths to all the schemas used by your application. See "Creating a Sync Schema" (page 39) for how to create your own schema or extend an existing schema.

Providing Entity Names

The driver needs to know which entities from the client description property list you intend to push and pull during this sync session. The data source may optionally implement the `entityNamesToSync` method to return a subset of the entities declared in the client description property list. If the data source does not implement this method, the default is to sync all entities in the client description property list. The implementation of this method simply returns an NSArray object (containing NSString objects) and is application-specific.

The data source can optionally implement the `entityNamesToPull` method to specify a subset of entities to pull. Implement this method only if the entities are different from those returned by the `entityNamesToSync` method or declared in the client description property list.

Negotiating

All clients need to negotiate a sync mode before pushing and pulling records. The driver checks to see whether the data source prefers a particular sync mode.

The data source must implement the `preferredSyncModeForEntityName:` method. You may return a different result for each entity name that is passed as a parameter to this method. It is therefore the responsibility of the application to remember the previous sync mode and track errors per entity name if applicable. This method may return one of the constants described in *ISyncSessionDriverDataSource Protocol Reference*.

For example, when you first sync you might want to request a slow sync and thereafter, a fast sync. However, if you failed to save records during the last sync, you might request a refresh or slow sync. The data source maintains the state of its local records and tracks changes so it can make the appropriate request; otherwise, it should always request a slow sync.

Pushing

What methods are invoked during the pushing phase depends on how the data source tracks changes to local records and what the current sync mode is.

Pushing Changes

During a slow or refresh sync, the driver requests all the client records so that it can compare them with the records in the truth database and resolve conflicts. Therefore, the data source is required to implement the `recordsForEntityName:moreComing:error:` method to return all records. Records are expected to be a dictionary representation. Read "`Pushing Records`" (page 31) for a description of a sync record.

During a fast sync, the driver sends a request to the data source for all changes to local records. If the data source tracks changes to individual properties then it can push just the changes. To do this the data source implements the `changesForEntityName:moreComing:error:` method to return an array of `ISyncChange` objects that describe the changes. See *ISyncChange Class Reference* for how to create these objects.

Otherwise, if the data source knows only which records changed, it pushes all the changed records and the sync engine compares the old records with the new records to apply changes to individual properties. The data source implements the `changedRecordsForEntityName:moreComing:error:` method to return the changed records. Records are expected to be a dictionary representation. Read "Pushing Records" (page 31) for a description of a sync record.

The data source is required to implement one of these methods if the `preferredSyncModeForEntityName:` method may return `ISyncSessionDriverModeFast`.

Pushing Deletions

During a fast sync, the driver also requests the record identifiers for records that were deleted. The data source can either pass back deletion changes using an `ISyncChange` object returned by the `changesForEntityName:moreComing:error:` method, or implement the `identifiersForRecordsToDeleteForEntityName:moreComing:error:` method to return the record identifiers.

The `identifiersForRecordsToDeleteForEntityName:moreComing:error:` method is optional.

Batching

All the pushing methods support batching. Set the value referenced by the the `moreComing:` parameter to `YES` if you want to batch changes or records. The method is invoked repeatedly until the `moreComing:` parameter is set to `NO`.

Pulling

Typically, the data source just needs to apply changes during the pulling phase, but the data source can also reformat records and change record identifiers.

Applying Changes

Typically, the driver sends changes to the data source during the pulling phase. Changes may include deletions, additions, and changes to existing records. Therefore, the data source is required to implement the `applyChange:forEntityName:remappedRecordIdentifier:formattedRecord:error:` method. The implementation should send the `type` `ISyncChange` method to the `applyChange:` parameter to determine the type of change as in this sample code:

```
- (ISyncSessionDriverChangeResult)applyChange:(ISyncChange *)change
                                forEntityName:(NSString *)entityName
                          remappedRecordIdentifier:(NSString **)outRecordIdentifier
                                formattedRecord:(NSDictionary **)outRecord
                                          error:(NSError **)outError
{
    // Delete a record
    if ([change type] == ISyncChangeTypeDelete){
        ...
    }

    // Add a record
    else if ([change type] == ISyncChangeTypeAdd){
        ...
    }
    // Change a record
    else {
        ...
    }
    return ISyncSessionDriverChangeAccepted;
```

```
}
```

How you apply the changes to your local records is application dependent. See *ISyncChange Class Reference* for other methods the data source can use to get the changes.

This method returns one of the change constants described in *ISyncSessionDriverDataSource Protocol Reference* to notify the driver if the change is successfully applied. Otherwise, the local records can become out of sync with the truth database and the client should slow sync the next time it syncs.

Deleting All Records

Occasionally, the driver requests that the data source delete all local records—for example, when due to other circumstances the client must pull the truth. Therefore, the data source is required to implement the `deleteAllRecordsForEntityName:error:` method.

Formatting

When pulling records from the sync engine, the client can optionally change the format of property values without the sync engine interpreting the reformatting as changes to the values and mistakenly syncing the changes to other clients. Use the `formattedRecord:` parameter in the `applyChange:forEntityName:remappedRecordIdentifier:formattedRecord:error:` method to specify an alternate format for a record. Read "Formatting Records" (page 75) for more details.

Changing Record Identifiers

Occasionally a client needs to change the local record identifiers supplied by the sync engine, especially if the local data store uses its own record identifiers. If the data source knows the preferred record identifier when the `applyChange:forEntityName:remappedRecordIdentifier:formattedRecord:error:` method is invoked, it can use the `remappedRecordId:` parameter to pass back the new record identifier.

Otherwise, the data source can change the record identifiers later by sending `clientChangedRecordIdentifiers:` to the `ISyncSession` object as follows:

```
[[syncDriver session] clientChangedRecordIdentifiers:oldToNewDictionary];
```

The `session` method returns `nil` if the sync session is finished. Therefore, invoke this method before a sync session is finished. Read "Customizing Sync Behavior" (page 83) for methods the delegate can implement to do this.

Handling Sync Alerts

The session driver can optionally handle sync alerts for you. A sync alert is an invitation to join a sync session with other clients that you observe. You specify the types of clients to observe in the client description property list. By default a session driver does not handle sync alerts. Send `setHandlesSyncAlerts:` to the session driver passing `YES` as the parameter to turn on this feature.

If the session driver handles sync alerts, then it registers a sync alert handler and receives notifications for requests to join sync sessions. When the session driver receives a request, it initiates a sync session. It uses the `startAsynchronousSync:` not the `sync` method to start the sync session so it doesn't sync in the main thread.

If you prefer to handle sync alerts yourself, then you typically implement the `sessionDriver:didRegisterClientAndReturnError:` delegate method to set your own handler.

Customizing Sync Behavior

The delegate can implement any number of optional callback methods to intervene during the phases of a sync session. A client might do this to execute local database operations or monitor the progress of a sync session.

For example, the delegate might implement the `sessionDriver:didRegisterClientAndReturnError:` method to set up an application-specific sync alert handler. Read "Syncing with Other Clients" (page 65) for details on joining sync sessions.

The delegate might implement the `sessionDriver:didPullAndReturnError:` method to resolve relationships between local records after a pulling phase.

The delegate might implement the `sessionDriver:willFinishSessionAndReturnError:` method to save local changes and change record identifiers once they are known (typical actions if you are using a relational database to store your records).

Similarly the delegate might implement the `sessionDriverWillCancelSession:` method to revert the local records to the state before the sync session started.

Handling Errors

Syncing records is a complex process. Errors can occur at every level during this process including client-side errors that corrupt the local data store.

Most of the delegate and data source methods invoked by the driver during a sync session have an `...Error:` or `error:` parameter that your object can use to pass back descriptions of the error that occurred. When implementing these methods, you should return `NO` if an error occurred and set the `...Error:` or `error:` parameter accordingly. Other methods that have an `...Error:` or `error:` parameter may return `ISyncSessionDriverChangeError` to indicate an error.

If an error occurs, the driver prematurely cancels the sync session. Eventually, an `NSError` object is passed back to the method that originally started the sync session. For example, the `sync` method returns `NO` if an error occurred in either Sync Services, the data source, or the delegate. Use the `lastError` method to get and handle the error. For example, the code fragment in "Starting a Sync Session" (page 78) displays an alert panel.

Syncing Core Data Applications

Core Data provides a robust and reliable persistent store ideal for syncing applications. Using Core Data to store local sync records greatly simplifies your application. When you use Core Data, you don't need to worry about modeling your entities, saving and loading records, and tracking changes between sync sessions. However, sync clients and sync sessions are not automatically created for you, so you still need to follow some of the steps described in this document to sync your application. Managing a sync session is also simplified using a delegate, similar to the `ISyncSessionDriver` class and `ISyncSessionDriverDataSource` protocol. This article describes some of the same and alternate steps to create a syncing Core Data application.

Follow these general steps to create a Core Data application that syncs managed objects:

1. Create a Core Data application using Xcode and add `SyncServices.framework` to the project.
2. Create a managed object model using the data modeling tool in Xcode.
3. Add additional syncing information to the entities and properties in the model that you want to sync.

   Read "Adding Information to Managed Object Models" (page 86) in this article for how to add this information.
4. Create a sync schema that registers the managed object model directly.

   Read "Registering Managed Object Models" (page 88) in this article for how to add managed object models to sync schemas.
5. Register the schema.

   Read "Registering Schemas" (page 55) for how to register your schema.
6. Create and register your sync client.

   Read "Registering Clients" (page 57) for how to create clients.
7. Start a sync session.

   Read "Starting Core Data Sync Sessions" (page 89) in this article for how to start a Core Data sync session.
8. Optionally, register a handler to control a Core Data sync session.

   Read "Controlling Core Data Sync Sessions" (page 90) in this article for how to intervene during a Core Data sync session.

This chapter uses code fragments from the *StickiesWithCoreData* sample code project. Also read *ISyncSessionDriverDataSource Protocol Reference* and *ISyncSessionDriver Class Reference* for a complete description of the data source and delegate methods. See *Core Data Programming Guide* and *Core Data Framework Reference* for more information about Core Data.

Note: Core Data syncing and the managed object models schema property are available in Mac OS X v10.5 and later.

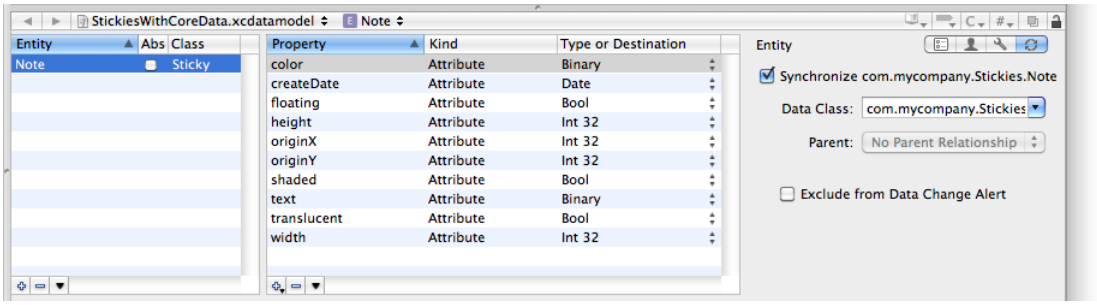
Adding Information to Managed Object Models

Use the Xcode data modeling tool to create managed object models that work with Sync Services. However, you need to add additional information to entities and properties in the model so Sync Services knows how to handle the corresponding records during a sync session. This information is in the form of key-value pairs similar to the properties you use to create a sync schema from scratch, described in "Creating a Sync Schema" (page 39).

Typically, you use the sync pane in the Xcode data modeling tool to set the keys, as shown in Figure 1. The controls in this pane correspond to the keys described in Table 1 (page 87). You can also use the user information dictionary pane in the data modeling tool to set these keys directly. If you are creating your managed object model programmatically, use the `setUserInfo:` method of the `NSEntityDescription` and `NSPropertyDescription` classes to set these keys. This article describes how to set these keys using the data modeling tool only.

Open the managed object model in Xcode and set the keys using the controls in the sync pane. Select the managed object model file—for example, select `Stickies.xcdatamodel`—in the Groups & Files outline view to edit the model. Select an entity in the entities pane of the model browser and click the sync pane icon in the detail pane as shown in Figure 1.

Figure 1 A sync pane for an entity



Most of the settings have default values. By default, all entities in the managed object model are synced unless you specifically deselect the Synchronize... option. You cannot change the `com.apple.syncservices.SyncName` key option using the sync pane because this key can be derived from the entity name. You can choose a data class from the Data Class combo box or type one in. The data class should use a reverse DNS-style name as in `com.mycompany.stickies` and match the data class name used in the schema file as shown in Listing 1 (page 89). You can also select a parent from the Parent pop-up menu if applicable.

Both the entity and the property sync pane contain an "Exclude from Data Change Alert" option that corresponds to the `com.apple.syncservices.ExcludeFromDataChangeAlert` key described in Table 1 (page 87). Use this option to reduce the number of data change alerts. Only changes to entities and properties that the user recognizes and deems significant should cause data change alerts to appear. By default, all changes cause data change alerts, so you definitely should consider selecting this option to exclude some types of changes.

For example, in the StickiesUsingCoreData application, changes to the appearance of a sticky note do not trigger data change alerts but changes to the creation date and text attributes do. To configure this behavior, select the Exclude from Data Change Alert option for all the entities and properties you want to exclude. For example, exclude changes to the `color`, `floating`, `shaded`, and `translucent` attributes of a `Note` entity.

Table 1 Entity user information dictionary keys

Key	Description
`com.apple.syncservices.SyncName`	The name of the entity that is registered with Sync Services. Typically, the sync name is different from the name used by the managed object model to refer to this entity. This key is the same as the `Name` property of an entity in a sync schema. Read "Entity Properties" (page 42) for a complete description of this value. This key is optional. If this key is not in the user information dictionary, then Sync Services attempts to construct a unique sync name as follows. If the managed object model entity name contains a `.` then it is used as the sync name. Otherwise, the sync name is the concatenation of the data class name, dot (`.`), and the entity name. For example, if the data class name is `com.apple.contacts` and the entity name is `Contact`, then the sync name is `com.apple.contacts.Contact`.
`com.apple.syncservices.DataClass`	The name of the entity's data class. This key is the same as the `Name` property of a data class in a sync schema. Read "Data Class Properties" (page 42) for a complete description of this value. This key is required.
`com.apple.syncservices.Parent`	This key is the same as the `Parent` property of an entity in a sync schema. Read "Entity Properties" (page 42) for a complete description of this value. This key is optional.
`com.apple.syncservices.Syncable`	A Boolean value of `YES` or `NO`. If `YES`, this entity is synced; otherwise, it is ignored. The default value is `YES`. This key is optional.
`com.apple.syncservices.Exclude-FromDataChangeAlert`	A Boolean value of `YES` or `NO`. If `YES`, a change to this entity is used by a data change alert; otherwise, it is ignored. This key is the same as the `ExcludeFromDataChangeAlert` property of an entity in a sync schema. Read "Entity Properties" (page 42) for a complete description of this value. The default value is `NO`. This key is optional.
`com.apple.syncservices.Identity-Properties`	The set of properties used to identify records of this entity. An array of arrays containing strings where the first array contains the identity property names (currently, any additional arrays in the parent array are ignored). This key is similar to the `IdentityProperties` property of an entity in a sync schema. Read "Identity Properties" (page 49) for a complete description. By default, none of the properties are identity properties. This key is optional.

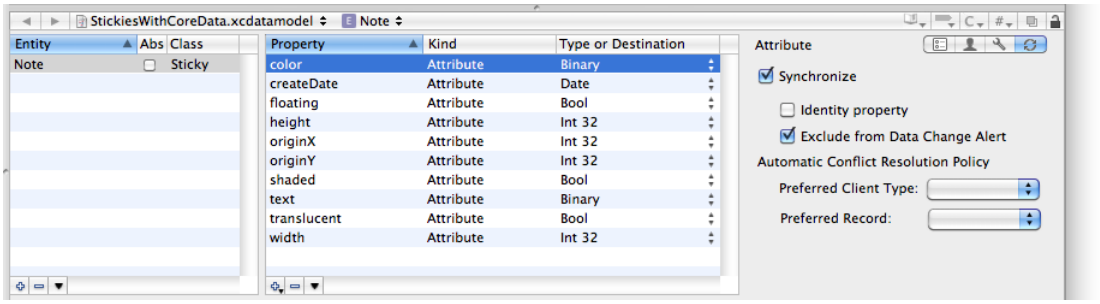
Table 2 describes the keys you set for a property in the managed object model. The keys are optional and have default values. For example, by default, all properties are synced and none of the properties are used to identify a record if a conflict occurs.

Table 2 Property user information dictionary keys

Key	Description
`com.apple.syncservices.Syncable`	A Boolean value of `YES` or `NO`. If `YES`, this property is synced; otherwise, it is ignored. The default value is `YES`. This key is optional.
`com.apple.syncservices.Exclude-FromDataChangeAlert`	A Boolean value of `YES` or `NO`. If `YES`, a change to this property is used by a data change alert; otherwise, it is ignored. This key is the same as the `ExcludeFromDataChangeAlert` property of an attribute or relationship in a sync schema. Read "Attribute Properties" (page 44) for a complete description of this value. The default value is `NO`. This key is optional.

Use the sync panel, shown in Figure 2, for a property to configure these keys. Select the Synchronize option to sync a property. Select the "Identity property" option if a property should be used to identify a record. Select the "Exclude from Data Change Alert" option if a change to a property should not display a change alert. For example, in Figure 2, the `color` attribute is synced and excluded from data change alerts —its value affects only the appearance of a sticky note.

Figure 2 A sync pane for a property



For a property such as `createDate`, the "Identity property" option would be selected but not the "Exclude from Data Change Alert" option because a change to this property should display a change alert.

Registering Managed Object Models

If you define your entities and properties entirely in the managed object model, then all you need to do is create a sync schema and set the `Name`, `DataClasses`, and `ManagedObjectModels` properties. Read "Creating a Sync Schema" (page 39) for how to create a sync schema from scratch.

Use the `ManagedObjectModels` property to register the Core Data entities directly with Sync Services. For example, Listing 1 shows the `Schema.plist` for the Stickies sync schema used by the FrancyStickiesUsingCoreData application. You do not need to declare the entities in the schema file because this information is already in the managed object model. Next register the schema with Sync Services by following the steps in "Registering Schemas" (page 55).

Listing 1 Stickies schema property list

```
<plist version="1.0">
<dict>
    <key>DataClasses</key>
    <array>
        <dict>
            <key>ImagePath</key>
            <string>Stickies.icns</string>
            <key>Name</key>
            <string>com.mycompany.Stickies</string>
        </dict>
    </array>
      <key>ManagedObjectModels</key>
    <array>
        <string>../../../StickiesUsingCoreData.mom</string>
    </array>
    <key>Name</key>
    <string>com.mycompany.Stickies</string>
</dict>
</plist>
```

Starting Core Data Sync Sessions

Although managing a sync session is handled for you, you still need to register your client and start sync sessions programmatically. Read "Registering Clients" (page 57) for how to create and register a client. The sync sessions for your Core Data application are not started automatically—it's application-dependent how often you sync and trickle sync. For example, you might sync every time the user saves the managed object context.

Before starting any sync sessions, you need to specify where to store information about which entities to sync in the next sync session. Don't skip this step if you want your application to trickle sync—fast sync frequently to improve performance. Send `setStoresFastSyncDetailsAtURL:forPersistentStore:` to your `NSPersistentStoreCoordinator` object to specify the detail file that is updated every sync session. Typically, you do this just after you create the `NSPersistentStoreCoordinator` object.

Send `syncWithClient:inBackground:handler:error:` to your `NSPersistentStoreCoordinator` object to begin a sync session, as in this code fragment:

```
[[[self managedObjectContext] persistentStoreCoordinator] syncWithClient:client
 inBackground:NO handler:self error:&error];
```

Optionally, you can sync asynchronously in the background by passing `YES` as the `inBackground:` parameter. Otherwise, you block the user interface while syncing. The advantage of blocking the user interface is that users can't edit managed objects and create conflicts that need to be resolved after syncing.

If you need to process managed objects before the pushing and after the pulling phases of a sync session—for example, convert Core Data data types to Sync Services data types—then read "Controlling Core Data Sync Sessions" (page 90).

Controlling Core Data Sync Sessions

If you register a handler with your `NSPersistentStoreCoordinator` object that conforms to the `NSPersistentStoreCoordinatorSyncing` protocol, then you can intervene during a sync session to control how managed objects are pushed and pulled. This is very similar to how you control a sync session using the `ISyncSessionDriver` class described in "Using a Session Driver" (page 77). The responsibility of the handler is to optionally intervene during the syncing process to perform application-specific tasks.

You set a handler when you start a sync session using the `syncWithClient:inBackground:handler:error:` method as show in "Starting Core Data Sync Sessions" (page 89). All the methods in the `NSPersistentStoreCoordinatorSyncing` protocol are optional, although some are recommended.

If you sync asynchronously, by passing `YES` as the `inBackground:` parameter to the `syncWithClient:inBackground:handler:error:` method, then you should also implement the `managedObjectContextsToMonitorWhenSyncingPersistentStoreCoordinator:` protocol method.

It is recommended that you at least implement the `managedObjectContextsToReloadAfterSyncingPersistentStoreCoordinator:` method to return your persistent store coordinator objects; otherwise, it's your responsibility to reload the persistent store coordinator objects after a sync session. Use the `refreshObject:mergeChanges:` method of `NSManagedObjectContext` to refresh an object.

Use the other methods in the `NSPersistentStoreCoordinator` to modify the behavior of a Core Data application sync session. For example, the StickiesUsingCoreData application implements the `persistentStoreCoordinator:willPushRecord:forManagedObject:inSyncSession:` method to transform the data type of the `color` property of a Note object from an `NSData` to an `NSColor` object before pushing the record. Conversely, StickiesUsingCoreData implements the `persistentStoreCoordinator:willApplyChange:toManagedObject:inSyncSession:` method to transform the data type from an `NSColor` to an `NSData` object before applying the change.

Read *NSPersistentStoreCoordinator Sync Services Additions Reference* and *NSPersistentStoreCoordinatorSyncing Protocol Reference* for details on programmatically controlling aspects of a Core Data sync session.

Syncing Preferences

Users can sync application preferences to multiple computers over MobileMe. By syncing preferences, the user preserves their application settings when moving from computer to computer similar to the advantages of using a network home directory. The user enables this feature, in a way similar to selecting sync schemas, using the MobileMe Sync pane in System Preferences. The user must first configure MobileMe and enable syncing before enabling this feature. For many applications this is a convenient mechanism that requires no source code changes. However, if syncing preferences does not make sense for your application, you can turn this feature off for all or selected application preferences. This article describes several approaches to controlling what application preferences are synced.

Note: Syncing preferences is available in Mac OS X v10.5 and later.

Controlling Syncing Preferences

Set the `com.apple.PreferenceSync.ExcludeAllSyncKeys` key to `true` to exclude all your application preferences from syncing. The default value is `false`—all application preferences are synced.

If you want to exclude just some preferences, set the `com.apple.PreferenceSync.ExcludeSyncKeys` key to an array of keys that you want to exclude from syncing. The keys are represented by string values and are not localized.

You can either use the application information property list, user defaults, or application preferences to set these keys. The defaults for the preference domain is first checked for these settings. If they are not found in the defaults, the application's `info.plist` file is checked. Therefore, if you set these keys programmatically at runtime, you override any values set in the application's `info.plist` file.

You can also use preference domains to exclude some keys from being synced. Use preference domains to specify that some application preferences are host-specific and should not be synced. This is recommended for preferences that are intended for a single computer only. Read Preference Domains in *Preferences Programming Topics for Core Foundation* for more information.

Excluding Preferences Using the Information Property List

Set the `com.apple.PreferenceSync.ExcludeAllSyncKeys` key to `true` in the application's `info.plist` file to exclude all application preferences from syncing as show in Listing 1.

Listing 1 Excluding all preferences using the information property list

```
<plist version="1.0">
<dict>
    ...
```

```
    <key>com.apple.PreferenceSync.ExcludeAllSyncKeys</key>
    <true/>
</dict>
</plist
```

Alternatively, exclude selected keys by setting the `com.apple.PreferenceSync.ExcludeSyncKeys` key to an array of keys in the application's `info.plist` file as show in Listing 2.

Listing 2 Excluding selected preferences using the information property list

```
<plist version="1.0">
<dict>
    ...

    <key>com.apple.PreferenceSync.ExcludeSyncKeys</key>
    <array>
        <string>preferenceX</string>
        <string>preferenceY</string>
    </array>
</dict>
</plist
```

The advantage of setting these keys in the application's `info.plist` file is that your application or an installer doesn't need to run to set these keys.

Excluding Preferences Using Preferences or User Defaults

In addition, you can set these keys programmatically using application preferences or user defaults.

The following code fragment uses the `CFPreferences` opaque type to set the `com.apple.PreferenceSync.ExcludeAllSyncKeys` key to `kCFBooleanTrue`:

```
CFPreferencesSetAppValue(CFSTR("com.apple.PreferenceSync.ExcludeAllSyncKeys"),
 kCFBooleanTrue, kCFPreferencesCurrentApplication);
```

Read *Preferences Programming Topics for Core Foundation* to learn more about the `CFPreferences` opaque type.

The following Objective-C code fragment uses user defaults to exclude all application preferences from syncing by setting the `com.apple.PreferenceSync.ExcludeAllSyncKeys` key to `YES`:

```
[[NSUserDefaults standardUserDefaults] setObject:[NSNumber numberWithBool:YES]
 forKey:@"com.apple.PreferenceSync.ExcludeAllSyncKeys"];
```

Read *User Defaults Programming Topics* to learn more about the `NSUserDefaults` class.

You can also set user defaults from the command line. The following command sets the `com.apple.PreferenceSync.ExcludeSyncKeys` key to exclude the `preferenceX` and `preferenceY` preferences from syncing:

```
defaults write com.apple.myapp com.apple.PreferenceSync.ExcludeSyncKeys -array
 preferenceX preferenceY
```

The disadvantage of using preferences or user defaults is that your application or an installer needs to run to set these keys.

Document Revision History

This table describes the changes to *Sync Services Programming Guide*.

Date	Notes
2009-08-06	Minor edits throughout.
2009-05-28	Updated for Mac OS X v10.6.
2007-10-31	Updated sync schema description to reflect the addition of the UIHelperClass key.
2007-07-11	Updated for Mac OS X v10.5. Added "Using Sync Anchors," "Using a Session Driver," "Syncing Core Data Applications", and "Syncing Preferences" articles to the guide. Fixed miscellaneous errors and typos.
2006-03-08	Fixed client description example.
2006-02-07	Minor edits throughout. Linked methods and constants to API reference.
2005-07-07	Added more information on creating a schema--for example, explains the difference between weak vs. strong ordering and when DataClasses and ExtensionName are required. Otherwise, minor editorial corrections throughout.
2005-04-29	Added description of the Parent, Subtype and EnumType keys to the schema description.
	Added content on filtering, schema extensions, and localization of data alert panels. Minor editorial corrections throughout. First public version.
	Added the "Filtering Records" (page 71) article, "Creating a Schema Extension" (page 51) section in the "Creating a Sync Schema" (page 39)article, and details on alert messages to the "Localizing Property Names" (page 50) section in "Creating a Sync Schema" (page 39).
2004-11-02	Changed the title from *Sync Services* to be consistent with the titles of similar documentation. Added "Formatting Records" (page 75), updated "Creating a Sync Schema" (page 39) per schema, attribute types and localization changes, added and updated code samples throughout. Refined some of the API descriptions.
2004-08-12	Added these new articles: "Syncing Relationships" (page 61) and "Syncing with Other Clients" (page 65). Added more details and examples to these articles: "Creating a Sync Schema" (page 39) (formerly called "Schema Design"), and "Registering Clients" (page 57). Made minor edits to these articles: "Sync Services Overview" (page 17), "Managing Your Sync Session" (page 25) and "Registering Schemas" (page 55).

Date	Notes
2004-06-29	First release of conceptual and task material covering Sync Services framework available in Mac OS X 10.4 and later.