Sample Code

Managing location-based reminders

Access reminders set up with geofence-enabled alarms on a person's calendars.

Download

iOS 26.0+ | iPadOS 26.0+ | Xcode 26.0+

Overview

With the Reminders app, people can create reminders with alarms based on time and location. When Location Services is turned on, people receive location-based reminders when entering or leaving a specified geographic area or geofence. This sample code demonstrates how to add, fetch, complete, remove, filter, and sort location-based reminders. You app must first request full access to reminders from the person using the app before it can access their reminder data. An app with full access can create, edit, save, delete, and fetch all reminders on all of the person's calendars. For more information, see Accessing the event store. Next, register your app for `EKEventStoreChanged` notifications at launch to listen for any changes to the person's Calendar database. When your app receives this notification, consider your current reminder data are stale or invalid and refetch all your reminders. For more information, see Updating with Notifications.

Configure the sample code project

Before running the sample code project in Xcode, select the sample target, then configure it to use your team for signing. For more information, see Assign the project to a team in Preparing your app for distribution.

Provide a map annotation

The sample app presents a map with custom annotations that someone can use to create location-based reminders within the app. It uses location-specific data saved in the `MapData.json` file to create annotations for the map. The sample defines a `MapAnnotation` data type to represent each annotation. `MapData.json` contains three `MapAnnotation` entries. To test reminders around other locations, duplicate and update a `MapAnnotation` entry in `MapData.json` with other data as needed.

The sample displays a settings button that allows the person to grant or deny the app access to location services. If the person grants permission, the app uses the person's current location to add a user annotation to the map. If the person denies access, the app does nothing.

> **Important**
>
> The app includes the `NSLocationWhenInUseUsageDescription` key in its `Info.plist`. This key is required for apps that access the person's location services. For more information on using location services, see Configuring your app to use location services.

Provide a full-access usage description for reminders

The sample includes the `NSRemindersFullAccessUsageDescription` key in its `Info.plist` file. The value of the key is a string that explains why the app needs access to a person's reminders. The system displays the string when prompting the person for authorization.

> **Important**
>
> The `NSRemindersFullAccessUsageDescription` key is required for apps that access a person's reminders. On iOS 17 or later, if your app doesn't include `NSRemindersFullAccessUsageDescription` or the older `NSRemindersUsageDescription` key, iOS automatically denies any access request without a prompt. For more information, see Accessing the event store.

Check your app authorization status

The sample app verifies its authorization status upon launching. The authorization status of the app is `.notDetermined` until the person authorizes or denies access. The person can grant or deny the app access to their reminder data, then change the authorization status later in the Settings app. To determine its status, the app calls the `authorizationStatus(for:)` class method of `EKEventStore` with an entity type `reminder`:

```
authorizationStatus = EKEventStore.authorizationStatus(for: .reminder)
```

Request full access

If the authorization status is `.notDetermined`, the sample app initializes a single instance of `EKEventStore`, `eventStore`, then calls its `requestFullAccessToReminders()` method to prompt the person for full access:

```
return try await withCheckedThrowingContinuation { continuation in
    eventStore.requestFullAccessToReminders { granted, error in
        if let error {
            continuation.resume(throwing: error)
        }
        continuation.resume(returning: granted)
    }
}
```

If the person approves the request, the app receives a `.fullAccess` authorization status. It fetches location reminders in all of the person's calendars, then displays them organized by priority. If the person denies the request, the app gets no access and displays a message prompting the person to grant the app full access in Settings on their device.

> **Important**
>
> Set up your app to instantiate and use a single instance of `EKEventStore` that manages all reminder-related tasks. An `EKEventStore` object requires a significant amount of time to initialize and release.

Check for a default list

Creating a reminder requires a list, which is a calendar for these items. The app calls `defaultCalendarForNewReminders()` on `eventStore` to check whether the person has specified a default list for reminders.

```
eventStore.defaultCalendarForNewReminders() != nil
```

The app fetches and displays location reminders in all of the person's calendars if `defaultCalendarForNewReminders()` returns a value, and shows a message prompting the person to create a list, otherwise.

Create location-based reminders

A location-based reminder is a reminder created with a geofence-enabled alarm. A geofence-enabled alarm has a structured location and proximity configured. The structured location consists of a location object and radius. To use the default radius, set its value to 0. The sample uses the following steps to create a location-based reminder:

1. Create a reminder object.
2. Configure the reminder's calendar and title properties.
3. Add a structured location.
4. Add an alarm.
5. Save the reminder.

First, the sample app creates an EKReminder object using `init(eventStore:)`, then it sets the `title` and `calendar` properties, and other properties, such as priority and time zone:

```
let reminder = EKReminder(eventStore: eventStore)
reminder.calendar = calendar
reminder.title = entry.title
reminder.priority = entry.priority

/*
    The app creates reminders with a specific date and time. To create an
    all-day reminder, set `dueDateComponents` to a date component without
    hour, minute, and second components.
*/
reminder.dueDateComponents = Date.next7DaysComponents

/*
    A floating reminder is one that isn't associated with a specific time
    zone. Set `timeZone` to `nil` if you wish to have a floating reminder.
*/
reminder.timeZone = TimeZone.current
```

Important

The `calendar` and `title` properties are required and must be set before saving the reminder.

Next, the sample creates a structured location by using either `EKStructuredLocation`'s `init(title:)` or `init(mapItem:)` methods. When the location object has latitude and longitude coordinates, the app uses `init(title:)` to create the structured location. The sample

initializes an `CLLocation` object with the specified latitude and longitude, then assigns it to the created structured location's `geoLocation` property:

```
let structuredLocation = EKStructuredLocation(title: annotation.name)
structuredLocation.geoLocation = CLLocation(latitude: annotation.coordinates.latitud
```

When the location object is an `MKMapItem` object, the sample uses `init(mapItem:)` to create the structured location:

```
let structuredLocation = EKStructuredLocation(mapItem: mapItem)
```

EventKit defines the structured location's `radius` property in meters. When someone enters a value for the radius, the app checks the person's preferences for unit of length measurement. If the person's preferred unit of length is a unit other than meters, the sample converts the radius value to meters, then assigns the converted value to the structured location's `radius` property:

```
// Get the person's preferred unit of length measurement.
let preferredUnit = UnitLength(forLocale: .current, usage: .asProvided)
structuredLocation.radius = (preferredUnit == .meters) ? entry.radius : entry.radius
```

Next, the sample creates an `EKAlarm` object, then sets its `structuredLocation` property to the created structured location object. The sample then sets the `proximity` property to a value to finish configuring the alarm's geofence:

```
let alarm = EKAlarm(relativeOffset: 0)
alarm.structuredLocation = structuredLocation
alarm.proximity = entry.proximity
```

The app adds the created alarm to the reminder. For more information on adding alarms, see `Setting an Alarm`.

```
reminder.addAlarm(alarm)
```

Finally, it saves the reminder to the person's Calendar database:

```
try eventStore.save(reminder, commit: true)
```

Fetch location-based reminders

The `fetchReminders(matching:completion:)` method asynchronously fetches all reminders matching a given predicate. The app calls this method with `predicateForReminders(in:)` to fetch complete and incomplete reminders. The predicate takes `nil` or an array of `EKCalendar` objects in its `calendars` parameters. Pass `nil` to fetch from all of the person's calendars, and an array to fetch reminders from a subset of the person's calendars. The app passes `nil` to `predicateForReminders(in:)`:

```
let predicate = eventStore.predicateForReminders(in: nil)
```

Then, the app executes the fetch request. If the request succeeds, `fetchReminders(matching:completion:)` returns an array that contains both time-based and location-based reminders:

```
return await withCheckedContinuation { continuation in
    eventStore.fetchReminders(matching: predicate) { reminders in
        var result: [LocationReminder] = []

        if let reminders {
            result = reminders
                .filter(\.isLocation)
                .map { LocationReminder(reminder: $0) }
        }
        continuation.resume(returning: result)
    }
}
```

To retrieve location-based reminders, the app parses the returned array for reminders defined with an existing alarm that has a `structuredlocation` and `proximity` value:

```
/// Specifies whether the reminder is location-based.
var isLocation: Bool {
    guard let alarms else { return false }

    let proximityAlarms = alarms.filter {
        $0.structuredLocation != nil && ($0.proximity == .enter || $0.proximity ==
    }

    return !proximityAlarms.isEmpty
}
```

Filter and sort reminders

After fetching the location-based reminders, the app displays a segmented control that organizes the fetched reminders by priority: None, Low, Medium, and High. Fetching reminders from the Calendar database returns reminders sorted by creation date. The app offers a menu that lets people choose how to sort the reminders by creation date, due date, or title in ascending order. When someone selects a priority in the control, the sample inspects the fetch result. If the result contains location reminders with the priority the person selected, the app uses the person's sorting preferences to sort the reminders, then it displays them. The sample uses key paths to sort the fetched location-based reminders.

```
/// Sorts reminders by creation date, due date, or title in ascending order.
func reminders(sortedBy sort: ReminderSortValue) -> [LocationReminder] {
    switch sort {
    case .creationDate: return self.sorted(by: \.creationDate)
    case .dueDate: return self.sorted(by: \.dueDate)
    case .title: return self.sorted(by: \.title)
    }
}
```

If the fetch result contains no value, the app prompts the person to add some location reminders with the selected priority.

See Also

Events and reminders

- Creating events and reminders
  Create and modify events and reminders in a person's database.
- Retrieving events and reminders
  Fetch events and reminders from the Calendar database.
- Updating with notifications
  Register for notifications about changes and keep your app up to date.

`class` `EKEvent`
A class that represents an event in a calendar.

`class` `EKReminder`

A class that represents a reminder in a calendar.