Handoff Programming Guide

Developer

Contents

Listings

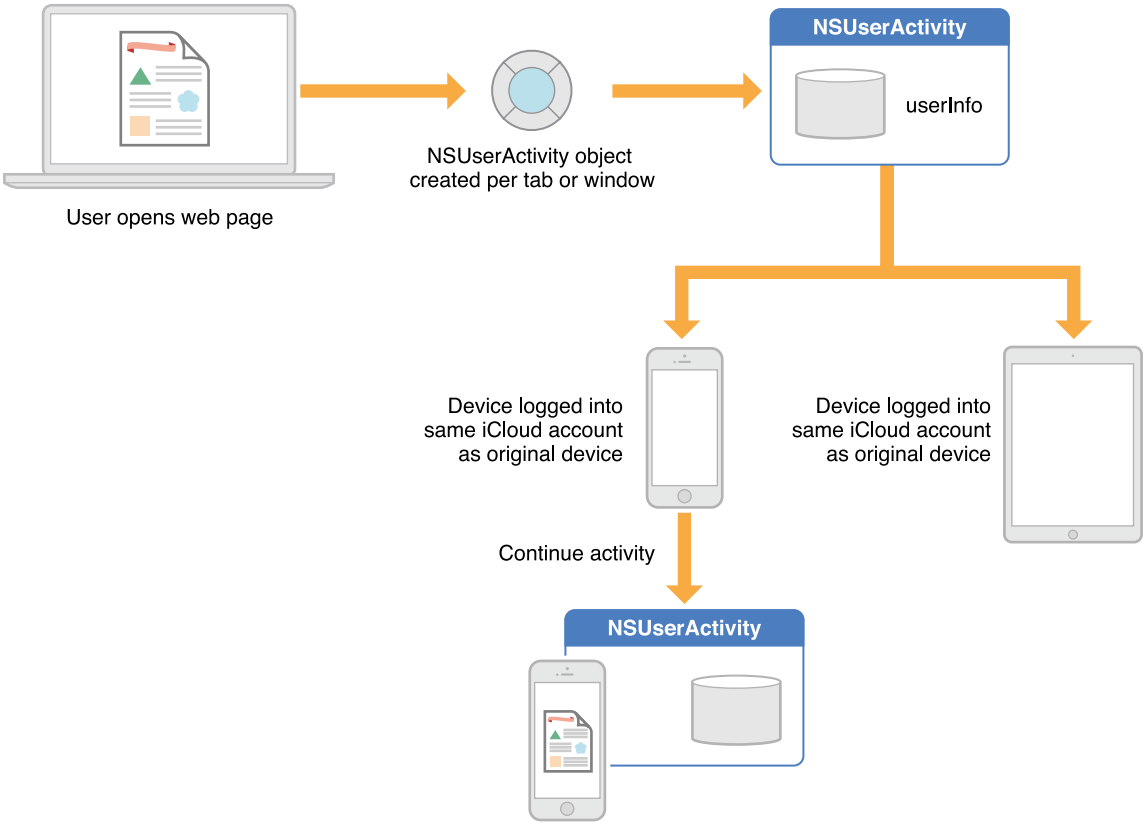
Objective-CSwift

About Handoff

SwiftObjective-C

Handoff is a capability introduced in iOS 8 and OS X v10.10 that transfers user activities among multiple devices associated with the same user.

Handoff enables the user to switch from one device to another and continue an ongoing activity seamlessly, without reconfiguring each device independently. For example, a user who is browsing a long article in Safari on a Mac can move to a nearby iOS device that's signed into iCloud with the same Apple ID and open the same webpage automatically in Safari on iOS, at the same scroll position as on the original device.



Apple apps, such as Safari, Mail, Maps, Contacts, Notes, Calendar, and Reminders use public APIs to implement Handoff for iOS 8 and OS X v10.10. A third-party developer can use the same APIs to implement Handoff in apps that share the developer's Team ID. Such apps must either be distributed through the App Store or signed by the registered developer.

Handoff Interactions

Handing off a user activity involves three phases:

- Create a user activity object for each activity the user engages in your app.
- Update the user activity object regularly with information about what the user is doing.
- Continue the user activity on a different device when the user requests it.

Document-based apps (that is, apps based on a subclass of `NSDocument` or `UIDocument`), provide built-in support for all three phases of the handoff scenario. Responder objects (subclasses of `NSResponder` and `UIResponder`) provide built-in support for updating user activities and managing their current status. Your app can also create, update, and continue user activities directly, working especially with the app delegate.

The Handoff mechanism depends primarily on objects of a single class in Foundation, `NSUserActivity`, with support of additional small APIs in UIKit and AppKit. Apps encapsulate information about a user's activities in `NSUserActivity` objects, and those activities become candidates for continuation on other devices. Handoff of a given user activity requires the originating app to designate that activity's `NSUserActivity` object as the current activity, save pertinent data for continuation on another device, and send the data to the resuming device. Handoff passes only enough information between the devices to describe the activity itself, while larger-scale data synchronization is handled through iCloud.

On the continuing device, the user is notified that an activity is available for continuation. If the user chooses to continue the activity, an appropriate app is launched and provided with the activity's payload data. A user activity can be continued only in an app that has the same developer Team ID as the activity's source app and that supports the activity's type. Supported activity types are specified in the app's `Info.plist` under the `NSUserActivityTypes` key. So, the continuing device chooses the appropriate app based on the target Team ID, activity type property of the originating `NSUserActivity` object, and optionally the activity object's title property. From the information in the user activity object's `userInfo` dictionary, the continuing app can then configure its user interface and state appropriately for seamless continuation of the user's activity.

Optionally, if continuing an activity requires more data than can be efficiently transferred by the initial transport mechanism, a resuming app can call back to the originating app's activity object to open streams between the apps and transfer more data. For example, if the activity to be continued is composing an email message that contains an image, then the streams option is the best way to transfer the data needed to continue the composition on another device. For more information, see Using Continuation Streams (page 19).

Document-based apps on iOS and OS X automatically support Handoff, as described in Supporting a User Activity in Document-Based Apps (page 8).

User Activity Object

An `NSUserActivity` object encapsulates the state of a user activity in an app on a particular device. It is the primary object in the Handoff mechanism. The originating app creates a user activity object for each user activity it supports for possible handoff to another device. For example, a web browser would create a user activity object for each open tab or window in which the user is browsing URLs. However, only the activity object corresponding to the frontmost tab or window is current at a given time, and only the current activity is available for continuation.

An `NSUserActivity` object is identified by its `activityType` and `title` properties. It has a `userInfo` dictionary to contain its state data and a dirty flag named `needsSave` to support lazy updating of its state by its delegate. The `NSUserActivity` method `addUserInfoEntriesFromDictionary:` enables the delegate and other clients to merge state data into its `userInfo` dictionary.

For more information, see *NSUserActivity Class Reference*.

User Activity Delegate

The user activity delegate is an object that conforms to the `NSUserActivityDelegate` protocol. It is typically a top-level object in the app, such as a view controller or the app delegate, that manages the activity's interaction with the app.

The user activity delegate is represented by the `delegate` property of `NSUserActivity` and is responsible for keeping the data in the `NSUserActivity` object's user info dictionary up to date so that it can be handed off to another device. When the system needs the activity to be updated, such as before the activity is continued on another device, it calls the delegate's `userActivityWillSave:` method. You can implement this callback to make updates to the object's data-bearing properties such as `userInfo`, `title`, and so on. Once the system calls this method, it resets `needsSave` to `NO`. Change this value to `YES` if something happens that changes the `userInfo` or other data-bearing properties again.

Alternatively, instead of implementing the delegate's `userActivityWillSave:` method as described in the preceding paragraph, you can have UIKit or AppKit manage the user activity automatically. The app opts into this behavior by setting a responder object's `userActivity` property and implementing the responder's `updateUserActivityState:` callback, as described in Managing a User Activity With Responders (page 8). This arrangement is preferred if it works for your user activity.

For more information, see *NSUserActivityDelegate Protocol Reference*.

App Framework Support

UIKit and AppKit provide support for Handoff in the document, responder, and app delegate classes. Although there are minor behavioral differences between the platforms, the basic mechanism, which enables apps to save and restore user activities, is the same, and the APIs are the same.

Supporting a User Activity in Document-Based Apps

A document-based app on iOS and OS X automatically supports Handoff if you add an `NSUbiquitousDocumentUserActivityType` key and value for each `CFBundleDocumentTypes` entry in your app's `Info.plist` property list file. If this key is present, `NSDocument` and `UIDocument` automatically create `NSUserActivity` objects for iCloud-based documents of the given document type. The value of `NSUbiquitousDocumentUserActivityType` is a string that represents the `NSUserActivity` object's activity type. That is, you provide an activity type for each document type supported by your document-based app. Multiple document types can have the same activity type. `NSDocument` and `UIDocument` automatically put the value of their `fileURL` property into the activity object's `userInfo` dictionary with the `NSUserActivityDocumentURLKey`.

In OS X, AppKit can automatically restore `NSUserActivity` objects created in this way. It does so if the app delegate method `application:continueUserActivity:restorationHandler:` returns `NO` or is unimplemented. In this situation, the document is opened with the `NSDocumentController` method `openDocumentWithContentsOfURL:display:completionHandler:` and receives a `restoreUserActivityState:` message.

For more information, see Adopting Handoff in Document-Based Apps (page 10). Also see *NSDocument Class Reference* and *UIDocument Class Reference*.

Managing a User Activity with Responders

UIKit and AppKit can manage a user activity if you set it as a responder object's `userActivity` property. When the responder knows that the activity state is dirty, it must set the object's `needsSave` property to `YES`. The system automatically saves the `NSUserActivity` object at appropriate times, first giving the responder an opportunity to update the activity's state through the `updateUserActivityState:` callback. Your responder subclass must override the `updateUserActivityState:` method to add state data to the user activity object. If multiple responders share a single `NSUserActivity` object, they all receive an `updateUserActivityState:` callback when the system updates the user activity object. Before the update callbacks are sent, the activity object's `userInfo` dictionary is cleared.

On OS X, `NSUserActivity` objects managed by AppKit and associated with responders automatically become current based on the main window and the responder chain, that is, when the document's window becomes the main window. On iOS, however, for `NSUserActivity` objects managed by UIKit, you must either call `becomeCurrent` explicitly or have the document's `NSUserActivity` object set on a `UIViewController` object that is in the view hierarchy when the app comes to the foreground.

A responder can set its `userActivity` property to `nil` to disassociate itself from an activity. When an `NSUserActivity` object managed by the app framework has no more associated responders or documents, it is automatically invalidated.

For more information, see Adopting Handoff in Responders (page 15). Also see *NSResponder Class Reference* or *UIResponder Class Reference*.

Continuing an Activity Using the App Delegate

The app delegate is the primary entry point for continuing a user activity in a non-document-based app. As soon as the user responds to the notification by choosing to continue an activity, Handoff launches the appropriate app and sends the app's delegate an `application:willContinueUserActivityWithType:` message. The app lets the user know that the activity will continue shortly. Meanwhile, the `NSUserActivity` object is delivered when the delegate receives an `application:continueUserActivity:restorationHandler:` message. You should implement this method to configure your app in such a way that it can resume the activity represented by the user activity object.

The `application:continueUserActivity:restorationHandler:` message includes a block, the restoration handler, that you can optionally call if your app uses auxiliary responder or document objects to perform the resuming user activity. Create these objects (or fetch them if cached) and pass them to the restoration handler in its `NSArray` parameter. The system then sends each object a `restoreUserActivityState:` message, passing the user activity object. Each object can use the activity's `userInfo` data to resume the activity. For more information about using this restoration handler, see the description of the `application:continueUserActivity:restorationHandler:` method in *NSApplicationDelegate Protocol Reference*.

If you do not implement `application:continueUserActivity:restorationHandler:` or return `NO` from it, and your app is document-based, AppKit can automatically resume the activity, as described in Supporting User Activity in Document-Based Apps (page 8). For more details, see Continuing an Activity (page 15).

Adopting Handoff

Objective-CSwift

User activities can be shared among apps that are signed with the same developer team identifier and supporting a given activity type. If an app is document-based, it can opt to support Handoff automatically. Otherwise, apps must adopt a small API in Foundation, as described in this chapter.

Identifying User Activities

The first step in implementing Handoff is to identify the types of user activities that your app supports. For example, an email app could support composing and reading messages as two separate user activities. A list-handling app could support creating (and editing) list items as one user activity type, and it could support browsing lists and items as another. Your app can support as many activity types as you wish, whatever users do in your app. For each activity type, your app needs to identify when an activity of that type begins and ends, and it needs to maintain up-to-date state data sufficient to enable the activity to continue on another device.

User activities can be shared among any apps signed with the same team identifier, and you don't need a one-to-one mapping between originating and resuming apps. For example, one app creates three different types of activities, and those activities are resumed by three different apps on the second device. This asymmetry can be a common scenario, given the preference for iOS apps to be smaller and more focused on a dedicated purpose than more comprehensive Mac apps.

Adopting Handoff in Document-Based Apps

Document-based apps on iOS and OS X automatically support Handoff by automatically creating `NSUserActivity` objects for iCloud-based documents if the app's `Info.plist` property list file includes a `CFBundleDocumentTypes` key of `NSUbiquitousDocumentUserActivityType`, as shown in Listing 2-1. The value of `NSUbiquitousDocumentUserActivityType` is a string used for the `NSUserActivity` object's activity type. The activity type correlates with the app's role for the given document type, such as editor or viewer, and an activity type can apply to multiple document types. In Listing 2-1 the string is a reverse-DNS app designator with the name of the activity, `editing`, appended. If they are represented in this way, the activity type entries do not need to be repeated in the `NSUserActivityTypes` array of the app's `Info.plist`.

Listing 2-1 Info.plist entry for Handoff in document-based apps

```
<key>CFBundleDocumentTypes</key>
<array>
    <dict>
        <key>CFBundleTypeName</key>
        <string>NSRTFDPboardType</string>
        . . .
        <key>LSItemContentTypes</key>
        <array>
            <string>com.myCompany.rtfd</string>
        </array>
        . . .
        <key>NSUbiquitousDocumentUserActivityType</key>
        <string>com.myCompany.myEditor.editing</string>
    </dict>
</array>
```

The document's URL is put into the `userInfo` dictionary with the `NSUserActivityDocumentURLKey`.

The automatically created user activity object is available through the document's `userActivity` property and can be referenced by other objects in the app, such as a view controller in iOS or window controller in OS X. This referencing enables apps to track position in a document, for example, or to track the selection of particular elements. The app sets the activity object's `needsSave` property to `YES` whenever that state changes and saves the state in its `updateUserActivityState:` callback.

The `userActivity` property can be used from any thread. It conforms to the key-value observing (KVO) protocol so that a `userActivity` object can be shared with other objects that need to be kept in sync as the document moves into and out of iCloud. A document's user activity objects are invalidated when the document is closed.

Implementing Handoff Directly

Adopting Handoff in your app requires you to write code that uses APIs in UIKit and AppKit provided for creating a user activity object, updating the state of the object to track the activity, and continuing the activity on another device.

Creating the User Activity Object

Every user activity that can potentially be handed off to a continuing device is represented by a user activity object instantiated from the `NSUserActivity` class. An originating app creates a user activity object for each user activity it supports. The nature of those user activities depends on the app. For example, a web browser might designate the user browsing a web page as one activity. The app creates an `NSUserActivity` instance, as shown in Listing 2-2, whenever the user opens a new window or tab displaying content from a URL, placing the URL in the activity object's `userInfo` dictionary, along with the scroll position of the page. Place this code in a controller object such as a window or view controller that has knowledge of the current state of the activity and that can update the state data in the activity object as necessary.

Listing 2-2 Creating the user activity object

```
NSUserActivity* myActivity = [[NSUserActivity alloc]
                      initWithActivityType: @"com.myCompany.myBrowser.browsing"];
myActivity.userInfo = @{ ... };
myActivity.title = @"Browsing";
[myActivity becomeCurrent];
```

When your app is finished with an `NSUserActivity` object, it should call `invalidate` before deallocating the object. This makes the object disappear from all devices (if it was present) and frees up any system resources devoted to that user activity object.

Specifying an Activity Type

The activity type identifier is a short string appearing in your app's `Info.plist` property list file in its `NSUserActivityTypes` array, which lists all the activity types your app supports. The same string is passed when you create the activity, as shown in Listing 2-2 (page 12) where the activity object is created with the activity type of `com.myCompany.myBrowser.browsing`, a reverse-DNS-style notation meant to avoid collisions. When the user chooses to continue the activity, the activity type (along with the app's Team ID) determines which app to launch on the receiving device to continue the activity.

> **Note:** You specify the activity type of an `NSUserActivity` object when you create the instance. You cannot change the activity type of the object after it is created.

For example, a Reminders-style app serializes the reminder list the user is looking at. When the user clicks on a new reminder list, the app tracks that activity in the `NSUserActivityDelegate`. Listing 2-3 shows a possible implementation of a method that gets called whenever the user switches to a different reminder list. This app appends an activity name to the app's bundle identifier to create the activity type to use when it creates its `NSUserActivity` object.

Listing 2-3 Tracking a user activity

```
// UIResponder and NSResponder have a userActivity property
NSUserActivity *currentActivity = [self userActivity];

// Build an activity type using the app's bundle identifier
NSString *bundleName = [[NSBundle mainBundle] bundleIdentifier];
NSString *myActivityType =
                [bundleName stringByAppendingString:@".selected-list"];

if(![[currentActivity activityType] isEqualToString:myActivityType]) {
    [currentActivity invalidate];

    currentActivity = [[NSUserActivity alloc]
                                initWithActivityType:myActivityType];
    [currentActivity setDelegate:self];
    [currentActivity setNeedsSave:YES];

    [self setUserActivity:currentActivity];

} else {

    // Already tracking user activity of this type
    [currentActivity setNeedsSave:YES];
}
```

The code in Listing 2-3 uses the `setNeedsSave:` accessor method to mark the user activity object as needing to to be updated. This enables the system to coalesce updates and perform them lazily.

Populating the Activity Object's User Info Dictionary

The activity object has a user info dictionary that contains whatever data is needed to hand off the activity to the continuing app. The user info dictionary can contain `NSArray`, `NSData`, `NSDate`, `NSDictionary`, `NSNull`, `NSNumber`, `NSSet`, `NSString`, and `NSURL` objects. The system modifies `NSURL` objects that use the `file:` scheme and point at iCloud documents to point to those same items in the corresponding container on the receiving device.

> **Note:** Transfer as small a payload as possible in the `userInfo` dictionary—3KB or less. The more payload data you deliver, the longer it takes the activity to resume.

Listing 2-4 shows an example that creates a user activity object for an app that reads documents on a website. The activity type, set when the object is created, is shown in reverse-DNS-style notation that specifies the company, app, and finally the particular activity. The `webpageURL` property represents the URL where the document being read is located, and the user info dictionary is populated with keys and values representing the document's name and the current page number and scroll position. As the reader progresses through a document, your app needs to keep that information current.

Listing 2-4 Initializing a user info dictionary

```
NSUserActivity* myActivity = [[NSUserActivity alloc]
                      initWithActivityType: @"com.myCompany.myReader.reading"];

// Initialize userInfo
NSURL* webpageURL = [NSURL URLWithString:@"http://www.myCompany.com"];
myActivity.userInfo = @{
             @"docName" : currentDoc,
             @"pageNumber" : self.pageNumber,
             @"scrollPosition" : self.scrollPosition
};
```

Adopting Handoff in Responders

You can associate responder objects (inheriting from `NSResponder` on OS X or `UIResponder` on iOS) with a given user activity if you set the activity as the responder's `userActivity` property. The system automatically saves the `NSUserActivity` object at appropriate times, calling the responder's `updateUserActivityState:` override to add current data to the user activity object using the activity object's `addUserInfoEntriesFromDictionary:` method.

Listing 2-5 Responder override for updating an activity's state

```
- (void)updateUserActivityState:(NSUserActivity *)userActivity {
    . . .
    [userActivity setTitle: self.activityTitle];
    [userActivity addUserInfoEntriesFromDictionary: self.activityUserInfo];
}
```

Continuing an Activity

Handoff automatically advertises user activities that are available to be continued on iOS and OS X devices that are in physical proximity to the originating device and signed into the same iCloud account as the originating device. When the user chooses to continue a given activity, Handoff launches the appropriate app and sends the app delegate messages that determine how the activity is resumed, as described in Continuing an Activity Using the App Delegate (page 9).

Implement the `application:willContinueUserActivityWithType:` method to let the user know the activity will continue shortly. Use the the `application:continueUserActivity:restorationHandler:` method to configure the app to continue the activity. The system calls this method when the activity object, including activity state data in its `userInfo` dictionary, is available to the continuing app.

> **Note:** For URLs transferred in the `userInfo` dictionary of an `NSUserActivity` object, you must call `startAccessingSecurityScopedResource` and it must return `YES` before you can access the URL. Call `stopAccessingSecurityScopedResource` when you are done using the file.
>
> Exceptions to this requirement are URLs of `UIDocument` documents and those of `NSDocument` that are automatically created for apps specifying `NSUbiquitousDocumentUserActivityType` and returning `NO` from `application:continueUserActivity:restorationHandler:` (or leaving it unimplemented). See Adopting Handoff in Document-Based Apps (page 10).

Additional configuration of your app for continuing the activity can optionally be performed by objects you give to the restoration handler block that is passed in with the `application:continueUserActivity:restorationHandler:` message. Listing 2-6 shows a simple implementation of this method.

Listing 2-6 Continuing a user activity

```
- (BOOL)application:(NSApplication *)application
             continueUserActivity: (NSUserActivity *)userActivity
             restorationHandler: (void (^)(NSArray *))restorationHandler {

    BOOL handled = NO;

    // Extract the payload
    NSString *type = [userActivity activityType];
    NSString *title = [userActivity title];
    NSDictionary *userInfo = [userActivity userInfo];

   // Assume the app delegate has a text field to display the activity information
    [appDelegateTextField setStringValue: [NSString stringWithFormat:
        @"User activity is of type %@, has title %@, and user info %@",
        type, title, userInfo]];

    restorationHandler(self.windowControllers);
    handled = YES;

    return handled;
}
```

In this case, the app delegate has an array of `NSWindowController` objects, `windowControllers`. These window controllers know how to configure all of the app's windows to resume the activity. After you pass that array to the `restorationHandler` block, Handoff sends each of those objects a `restoreUserActivityState:` message, passing in the resuming activity's `NSUserActivity` object. The window controllers inherit the `restoreUserActivityState:` method from `NSResponder`, and each controller object overrides that method to configure its window, using the information in the activity object's `userInfo` dictionary.

To support graceful failure, the app delegate should implement the `application:didFailToContinueUserActivityWithType:error:` method. If you don't implement that method, the app framework nonetheless displays diagnostic information contained in the passed-in `NSError` object.

> **Note:** The `UIApplicationDelegate` methods for handoff, described in this section, are not called when either of the application delegate methods `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` returns `NO`.

Native App–to–Web Browser Handoff

When using a native app on the originating device, the user may want to continue the activity on another device that does not have a corresponding native app. If there is a web page that corresponds to the activity, it can still be handed off. For example, video library apps enable users to browse movies available for viewing, and mail apps enable users to read and compose email, and in many cases users can do the same activity though a web-page interface. In this case, the native app knows the URL for the web interface, possibly including syntax designating a particular video being browsed or message being read. So, when the native app creates the `NSUserActivity` object, it sets the `webpageURL` property, and if the receiving device doesn't have an app that supports the user activity's `activityType`, it can resume the activity in the default web-browser of the continuing platform.

A web browser on OS X that wants to continue an activity in this way should claim the `NSUserActivityTypeBrowsingWeb` activity type (by entering that string in its `NSUserActivityTypes` array in the app's `Info.plist` property list file). This ensures that if the user selects that browser as their default browser, it receives the activity object instead of Safari.

Web Browser–to–Native App Handoff

In the opposite case, if the user is using a web browser on the originating device, and the receiving device is an iOS device with a native app that claims the domain portion of the `webpageURL` property, then iOS launches the native app and sends it an `NSUserActivity` object with an `activityType` value of `NSUserActivityTypeBrowsingWeb`. The `webpageURL` property contains the URL the user was visiting, while the `userInfo` dictionary is empty.

The native app on the receiving device must opt into this behavior by claiming a domain in the `com.apple.developer.associated-domains` entitlement. The value of that entitlement has the format `<service>:<fully qualified domain name>`, for example, `activitycontinuation:example.com`. In this case the service must be `activitycontinuation`. Add the value for the `com.apple.developer.associated-domains` entitlement in Xcode in the Associated Domains section under the Capabilities tab of the target settings.

If that domain matches the `webpageURL` property, Handoff downloads a list of approved app IDs from the domain. Domain-approved apps are authorized to continue the activity. On your website, you list the approved apps in a signed JSON file named `apple-app-site-association`, for example, `https://example.com/apple-app-site-association`. (You must use an actual device, rather than the simulator, to test downloading the JSON file.)

The JSON file contains a dictionary that specifies a list of app identifiers in the format `<team identifier>.<bundle identifier>` in the General tab of the target settings, for example, `YWBN8XTPBJ.com.example.myApp`. Listing 2-7 shows an example of such a JSON file formatted for reading.

Listing 2-7 Server-side web credentials

```
{
    "activitycontinuation": {
    "apps": [    "YWBN8XTPBJ.com.example.myApp",
                 "YWBN8XTPBJ.com.example.myOtherApp" ]
    }
}
```

To sign the JSON file (so that it is returned from the server with the correct Content-Type of `application/pkcs7-mime`), put the content into a text file and sign it. You can perform this task with Terminal commands such as those shown in Listing 2-8, removing the white space from the text for ease of manipulation, and using the `openssl` command with the certificate and key for an identity issued by a certificate

authority trusted by iOS (that is, listed at http://support.apple.com/kb/ht5012). It need not be the same identity hosting the web credentials (https://example.com in the example listing), but it must be a valid TLS certificate for the domain name in question.

Listing 2-8 Signing the credentials file

```
echo '{"activitycontinuation":{"apps":["YWBN8XTPBJ.com.example.myApp",
"YWBN8XTPBJ.com.example.myOtherApp"]}}' > json.txt

cat json.txt | openssl smime -sign -inkey example.com.key
                             -signer example.com.pem
                             -certfile intermediate.pem
                             -noattr -nodetach
                             -outform DER > apple-app-site-association
```

The output of the `openssl` command is the signed JSON file that you put on your website at the `apple-app-site-association` URL, in this example, `https://example.com/apple-app-site-association`.

An app can set the `webpageURL` property to any web URL, but it only receives activity objects whose `webpageURL` domain is in its `com.apple.developer.associated-domains` entitlement. Also, the scheme of the `webpageURL` must be `http` or `https`. Any other scheme throws an exception.

Using Continuation Streams

If resuming an activity requires more data than can be efficiently transferred by the initial Handoff payload, a continuing app can call back to the originating app's activity object to open streams between the apps and transfer more data. In this case, the originating app sets its `NSUserActivity` object's Boolean property `supportsContinuationStreams` to `YES`, sets the user activity delegate, then calls `becomeCurrent`, as shown in Listing 2-9.

Listing 2-9 Setting up streams

```
NSUserActivity* activity = [[NSUserActivity alloc] init];
activity.title = @"Editing Mail";
activity.supportsContinuationStreams = YES;
activity.delegate = self;
[activity becomeCurrent];
```

On the continuing device, after users indicate they want to resume the activity, the system launches the appropriate app and begins sending messages to the app delegate. The app delegate can then request streams back to the originating app by sending its user activity object the `getContinuationStreamsWithCompletionHandler:` message, as shown in the override implementation in Listing 2-10.

Listing 2-10 Requesting streams

```
- (BOOL)application:(UIApplication *)application
        continueUserActivity: (NSUserActivity *)userActivity
       restorationHandler: (void(^)(NSArray *restorableObjects))restorationHandler
{
    [userActivity getContinuationStreamsWithCompletionHandler:^(
                        NSInputStream *inputStream,
                        NSOutputStream *outputStream, NSError *error) {

        // Do something with the streams

        }];

    return YES;
}
```

On the originating device, the user activity delegate receives the streams in a callback to its `userActivity:didReceiveInputStream:outputStream:` method, which it implements to provide the data needed to continue the user activity on the resuming device using the streams.

`NSInputStream` provides read-only access to stream data, and `NSOutputStream` provides write-only access. Therefore, data written to the output stream on the originating side is read from the input stream on the continuing side, and vice versa. Streams are meant to be used in a request-and-response fashion; that is, the continuing side uses the streams to request more continuation data from the originating side which then uses the streams to provide the requested data.

Continuation streams are an optional feature of Handoff, and most user activities do not need them for successful continuation. Even when streams are needed, in most cases there should be minimal back and forth between the apps. A simple request from the continuing app accompanied by a response from the originating app should be enough for most continuation events.

Best Practices

Implementing successful continuation of activities requires careful design because numerous and various components, apps, software objects, and platforms can be involved.

- Transfer as small a payload as possible in the `userInfo` dictionary—3KB or less. The more payload data you deliver, the longer it takes the activity to resume.
- When a large amount of data transfer is unavoidable, use streams, but recognize that they have a cost in terms of network setup and overhead.
- Plan for different versions of apps on different platforms to work well with each other or fail gracefully. Remember that the complementary app design can be asymmetrical—for example, a monolithic Mac app can route each of its activity types to smaller, special-purpose apps on iOS.
- Use reverse-DNS notation for your activity types to avoid collisions. If the activity pertains only to a single app, you can use the app identifier with an extra field appended to describe the activity type. For example, use a format such as `com.<company>.<app>.<activity type>`, as in `com.myCompany.myEditor.editing`. If you have a user activity that works across more than one app, you can drop the app field, as in `com.myCompany.editing`.
- To update the activity object's `userInfo` dictionary efficiently, configure its delegate and set its `needsSave` property to `YES` whenever the `userInfo` needs updating. At appropriate times, Handoff invokes the delegate's `userActivityWillSave:` callback, and the delegate can update the activity state.
- Be sure the delegate of the continuing app implements its `application:willContinueUserActivityWithType:` to let the user know the activity will be continued. The user activity object may not be available instantly.

Document Revision History

This table describes the changes to *Handoff Programming Guide*.

Date	Notes
2014-10-16	Added guidance that you must use an actual device, rather than the simulator, to test downloading the list of approved apps for web browser–to–native app handoff.
2014-09-17	New document that explains how to implement Handoff features in OS X and iOS apps.

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