Framework

Contacts

Access the user's contacts, and format and localize contact information.

iOS 9.0+ | iPadOS 9.0+ | Mac Catalyst 13.0+ | macOS 10.11+ | visionOS 1.0+ | watchOS 2.0+

Overview

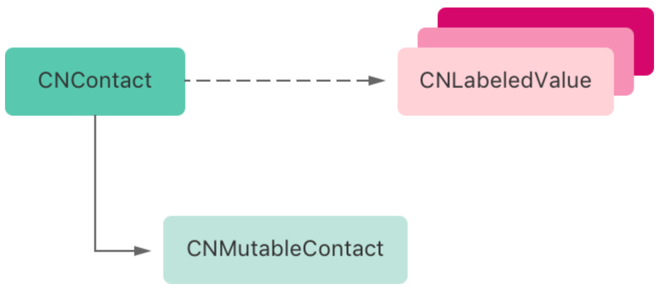
The Contacts framework provides Swift and Objective-C APIs to access the user's contact information. Because most apps read contact information without making any changes, this framework is optimized for thread-safe, read-only usage.

Working with the user's contacts

The Contacts framework is available on all Apple platforms, and replaces the Address Book framework in iOS and macOS.

Contact objects

The contact class (`CNContact`) is a thread-safe, immutable value object of contact properties, such as the contact's name, image, and phone numbers.



The contact class is like `NSDictionary` in that it has a mutable subclass, `CNMutableContact`, you can use to modify contact properties. For contact properties that can have multiple values, such as phone numbers and email addresses, the framework uses an array of `CNLabeledValue` objects. The labeled value class is a thread-safe, immutable tuple of labels and values. Labels describe each value to the user, allowing differentiation, such as home and work phone numbers. The Contacts framework provides some predefined labels and you can create your own custom labels.

```swift
import UIKit
import Contacts

// Create a mutable object to add to the contact.
let contact = CNMutableContact()

// Store the profile picture as data.
let image = UIImage(systemName: "person.crop.circle")
contact.imageData = image?.jpegData(compressionQuality: 1.0)

contact.givenName = "John"
contact.familyName = "Appleseed"

let homeEmail = CNLabeledValue(label: CNLabelHome, value: "john@example.com" as NSSt
let workEmail = CNLabeledValue(label: CNLabelWork, value: "j.appleseed@icloud.com" a
contact.emailAddresses = [homeEmail, workEmail]

contact.phoneNumbers = [CNLabeledValue(
    label: CNLabelPhoneNumberiPhone,
    value: CNPhoneNumber(stringValue: "(408) 555-0126"))]

let homeAddress = CNMutablePostalAddress()
homeAddress.street = "One Apple Park Way"
homeAddress.city = "Cupertino"
homeAddress.state = "CA"
homeAddress.postalCode = "95014"
contact.postalAddresses = [CNLabeledValue(label: CNLabelHome, value: homeAddress)]

var birthday = DateComponents()
birthday.day = 1
birthday.month = 4
birthday.year = 1988  // (Optional) Omit the year value for a yearless birthday.
contact.birthday = birthday

// Save the newly created contact.
let store = CNContactStore()
let saveRequest = CNSaveRequest()
saveRequest.add(contact, toContainerWithIdentifier: nil)
```

```
do {
    try store.execute(saveRequest)
} catch {
    print("Saving contact failed, error: \(error)")
    // Handle the error.
```

Formatting and localization

The Contacts framework helps you format and localize contact information. For example, you can correctly format a contact name (using `CNContactFormatter`) or format an international postal address (using `CNPostalAddressFormatter`).

```
// Formatting the contact name.
let fullName = CNContactFormatter.string(from: contact, style: .fullName)
print("\(String(describing: fullName))")
// John Appleseed

// Formatting the postal address.
let postalString = CNPostalAddressFormatter().string(from: homeAddress)
print("\(postalString)")
// One Apple Park Way
// Cupertino
// CA
// 95014
```

You can display localized object property names and predefined labels based on the current locale setting of the device. Many objects in the Contacts framework, such as `CNContact`, include the `localizedString(forKey:)` method, which lets you get the localized version of a key name. In addition, the `CNLabeledValue` class includes the `localizedString(forLabel:)` method, which lets you get the localized label for the predefined labels in the Contacts framework.

```
// The device locale is Spanish.
let displayName = CNContact.localizedString(forKey: CNContactNicknameKey)
print(displayName)
// Prints "alias"

let displayLabel = CNLabeledValue<NSString>.localizedString(forLabel: CNLabelHome)
print(displayLabel)
// Prints "casa".
```

Fetching contacts

You can fetch contacts using the contact store (CNContactStore), which represents the user's Contacts database. The contact store encapsulates all I/O operations and is responsible for fetching and saving contacts and groups. Because the contact store methods are synchronous, it's best practice to use them on background threads. If necessary, you can safely send immutable fetch results back to the main thread.

The Contacts framework provides several ways to constrain contacts that return from a fetch, including predefined predicates and the keysToFetch property.

CNContact provides predicates for filtering the contacts you want to fetch. For example, to fetch contacts that have the name *Appleseed*, use predicateForContacts(matchingName:) and pass in Appleseed.

```
let predicate = CNContact.predicateForContacts(matchingName: "Appleseed")
```

Note that the Contacts framework doesn't support generic and compound predicates.

You can use keysToFetch to limit the contact properties that you fetch. For example, if you want to fetch only the given name and the family name of a contact, you specify those contact keys in a keysToFetch array.

```
let keysToFetch = [CNContactGivenNameKey, CNContactFamilyNameKey] as [CNKeyDescripto
```

To fetch a contact using both a predicate (predicateForContacts(matchingName:)) and a keysToFetch array, use unifiedContacts(matching:keysToFetch:).

```
let store = CNContactStore()
do {
    let predicate = CNContact.predicateForContacts(matchingName: "Appleseed")
    let contacts = try store.unifiedContacts(matching: predicate, keysToFetch: keysT
    print("Fetched contacts: \(contacts)")
} catch {
    print("Failed to fetch contact, error: \(error)")
    // Handle the error.
}
```

The Contacts framework can also perform operations on the fetched contacts, such as formatting contact names. Each operation requires a specific set of contact keys to correctly perform the operation. The contact keys are key descriptor objects that you need to include within the keysToFetch array. For example, if you want to fetch the contact's email addresses and also be able to format the contact's name (using CNContactFormatter), include both CNContactEmailAddressesKey and the key descriptor object that descriptorForRequiredKeys(for:) returns in the keysToFetch array.

```
let keysToFetch = [CNContactEmailAddressesKey as CNKeyDescriptor, CNContactFormatte
```

Privacy

Users can grant or deny access to contact data on a per-app basis. Any call to CNContactStore blocks the app while asking the user to grant or deny access. Note that the user receives a prompt only the first time an app requests access; all subsequent CNContactStore calls use the existing permissions. To avoid having your app's UI main thread block for this access, you can use either the asynchronous method requestAccess(for:completionHandler:) or dispatch your CNContactStore usage to a background thread.

Important

An iOS app linked on or after iOS 10 needs to include in its Info.plist file the usage description keys for the types of data it needs to access or it crashes. To access Contacts data specifically, it needs to include NSContactsUsageDescription.

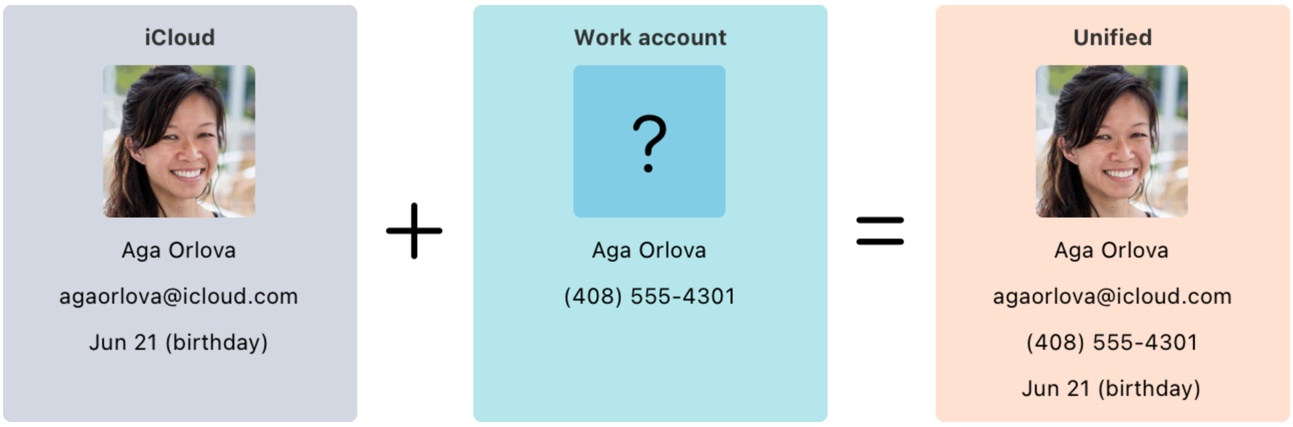
Partial contacts

A partial contact results when the system fetches only some of a contact object's properties from a contact store. All fetched contact objects are partial contacts. If you try to access a property value that the system didn't fetch, you get an exception. If you are unsure which keys the system fetched in the contact, check the availability of the property values before you access them. You can either use isKeyAvailable(_:) to check the availability of a single contact key, or areKeysAvailable(_:) to check multiple keys. If the desired keys aren't available, refetch the contact with them.

```
// Check whether the phone number is available for the given contact.
if contact.isKeyAvailable(CNContactPhoneNumbersKey) {
    print("\(contact.phoneNumbers)")
} else {
    // Refetch the keys.
    let keysToFetch = [CNContactGivenNameKey, CNContactFamilyNameKey, CNContactPhone

    do {
        let refetchedContact = try store.unifiedContact(withIdentifier: contact.ider
        print("\(refetchedContact.phoneNumbers)")
    } catch {
        print("Failed to fetch contact, error: \(error)")
        // Handle the error.
    }
}
```

Unified contacts

You can automatically link contacts in different accounts that represent the same person. Linked contacts display in macOS and iOS apps as unified contacts. A unified contact is an in-memory, temporary view of the set of linked contacts that the system merges into one contact.



By default the Contacts framework returns unified contacts. Each fetched unified contact object (`CNContact`) has its own unique identifier that's different from any individual contact's identifier in the set of linked contacts. When refetching a unified contact, be sure to use its identifier.

Saving contacts

The contact store (`CNContactStore`) also saves changes to the Contacts framework objects. The `CNSaveRequest` class enables save operations and allows batching of changes to multiple contacts and groups into a single operation. After adding all objects to the save request, it can execute with a contact store as the code example below shows. Don't access the objects in the save request while the save is executing, because the objects may have modifications.

> Note
>
> `CNSaveRequest` isn't available in watchOS.

Create and save a new contact.

```
// Create a new contact.
let newContact = CNMutableContact()
newContact.givenName = "John"
newContact.familyName = "Appleseed"

// Save the contact.
let saveRequest = CNSaveRequest()
saveRequest.add(newContact, toContainerWithIdentifier: nil)

do {
    try store.execute(saveRequest)
} catch {
    print("Saving contact failed, error: \(error)")
    // Handle the error.
}
```

Modify and save an existing contact.

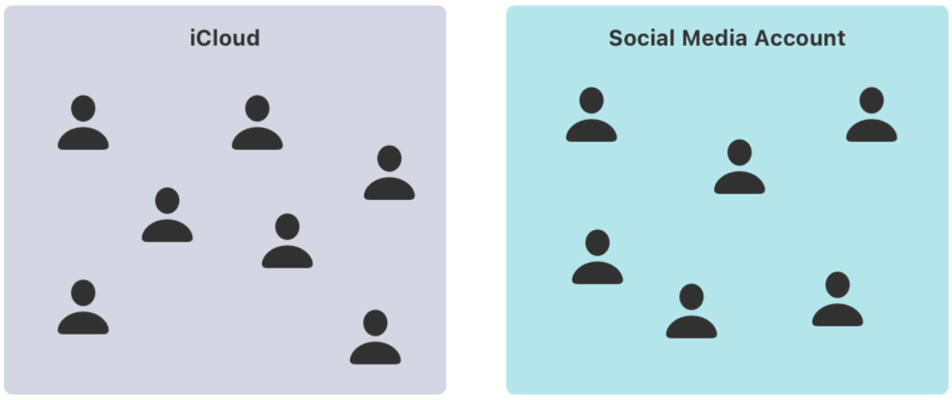
```
// Update the home email address for John Appleseed.
guard let mutableContact = contact.mutableCopy() as? CNMutableContact else { return
let newEmail = CNLabeledValue(label: CNLabelHome, value: "john@example.com" as NSSt
mutableContact.emailAddresses.append(newEmail)

let saveRequest = CNSaveRequest()
saveRequest.update(mutableContact)
do {
    try store.execute(saveRequest)
} catch {
    print("Saving contact failed, error: \(error)")
    // Handle the error.
}
```

Contacts changed notifications

After successfully executing a save, the contact store posts a `CNContactStoreDidChange` notification to the default notification center. If you cache any Contacts framework objects, you need to refetch those objects, either by their identifiers, or with the predicates that you used to originally fetch them, and then release the cached objects. Note that cached objects are stale, but not invalid.

Containers and groups

A user may have contacts in their device's local account or server accounts that they configure to sync contacts. Each account has at least one container of contacts. A contact can be in only one container.



A group is a set of contacts within a container. Not all accounts support groups, and some accounts also support subgroups. An iCloud account has only one container and may have many groups, but no subgroups. An Exchange account doesn't support groups, but may have multiple containers representing Exchange folders.

Topics

Essentials

Accessing the contact store

Request permission from the person to read and write their contact data.

Accessing a person's contact data using Contacts and ContactsUI

Allow people to grant your app access to contact data by adding the Contact access button and Contact access picker to your app.

`class CNContactStore`

The object that fetches and saves contacts, groups, and containers from the user's Contacts database.

`NSContactsUsageDescription`

A message that tells people why the app is requesting access to their contacts.

`com.apple.developer.contacts.notes`

A Boolean value that indicates whether the app may access the notes in contact entries.

Contact data

`class CNContact`

An immutable object that stores information about a single contact, such as the contact's first name, phone numbers, and addresses.

`class CNMutableContact`

A mutable object that stores information about a single contact, such as the contact's first name, phone numbers, and addresses.

Data Objects

Access contact-related data, such as the user's postal address and phone number.

Contact Keys

Specify contact-related properties during fetch operations.

Fetch and save requests

`class CNContactFetchRequest`

An object that defines the options to use when fetching contacts.

`class CNFetchRequest`

The base class for contact fetch requests.

`class CNFetchResult`

An object that represents the result of a change-history fetch request.

`class CNSaveRequest`

An object that collects the changes you want to save to the user's contacts database.

Change history data

`class CNChangeHistoryAddContactEvent`

An object that represents a user adding a contact.

`class CNChangeHistoryAddGroupEvent`

An object that represents a user adding a group.

`class CNChangeHistoryAddMemberToGroupEvent`

An object that represents a user adding a contact to a group.

`class CNChangeHistoryAddSubgroupToGroupEvent`

An object that represents a user adding a subgroup to a group.

`class CNChangeHistoryDeleteContactEvent`

An object that represents a user deleting a contact.

`class CNChangeHistoryDeleteGroupEvent`

An object that represents a user deleting a group.

`class CNChangeHistoryDropEverythingEvent`

An object that indicates the delegate should drop all contacts and groups before handling change events.

`class CNChangeHistoryEvent`

An object that represents the user adding, updating, or deleting a contact or group.

`class CNChangeHistoryFetchRequest`

An object that specifies the criteria for fetching change history.

`class CNChangeHistoryRemoveMemberFromGroupEvent`

An object that represents a user removing a contact from a group.

`class CNChangeHistoryRemoveSubgroupFromGroupEvent`

An object that represents a user removing a subgroup from a group.

`class CNChangeHistoryUpdateContactEvent`

An object that represents a user updating a contact.

`class CNChangeHistoryUpdateGroupEvent`

An object that represents an updated group event.

`protocol CNChangeHistoryEventVisitor`

An interface for receiving notice of changes to contacts and groups.

Formatters

`class CNContactFormatter`

An object that you use to format contact information before displaying it to the user.

`class CNPostalAddressFormatter`

An object that you use to format a contact's postal addresses.

`class CNContactVCardSerialization`

An object you use to convert to and from a vCard representation of the user’s contacts.

`class CNContactsUserDefaults`

An object that defines the default options to use when displaying contacts.

Errors

Error Information

Diagnose errors generated by the Contacts framework.