
Calendar Store Programming Guide

Data Management: Calendar Data



2009-07-28



Apple Inc.
© 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Cocoa, iCal, Mac, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR

CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Calendar Store Programming Guide 7

- Who Should Read This Document? 7
- Organization of This Document 7
- See Also 8

Calendar Store Overview 9

- Calendar Store Architecture 9
- Calendar Store Objects 10
- Using Predicates 11
- Updating Your Objects 11
- Fetching in Batches 12

Fetching Objects 13

- Fetching Calendars 13
- Fetching Events 13
 - Fetching Individual Events 14
 - Using Predicates to Fetch Events 14
 - Fetching Recurring Events 14
- Fetching Tasks 15
 - Fetching Individual Tasks 15
 - Using Predicates to Fetch Tasks 15

Creating Objects 17

- Creating Calendars 17
- Creating Events 17
- Creating Tasks 18
- Creating Alarms 18

Creating Recurring Events 19

- Recurrence Rule Classes and Methods 19
- Specifying an Interval 20
- Creating a Recurrence End 20
- Creating Daily Recurrence Rules 21
- Creating Weekly Recurrence Rules 21
- Creating Monthly Recurrence Rules 22
- Creating Yearly Recurrence Rules 22

Saving Changes 25

Saving Changes to Calendars 25

Saving Changes to Events 25

Saving Changes to Tasks 26

Observing Changes 27

Observing External Notifications 27

Observing Internal Notifications 27

Applying Changes 28

Observing User Changes Using Cocoa Bindings 29

Document Revision History 31

Figures, Tables, and Listings

Calendar Store Overview 9

Figure 1	Calendar Store architecture	9
Figure 2	CalEvent object diagram	10
Figure 3	CalTask object diagram	11

Fetching Objects 13

Listing 1	Fetching Current Year Events	14
-----------	------------------------------	----

Creating Recurring Events 19

Table 1	Initialization method arguments	20
---------	---------------------------------	----

Saving Changes 25


Listing 1	Saving events	25
-----------	---------------	----

Observing Changes 27

Listing 1	Applying changes	28
-----------	------------------	----

Introduction to Calendar Store Programming Guide

Calendar Store is a framework that allows Cocoa applications to access iCal data. You can fetch iCal records—such as calendars, events, and tasks—and receive notifications when these records change in iCal. You can also make some local changes to records and save them to the Calendar Store database. This document describes Calendar Store concepts and common programming tasks.

 **Warning:** If your application uses the Sync Services and Calendar Store frameworks together, then you should not use Sync Services to sync data shared with the Calendar Store framework. The Calendar Store framework already syncs its records with Sync Services, so applications sharing the Calendar Store 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 Calendar Store framework.

Who Should Read This Document?

You should read this document if you want to display or edit iCal data in your application. Calendar Store is ideal for integrating subsets of iCal data into your application. Calendar Store simplifies fetching and saving changes to records since you don't have to implement your own persistent storage or deal with the complexity of the Calendars schema. Calendar Store also notifies applications of changes made in iCal so your application data stays fresh. It is suitable for developing widgets, plug-ins, and augmenting other types of applications that use calendar data. It is not suitable for implementing full-featured calendar applications.

Note: Calendar Store does not provide complete read/write database access to all iCal records. Use Sync Services and the Calendars schema if you need access to every calendar entity and property.

Concurrency Note: The Calendar Store framework is thread safe. No additional locking or other synchronization is required. The framework handles synchronization for you.

Organization of This Document

You should read these articles if you just want to fetch Calendar Store objects:

- [“Calendar Store Overview”](#) (page 9) describes the Calendar Store architecture and core classes.
- [“Fetching Objects”](#) (page 13) explains how to fetch calendar, event, and task objects.
- [“Observing Changes”](#) (page 27) explains how to observe changes made to these objects by other processes.

You should also read these articles if you want to create or modify Calendar Store objects:

- [“Creating Objects”](#) (page 17) explains how to create commonly used objects: calendars, events, tasks, and alarms.
- [“Creating Recurring Events”](#) (page 19) explains how to create recurring events—events that repeat according to a custom pattern.
- [“Saving Changes”](#) (page 25) explains how to save changes you make locally to Calendar Store objects.

See Also

For an in-depth description of the Calendar Store API, read:

- *Calendar Store Framework Reference*

The following projects contain more sample code:

- *Checklist*
- *SimpleCalendar*

If you decide to use Sync Services to access records in the Calendars schema directly, read:

- *Apple Applications Schema Reference*
- *Sync Services Programming Guide*
- *Sync Services Framework Reference*

Calendar Store Overview

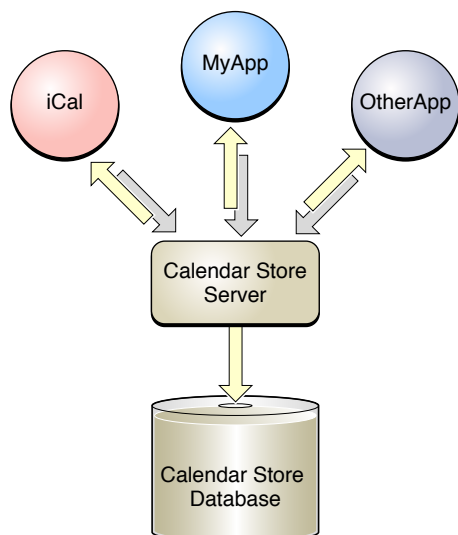
The goal of Calendar Store is to provide robust and reliable access to some iCal data for applications that want to integrate iCal features—such as displaying events or tasks—but don't need read/write access to every record used by iCal. For this reason, Calendar Store only allows you to programmatically create objects that you can create using the iCal controls—for example, using this framework you can not create more sophisticated recurring events than you can using iCal. Calendar Store greatly simplifies the tasks of fetching, updating, and saving records—you do not need to implement your own persistent storage to use Calendar Store.

Calendar Store Architecture

Applications that use the Calendar Store framework have the ability to fetch and save a subset of the records used by iCal. Your application, as well as iCal, are clients of the Calendar Store server as depicted in Figure 1.

There is one Calendar Store server and Calendar Store database for each user on each computer. The Calendar Store database stores the local copies of records belonging to the Calendars schema. If .Mac is configured to sync the Calendars schema, then your changes to these records, using the Calendar Store programming interface, are also synced. Read *Sync Services Programming Guide* for more information on syncing over .Mac.

Figure 1 Calendar Store architecture



Calendar Store Objects

The Calendar Store database stores records, but the Calendar Store programming interface returns objects to your application. These objects hide the complexity of the Calendars schema and encapsulate a common subset of the data useful to most applications. The primary objects you fetch from Calendar Store are calendars, events, and tasks.

Figure 2 depicts the relationships between an event object, an instance of `CalEvent`, and other objects. An event object has a to-one relationship to its calendar and a to-many relationship to its attendees and alarms. Attendees are instances of `CalAttendee` that may correspond to a person in the Address Book.

The other classes in the diagram help describe the recurrence rule for recurring events—for example, an event that occurs every Tuesday and Thursday of the week for the next two months. A `CalRecurrenceEnd` object describes how a recurring event ends, and a `CalNthWeekDay` object helps describe the recurring pattern.

Figure 2 CalEvent object diagram

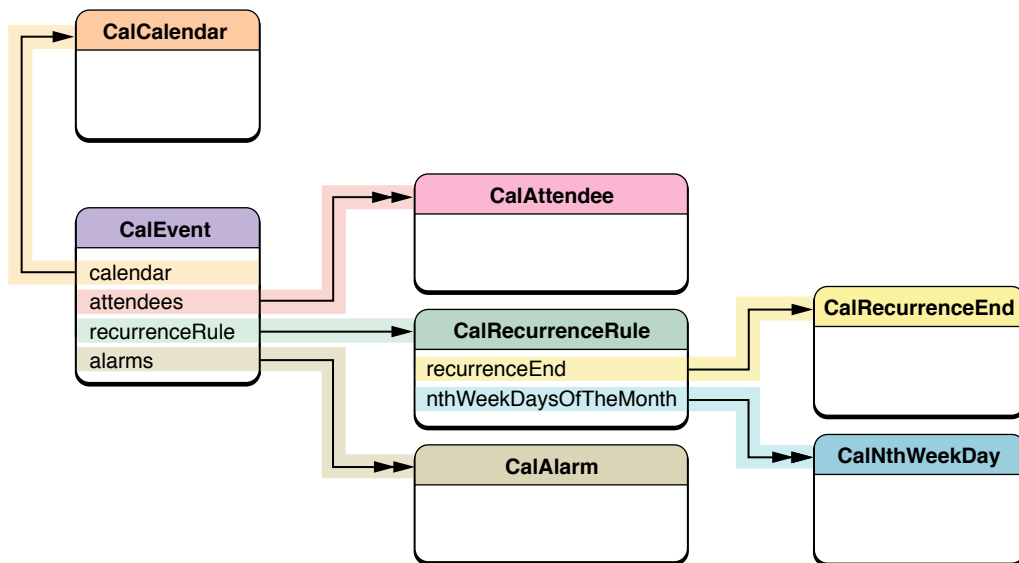


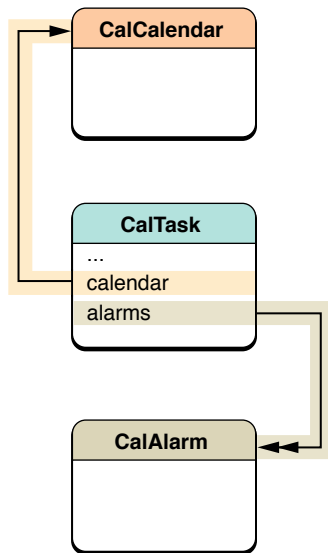
Figure 3 depicts the relationships between a task object, an instance of `CalTask`, and other objects. A task object is much simpler than an event object. A task has just a to-one relationship to a calendar and a to-many relationship to its alarms.

Both events and tasks can have multiple alarms. The `CalAlarm` class encapsulates information on the type of alarm and how it is triggered.

Notice that events and tasks have a to-one relationship to a calendar, but there is not an inverse to-many relationship from a calendar to its events and tasks. This is purposely done to make fetching objects more efficient. Fetching a calendar object does not automatically fetch its events and tasks. This would be grossly inefficient, and in some cases impossible, because recurring events can result in an infinite number of event objects—recurrences are represented by multiple event objects with the same unique identifier (UID).

Instead you fetch the calendars, events, and tasks separately by sending messages to a shared `CalCalendarStore` object. The `CalCalendarStore` object is a direct interface to the Calendar Store database. The `CalCalendarStore` class provides many convenience methods for fetching just the calendars, events, and tasks your application needs.

Figure 3 CalTask object diagram



Using Predicates

A predicate is an object used to define logical conditions to constrain a search or to filter objects. In database terminology, a predicate is equivalent to a query.

Predicates provide the maximum flexibility in specifying a subset of the objects you want to fetch. For example, using predicates, you can fetch events that occur on a single day, month, or year. Or specify an exact date/time range. You can also fetch events that belong to particular calendars. Similarly, you can fetch tasks that have specific completion dates or all tasks that have been completed by a specific date.

The `CalCalendarStore` class makes it simple to use predicates by providing convenience methods for creating common queries. Read “Fetching Objects” for more information on the different ways to fetch objects.

Updating Your Objects

When designing your application, you need to decide if you are going to retain the objects fetched by Calendar Store or use the data you obtain and release the objects. Note that Calendar Store doesn’t automatically update objects that were previously fetched. Instead you register for notifications when Calendar Store objects change internally or externally—for example, when the user changes an event in iCal. Then implement your handler to apply the external changes to your local objects.

If you do not retain the objects you fetch, then you should at least retain the unique identifier for each object. The notification object user information dictionary contains the unique identifier for the object that was either added, updated, or deleted. You can use the unique identifier to find the corresponding local object and apply the change.

Read [“Observing Changes”](#) (page 27) for more information on updating objects, including tips on using Cocoa bindings.

Fetching in Batches

As stated above, for performance reasons, you cannot simply fetch all events. Recurring events are represented by multiple event objects, which are infinite in quantity if the recurring event never ends. Therefore, Calendar Store limits the time span for fetching recurring events to just four years. For example, if you define a predicate that fetches all events for a ten year span, the shared `CalCalendarStore` object fetches only recurring events for the first four years.

Because of this, you typically design your application to fetch events in batches. Use predicates to define a custom fetch that is efficient for your application. Basically, just fetch the events that your application needs at the moment. For example, fetch events for the current month and fetch events for the next month only when the user clicks the Next button.

Fetching Objects

Calendar Store provides several methods for fetching objects—such as calendars, events, and tasks—from the Calendar Store database. You can fetch all calendars or specific ones using a unique identifier. You can fetch events and tasks specifying a date range or use a predicate to fetch a custom set of objects. However, you cannot fetch all events because recurring events may result in an infinite number of event objects—there are multiple event objects for a recurring “master” event. Typically, you fetch objects in batches as your application uses them—for example, fetch the next month or year of events when the user clicks the Next button. This article describes the methods you use to fetch these Calendar Store objects.

Note: There is one Calendar Store database for each user on each computer represented by a shared `CalCalendarStore` object. Use the `defaultCalendarStore` method to get the shared `CalCalendarStore` object.

Fetching Calendars

You can fetch all calendars in the Calendar Store database or specific calendars using a unique identifier called a UID.

Use the `calendars` method if you want to fetch all calendars. Note that there is a to-one relationship from an event or task to a calendar object. However, there is not an inverse to-many relationship from a calendar to its events and tasks. Therefore, when you fetch calendars, you are just creating instances of the `CalCalendar` class. Read [“Fetching Events”](#) (page 13) and [“Fetching Tasks”](#) (page 15) for how to fetch other types of objects.

A calendar object’s UID is guaranteed to be unique for the lifetime of the calendar object. So you may retain the UID and use it to fetch individual calendar objects on an as needed basis. Use the `calendarWithUID:` method to fetch a calendar object using a UID.

Fetching Events

You can fetch individual events specifying a UID—for example, if you retain an event UID from a previous fetch. However, you typically fetch events using a predicate, which provides the maximum flexibility. You essentially create a database query specifying the property values or range of property values you want the results to match. For your convenience, Calendar Store provides methods for common types of fetches. The following sections progress from the most simple to the most complex fetches.

Fetching Individual Events

To fetch an individual event, use the `eventWithUID:occurrence:` method, specifying a UID. If the event is recurring, specify an occurrence date of the individual event you want—recurring events have the same UID. If you don't know the UID for an event, read [“Using Predicates to Fetch Events”](#) (page 14) for how to fetch events using a predicate.

Using Predicates to Fetch Events

It's very common to fetch events in a date range. For example, fetch all events that occur in the current month and then fetch all events that occur in the previous and next months. The `CalCalendarStore` class provides convenience methods for creating these types of predicates that can be passed to the `eventsWithPredicate:` method to perform the actual fetch.

For example, the following code fragment in Listing 1 uses the `eventPredicateWithStartDate:endDate:calendars:` class method to create an event predicate for the current year, and the `eventsWithPredicate:` method for fetching the events.

Listing 1 Fetching Current Year Events

```
// Create a predicate to fetch all events for this year
NSInteger year = [[NSDate date] yearOfCommonEra];
NSDate *startDate = [[NSDate dateWithYear:year month:1 day:1 hour:0
minute:0 second:0 timeZone:nil] retain];
NSDate *endDate = [[NSDate dateWithYear:year month:12 day:31 hour:23
minute:59 second:59 timeZone:nil] retain];
NSPredicate *eventsForThisYear = [CalCalendarStore
eventPredicateWithStartDate:startDate endDate:endDate
calendars:[CalCalendarStore defaultCalendarStore] calendars]];

// Fetch all events for this year
NSArray *events = [CalCalendarStore defaultCalendarStore]
eventsWithPredicate:eventsForThisYear];
```

Note: Currently, the `eventsWithPredicate:` method only supports predicates created using either the `eventPredicateWithStartDate:endDate:calendars:` or `eventPredicateWithStartDate:endDate:UID:calendars:` class method.

Fetching Recurring Events

The Calendar Store framework doesn't allow you to fetch all events because recurring events may result in an infinite number of event objects.

You can use an event predicate to fetch all events belonging to the same recurrence within a date range. Recurring events have the same UID, so use the `eventPredicateWithStartDate:endDate:UID:calendars:` method similar to the `eventPredicateWithStartDate:endDate:calendars:` method of Listing 1, except specify the common UID. The events returned include all events within the date range belonging to the same recurrence, or a single event if it's not a recurring event.

As explained in “[Fetching in Batches](#)” (page 12), the events returned by the `eventsWithPredicate:` method contain only recurring events for the first four years of a date range. For this reason, you typically fetch events in batches—no greater than a four year span—as you need them in your application.

Fetching Tasks

Similar to events, you can fetch individual tasks by specifying a UID or multiple tasks by using a predicate. Again, fetching tasks using a predicate provides the maximum flexibility. You essentially create a database query specifying the property values or range of property values you want the results to match. Similar to creating event predicates, Calendar Store provides convenience methods for creating common task predicates.

Fetching Individual Tasks

To fetch an individual task, use the `taskWithUID:` method, specifying a UID. If you don't know the UID for a task, read “[Using Predicates to Fetch Tasks](#)” for how to fetch tasks with specific criteria.

Using Predicates to Fetch Tasks

You can create a predicate to fetch all tasks, fetch all incomplete tasks, fetch all incomplete tasks due before a specific date, or fetch all completed tasks since a specified date. In all cases, you can specify what calendars the tasks must belong to.

For example, this code fragment creates a predicate using the `taskPredicateWithCalendars:` method to fetch all tasks in all calendars:

```
NSPredicate *predicate = [CalCalendarStore
taskPredicateWithCalendars:[CalCalendarStore defaultCalendarStore] calendars];
NSArray *tasks = [[CalCalendarStore defaultCalendarStore]
tasksWithPredicate:predicate];
```

Similarly, use the `taskPredicateWithTasksCompletedSince:calendars:`, `taskPredicateWithUncompletedTasks:`, and `taskPredicateWithUncompletedTasksDueBefore:calendars:` methods to create predicates to perform other types of fetches.

Creating Objects

Your application can also create Calendar Store objects—such as calendars, events, and tasks—that are visible in iCal. Each type of object has properties you set that are either optional or mandatory—for example, events and tasks have a title and calendar property. Some properties you do not set—they are read-only. This article describes how to create each type of object.

Note that none of the changes you make to Calendar Store objects, including creating new objects, persist unless you save the object using one of the `save...` methods in `CalCalendarStore`. For example, if you create a `CalEvent` object, then use the `saveEvent:span:error:` method of `CalCalendarStore` to save it. Read [“Saving Changes”](#) (page 25) for more information on saving Calendar Store objects.

Event and task objects inherit common properties from the `CalCalendarItem` class. Note that you must set the `calendar` property of items before you attempt to save them. (Read [“Fetching Calendars”](#) (page 13) for how to fetch calendar objects that you can use to set the `calendar` property.) All other properties defined in the `CalCalendarItem` class are either optional or read-only. The `dateStamp` and `uid` properties are read-only. All other calendar item properties—such as `notes`, `url`, `title`, and `alarms`—are optional.

Creating Calendars

You create a calendar object using the `calendar` class method of `CalCalendar`. Optionally, you can then set the `color`, `notes`, or `title` properties. Currently, you can create iCal calendars only. For example, the following code fragment creates a new calendar titled “Kids”:

```
// Create an iCal calendar
CalCalendar *kidsCalendar = [CalCalendar calendar];
kidsCalendar.title = @"Kids";
```

Creating Events

You create a `CalEvent` object using the `event` class method of `CalCalendar`, and, typically, set the `calendar`, `startDate`, `endDate`, and `title` properties. For example, the following code fragment creates a one hour appointment:

```
// Create a simple event.
CalEvent *event = [CalEvent event];
event.calendar = calendar;
event.title = @"Dentist";
event.startDate = [NSDate dateWithNaturalLanguageString:@"3pm January 9, 2007"];
event.endDate = [NSDate dateWithNaturalLanguageString:@"4pm January 9, 2007"];
event.location = @"1123 Fremont Avenue";
```

Read [“Creating Recurring Events”](#) (page 19) for how to create events that repeat according to a specified daily, weekly, monthly, or yearly pattern.

Creating Tasks

You create a task object using the `task` `CalTask` class method, and typically, set the `calendar`, `title`, `dueDate`, and `priority` properties. For example, the following code fragment creates a task:

```
// Create a simple task.
CalTask *task = [CalTask task];
task.calendar = calendar;
task.title = @"File Tax Return";
task.dueDate = [NSDate dateWithNaturalLanguageString:@"12am April 15, 2007"];
task.priority = CalPriorityHigh;
```

Note that the `isCompleted` and `completedDate` properties are interdependent. If you set one of these properties, the value of the other property changes too. Read *CalTask Class Reference* for more details.

Creating Alarms

You can add alarms to both event and task objects. You create an alarm object using the `alarm` `CalAlarm` class method and then set the `action` property.

If you set the `action` property to `CalAlarmActionEmail`, set the `emailAddress` property to the email address that is sent a notification when the alarm triggers. If you set the `action` property to `CalAlarmActionSound`, set the `sound` property to the sound that is played when the alarm triggers. If you set the `action` property to `CalAlarmActionProcedure`, set the `URL` property to the file that opens when the alarm triggers.

Optionally, you can specify a relative or an absolute trigger. Only one of these is active at a time. If you set the `relativeTrigger` property, then the `absoluteTrigger` property is set to `nil`, and vice versa. Use the `relativeTrigger` property if you want the alarm to trigger a specified interval before the task `dueDate` or event `startDate` properties. For example, you might want the alarm to trigger 15 minutes or 24 hours before a task is due. Use the `absoluteTrigger` property if you want the alarm to trigger at a precise time on a specified date.

For example, the following code fragment adds an alarm to a task that triggers 15 minutes before the due date:

```
// Add an alarm to a task.
CalAlarm *alarm = [CalAlarm alarm];
alarm.action = CalAlarmActionSound;
alarm.sound = @"Basso";
alarm.relativeTrigger = -15*60;
[task addAlarm:alarm];
```

The `addAlarm:` method is inherited from the `CalCalendarItem` class along with other methods for adding and removing alarms. Note that alarms you add to events and tasks are automatically saved when the events and tasks are saved.

Creating Recurring Events

Recurring events are events that repeat daily, weekly, monthly, or yearly. Actually, the pattern that repeats can be quite complex, such as every Tuesday and Thursday of the first and second week of every month of the year.

Typically, you create an event by sending `event` to the `CalEvent` class. Then you set the `startDate` and optionally the `endDate` properties of the new event object returned by this method. Read [“Creating Events”](#) (page 17) for how to create a basic event object. If you are creating a recurring event, you also need to set the `recurrenceRule` property. The recurrence rule is the object that describes the recurring pattern.

This article describes how to create recurrence rules.

Recurrence Rule Classes and Methods

A recurrence rule defines a time/date pattern that repeats given some interval. A combination of classes are used to represent a recurrence rule including `CalRecurrenceRule`, `CalRecurrenceEnd`, and `CalNthWeekDay`. [Figure 2](#) (page 10) in [“Calendar Store Overview”](#) (page 9) depicts the relationship between these classes and an event.

The `CalRecurrenceRule` class is used to describe the recurrence rule—the pattern that repeats—for a recurring event. The patterns you can create are limited to the types of recurring events the user can create in iCal. `CalRecurrenceRule` provides convenience methods for creating different types of recurrence rules.

The convenience methods are initialization methods that you send to a newly created `CalRecurrenceRule` object. The initialization methods set all the properties of a `CalRecurrenceRule` object—you cannot set the properties directly since they are read-only.

Each initialization method creates a particular type of recurrence rule that uses a time unit of measurements between intervals. You can create daily, weekly, monthly, or yearly recurrence rules. If the initialization method begins with `initMonthly...`, then the object returned is a monthly recurrence rule that uses a single month as the unit of time between intervals. Similarly, use `initDaily...`, `initWeekly...`, and `initYearly...` methods to create daily, weekly, and yearly recurrence rules respectively. Use the `recurrenceType` property of a `CalRecurrenceRule` to determine the type.

Each of the initialization methods has an `...Interval` and an `end:` argument described in [Table 1](#). The `end` rule is a `CalRecurrenceEnd` object that specifies how a recurring event ends. The other arguments that may appear in the different initialization methods are used to create more complex recurrence rules—for example, they specify the day of the week or week of the month in the pattern. [Table 1](#) describes the values of these additional arguments.

The `CalNthWeekDay` class depicted in [Figure 2](#) (page 10) is used to get information about a recurring event, not create one. The `nthWeekDaysOfTheMonth` property of a `CalRecurrenceEnd` object is an array of `CalNthWeekDay` objects. See [CalNthWeekDay Class Reference](#) for more information about this class.

Table 1 Initialization method arguments

Argument	Value	Description
<code>...Interval</code>	An integer greater than 0.	The number of time units between occurrences of an event where time units are a day, week, month, or year depending on the <code>init...</code> method used.
<code>forDayOfTheWeek:</code>	An integer ranging from 1 to 7 where Sunday is 1.	The day of the week when an event occurs.
<code>forDaysOfTheWeek:</code>	An array of integers ranging from 1 to 7 where Sunday is 1.	The days of the week when an event occurs.
<code>forWeekOfTheMonth</code>	An integer value of either 1, 2, 3, 4, or -1 where -1 is the last week of the month.	The week of the month when an event occurs.
<code>forDaysOfTheMonth</code>	An array of integers ranging from 1 to 31.	The day of the month when an event occurs.
<code>forMonthsOfTheYear</code>	An array of integers ranging from 1 to 12.	The day of the month when an event occurs.
<code>end:</code>	A <code>CalRecurrenceEnd</code> object.	Specifies how a recurring event ends.

Specifying an Interval

The interval argument is an integer greater than 0 that specifies how often a pattern repeats given the recurrence end's unit of time. For example, if the time unit is a week and the interval is 1, then the pattern repeats every week. If the time unit is a month and the interval is 2, then the pattern repeats bimonthly. If the time unit is a year and the interval is 3, then the pattern repeats every three years. The convenience method you use to create the recurrence rule determines the time unit.

Creating a Recurrence End

The recurrence end argument is optional and specifies how the recurring event ends. The recurrence end argument is a `CalRecurrenceEnd` object that specifies either a counter or an ending date. If the recurrence end object uses a counter, the event ends after the counter decrements to 0. Otherwise, it ends on the specified date.

Use the `recurrenceEndWithOccurrenceCount: CalRecurrenceEnd` method to create a recurrence end that uses a counter, or the `recurrenceEndWithEndDate: CalRecurrenceEnd` method to create a recurrence end that uses an end date.

Creating Daily Recurrence Rules

The simplest and most common recurring event is one that occurs in daily intervals—for example, an event that occurs every day or an event that occurs every two days.

Use the `initWithDailyRecurrenceWithInterval:end:` method to create an event that occurs once a day and repeats in daily intervals. Use the `...Interval:` argument to specify the number of days between occurrences (see “Specifying an Interval”) and optionally, use the `end:` argument to specify when the event ends (see “Creating a Recurrence End”).

For example, the following code fragment creates a never-ending event that occurs every other day starting today:

```
// Create a daily event that occurs every other day.
CalEvent *event = [CalEvent event];
event.calendar = calendar;
event.title = @"Bath Nite";
event.startDate = [NSDate date];
event.endDate = [NSDate distantFuture];
event.recurrenceRule = [[[CalRecurrenceRule alloc]
initWithDailyRecurrenceWithInterval:2 end:nil] autorelease];
```

Creating Weekly Recurrence Rules

Weekly recurrence rules are useful for events that occur on the same days of the week—for example, a class that meets at the same time on Mondays, Wednesdays, and Fridays of every week.

Similar to creating a daily recurrence rule, use the `initWithWeeklyRecurrenceWithInterval:end:` method to create an event that occurs once a week and repeats in weekly intervals. The day of the week defaults to the day of the week of the start date. If the start date is a Tuesday, the weekly event repeats on Tuesdays at the specified interval. For example, use this method to create an event that occurs every-other week starting today as in the following code fragment:

```
// Create a weekly event that occurs every other week from the start date.
CalEvent *event = [CalEvent event];
event.calendar = calendar;
event.title = @"Pay Day";
event.startDate = [NSDate date];
event.endDate = [NSDate distantFuture];
event.recurrenceRule = [[[CalRecurrenceRule alloc]
initWithWeeklyRecurrenceWithInterval:2 end:nil] autorelease];
```

Use the `initWithWeeklyRecurrenceWithInterval:forDaysOfTheWeek:end:` method to create an event that occurs multiple times per week and repeats in weekly intervals. Use the `forDaysOfTheWeek:` argument to specify which days of the week the event occurs. This argument is an array of integers ranging from 1 to 7 where Sunday is equal to 1. For example, the following code fragment creates an event that occurs every 3rd week on Mondays and Wednesdays starting today:

```
CalEvent *event = [CalEvent event];
event.calendar = calendar;
event.title = @"3rd Week MW";
event.startDate = [NSDate date];
```

```

event.endDate = [NSDate distantFuture];
NSArray *days = [NSArray arrayWithObjects:[NSNumber numberWithInt:2], [NSNumber
    numberWithInt:4], nil];
event.recurrenceRule = [[[CalRecurrenceRule alloc]
    initWithWeeklyRecurrenceWithInterval:3 forDaysOfTheWeek:days end:nil] autorelease];

```

Note that the event occurs on the given start date regardless of whether it matches the weekly pattern. For example, if the start date is a Tuesday and the days of the weekly pattern are Mondays and Wednesdays, then the event occurs on the Tuesday start date and thereafter, Mondays and Wednesdays at the specified interval.

Creating Monthly Recurrence Rules

Similar to weekly recurrence rules, monthly recurrence rules are useful for creating patterns that repeat in monthly intervals.

Use the `initWithMonthlyRecurrenceWithInterval:end:` method to create an event that occurs once a month and repeats in monthly intervals. For example, use this method to create an event that occurs on the 15th of every month for the next 6 months.

```

// Create a monthly event that occurs on the 15th of every month for 6 months.
CalEvent *event = [CalEvent event];
event.calendar = calendar;
event.title = @"Loan Payment";
event.startDate = [NSDate dateWithNaturalLanguageString:@"01/15/07"];
event.endDate = [NSDate dateWithNaturalLanguageString:@"06/15/07"];
CalRecurrenceEnd *end = [CalRecurrenceEnd recurrenceEndWithOccurrenceCount:6];
event.recurrenceRule = [[[CalRecurrenceRule alloc]
    initWithMonthlyRecurrenceWithInterval:1 end:end] autorelease];

```

Use the `initWithMonthlyRecurrenceWithInterval:forDaysOfTheMonth:end:` method to create an event that occurs multiple times a month and repeats in monthly intervals. Use the `forDaysOfTheMonth:` argument to specify the days of the month that the event occurs. This argument is an array of integers ranging from 1 to 31 representing the days of the month. For example, use this method to create an event that occurs on the 1st and 15th of every month.

Use the `initWithMonthlyRecurrenceWithInterval:forDayOfTheWeek:forWeekOfTheMonth:end:` method to create an event that has both a weekly and monthly pattern that repeats in monthly intervals. Use the `forDayOfTheWeek:` argument to specify the day of the week that the pattern occurs. Use the `forWeekOfTheMonth:` argument to specify the weeks within a month that the pattern occurs. See [Table 1](#) (page 20) for details on these arguments.

Creating Yearly Recurrence Rules

Yearly recurrence rules are needed to create more complex recurring events. For example, events that occur quarterly or twice a year on the same month and day of the year. Even some complex monthly patterns can be easier to represent using a yearly recurrence rule.

Use the `initWithYearlyRecurrenceWithInterval:end:` method to create an event that occurs once a year and repeats in a yearly interval. For example, if the start date is July 4th, 2006 and the interval is 1, then the event would occur every 4th of July after the start date.

Use the `initWithYearlyRecurrenceWithInterval:forMonthsOfTheYear:end:` method to create an event that occurs multiple times a year and repeats in yearly intervals. For example, if the start date is April 10, 2007, the interval is 1, the `forMonthsOfTheYear:` argument is an array containing the integers 4 and 12, then this event occurs on April 10th and December 10th of every year after 2005 as shown in this code fragment:

```
// Create a yearly event that occurs on the 10th of April and December of every
year.
CalEvent *event = [CalEvent event];
event.calendar = calendar;
event.title = @"Property Taxes";
event.startDate = [NSDate dateWithNaturalLanguageString:@"April 10, 2007"];
event.endDate = event.endDate = [NSDate distantFuture];
NSArray *months = [NSArray arrayWithObjects:[NSNumber numberWithInt:4], [NSNumber
    numberWithInt:12], nil];
event.recurrenceRule = [[[CalRecurrenceRule alloc]
    initWithYearlyRecurrenceWithInterval:1 forMonthsOfTheYear:months end:nil]
    autorelease];
```

Use the

`initWithYearlyRecurrenceWithInterval:forDayOfTheWeek:forWeekOfTheMonth:forMonthsOfTheYear:end:` method to create an event that has a predictable weekly, monthly, and yearly pattern.

Saving Changes

Changes you make locally to calendar objects are not persistent until you save them to the Calendar Store database. This includes calendar, event, and task objects that you create locally. For example, instantiating a `CalEvent` object doesn't automatically add it and save it to the database. This article describes the `CalCalendarStore` methods you use to save each type of object.

Although errors are rare, you should always check the return value of the `save... CalCalendarStore` methods and if an error occurs, take the appropriate action. For example, the sample code in this article displays an alert panel when an error occurs.

Saving Changes to Calendars

Use the `saveCalendar:error: CalCalendarStore` method to save a new `CalCalendar` object or to save changes to an existing `CalCalendar` object. Note that changes to events or tasks belonging to a calendar are not automatically saved when you save the calendar. Read [“Saving Changes to Events”](#) (page 25) and [“Saving Changes to Tasks”](#) (page 26) for how to save events and tasks.

Saving Changes to Events

Use the `saveEvent:span:error: CalCalendarStore` method to save a new `CalEvent` object or to save changes to an existing `CalEvent` object. If you are adding a new object, the `calendar` property of the `CalEvent` object needs to be set before you invoke this method.

The code fragment in Listing 1 shows how to save changes to an event object.

Listing 1 Saving events

```
// Save changes to an event
NSError *calError;
if ([[CalCalendarStore defaultCalendarStore] saveEvent:event span:CalSpanThisEvent
    error:&calError] == NO){
    UIAlertView *alertPanel = [UIAlertView alertWithError:calError];
    (void) [alertPanel runModal];
    // terminate the application?
}
```

Similarly, use the `removeEvent:span:error: CalCalendarStore` method to delete an event from the Calendar Store database. This method returns YES if successful, and NO if an error occurred.

You use the `span:` argument for each of these methods to specify which events of a recurring event to apply the changes to. Pass the `CalSpanThisEvent` constant for a nonrecurring event. Otherwise, use the `CalSpanFutureEvents` constant to apply the changes to all future events or the `CalSpanAllEvents` constant to apply the changes to all events in the recurrence.

Saving Changes to Tasks

Use the `saveTask:error:CalCalendarStore` method to save a new `CalTask` object or to save changes to an existing `CalTask` object. If you are adding a new object, the `calendar` property of the `CalTask` object needs to be set before you invoke this method.

Use the `removeTask:error:CalCalendarStore` method to remove a task from the Calendar Store database. Again, check the return value of this method in case an error occurs.

Observing Changes

If you fetch Calendar Store objects, you typically want to observe changes to the objects so your application data is in sync with the Calendar Store database. You especially need to do this if you retain the calendar objects or display information about the objects to the user. For example, if an event's start date changes in iCal, you might want to update the view of that event in your application too. Similarly, if the user makes changes to an event in your application, you might want to make the change to the Calendar Store database so that iCal updates its display. To do this, you need to observe changes made externally and internally. How you observe end-user changes is application dependent. This article demonstrates how to do this using Cocoa bindings.

Observing External Notifications

`CalCalendarStore` defines several notifications that are posted when another process changes fetched objects. A separate notification is posted for each type of object (calendars, events, and tasks) and contains information about changes to multiple objects—it does not post a notification for each individual change. Similar to Core Data change notifications, the notification user dictionary contains information about what objects were inserted, updated, or deleted. The handlers for these notifications should update the local object to reflect the changes.

If you are displaying information about calendars, then observe the `CalCalendarsChangedExternallyNotification` notification. If you are displaying information about events, then observe the `CalEventsChangedExternallyNotification` notification. If you are displaying information about tasks, then observe the `CalTasksChangedExternallyNotification` notification.

For example, the following code fragment registers for external event changes:

```
[[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(eventsChanged:)
 name:CalEventsChangedExternallyNotification object:[CalCalendarStore
 defaultCalendarStore]];
```

Observing Internal Notifications

You can also observe changes made specifically by your application. For example, update the display when you change calendar objects internally. The internal notifications are `CalCalendarsChangedNotification`, `CalEventsChangedNotification`, and `CalTasksChangedNotification`.

For example, the following code fragment registers for internal event changes:

```
[[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(eventsChanged:)
 name:CalEventsChangedNotification object:[CalCalendarStore
 defaultCalendarStore]];
```

Applying Changes

The notification handler method should check to see what objects need to be deleted, inserted, or updated as shown in Listing 1. Since how you apply changes to local objects is application dependent, those portions of this method implementation are omitted in the listing.

Listing 1 Applying changes

```
- (void)eventsChanged:(NSNotification *)notification
{
    // Apply delete changes
    NSArray *deletedRecords = [[notification userInfo]
valueForKey:CalDeletedRecordsKey];
    if (deletedRecords != nil){
        // insert code that deletes local objects
    }

    // Apply insert changes
    NSArray *insertedRecords = [[notification userInfo]
valueForKey:CalInsertedRecordsKey];
    if (insertedRecords != nil){
        // insert code that adds new local objects
    }

    // Apply update changes
    NSArray *updatedRecords = [[notification userInfo]
valueForKey:CalUpdatedRecordsKey];
    if (updatedRecords != nil){
        // insert code that updates existing objects
    }

    return;
}
```

Note that the values for the keys contained in the user information dictionary—`CalInsertedRecordsKey`, `CalDeletedRecordsKey`, and `CalUpdatedRecordsKey`—are arrays of UIDs, not arrays of `CalCalendar`, `CalEvent`, or `CalTask` objects. Therefore, you need to retain either the corresponding `CalCalendar`, `CalEvent`, or `CalTask` objects when you originally fetched them, or retain their UIDs to match the remote changes with the local representation of these objects.

When inserting or updating objects, use the corresponding `calendarWithUID:`, `eventWithUID:occurrence:`, or `taskWithUID:` method to fetch the inserted or updated objects from the Calendar Store database.

If you are updating an object, you can either compare property values between the new and old object, or replace the old object with the new object. Calendar Store does not automatically update existing calendar objects—for example, it does not update a previously fetched `CalEvent` object. If you retain `CalEvent` objects—for example, add them to an array—then you typically replace the old `CalEvent` object with the new `CalEvent` object to apply an update.

For this reason, you should take precautions when using calendar objects with Cocoa bindings. For example, implement your own model object that uses the calendar object as an internal representation or release calendar objects after retaining the UID and other property values.

Observing User Changes Using Cocoa Bindings

If you have an end-user application that allows the user to edit calendar objects, then you also need to handle changes made by the user and propagate these changes back to the Calendar Store database. How you do this is largely application dependent unless you use Cocoa bindings.

If you use Cocoa bindings, then you should observe local changes to calendar object properties. If you retain a `CalEvent` object and the user can directly edit the `CalEvent` object—for example, you use an `NSArrayController` object to display an array of `CalEvent` objects in an editable table view—, then you should observe changes to `CalEvent` properties. At least observe changes to all `CalEvent` properties that you use in your application.

The following code fragment observes local start and end date changes to a `CalEvent` object.

```
// Observe changes to the start and end dates
[event addObserver:self forKeyPath:@"startDate"
    options:(NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)
    context:NULL];
[event addObserver:self forKeyPath:@"endDate"
    options:(NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)
    context:NULL];
```

Your implementation of the `observeValueForKeyPath:ofObject:change:context:` method of `NSObject` should save the local changes to the Calendar Store database using either the `saveCalendar:error:`, `saveEvent:span:error:`, or `saveTask:error:` method depending on the type of object that changed. See [“Saving Changes”](#) (page 25) for more information on saving changes to the Calendar Store database.

If the end-user can also add and delete calendar objects, then you also need to observe changes to your mutable array—the array that is providing the content to your `NSArrayController` object. See *Cocoa Bindings Programming Topics* for a complete description of Cocoa bindings.

Document Revision History

This table describes the changes to *Calendar Store Programming Guide*.

Date	Notes
2009-07-28	Added information on concurrency to the introduction.
2009-06-17	Minor edits throughout.
2007-10-31	New document that describes concepts and common tasks when using the Calendar Store framework to access iCal data.

