Inter-App Communication

Apps communicate only indirectly with other apps on a device. You can use AirDrop to share files and data with other apps. You can also define a custom URL scheme so that apps can send information to your app using URLs.

Note: You can also send files between apps using a `UIDocumentInteractionController` object or a document picker. For information about adding support for a document interaction controller, see *Document Interaction Programming Topics for iOS*. For information about using a document picker to open files, see *Document Picker Programming Guide*.

Supporting AirDrop

AirDrop lets you share photos, documents, URLs, and other types of data with nearby devices. AirDrop takes advantage of peer-to-peer networking to find nearby devices and connect to them.

Sending Files and Data to Another App

To send files and data using AirDrop, use a `UIActivityViewController` object to display an activity sheet from your user interface using. When creating this view controller, you specify the data objects that you want to share. The view controller displays only those activities that support the specified data. For AirDrop, you can specify images, strings, URLs, and several other types of data. You can also pass custom objects that adopt the `UIActivityItemSource` protocol.

To display an activity view controller, you can use code similar to that shown in Listing 6-1. The activity view controller automatically uses the type of the specified object to determine what activities to display in the activity sheet. You do not have to specify the AirDrop activity explicitly. However, you can prevent the sheet from displaying specific types using the view controller's `excludedActivityTypes` property. When displaying an activity view controller on iPad, you must use a popover.

Listing 6-1 Displaying an activity sheet on iPhone

```
- (void)displayActivityControllerWithDataObject:(id)obj {
   UIActivityViewController* vc = [[UIActivityViewController alloc]
                                initWithActivityItems:@[obj]
applicationActivities:nil];
    [self presentViewController:vc animated:YES completion:nil];
}
```

For more information about using the activity view controller, see *UIActivityViewController Class Reference*. For a complete list of activities and the data types they support, see *UIActivity Class Reference*.

Receiving Files and Data Sent to Your App

To receive files sent to your app using AirDrop, do the following:

- In Xcode, declare support for the document types your app is capable of opening.
- In your app delegate, implement the `application:openURL:sourceApplication:annotation:` method. Use that method to receive the data that was sent by the other app.

- Be prepared to look for files in your app's `Documents/Inbox` directory and move them out of that directory as needed.

The Info tab of your Xcode project contains a Document Types section for specifying the document types your app supports. At a minimum, you must specify a name for your document type and one or more UTIs that represent the data type. For example, to declare support for PNG files, you would include `public.png` as the UTI string. iOS uses the specified UTIs to determine if your app is eligible to open a given document.

After transferring an eligible document to your app's container, iOS launches your app (if needed) and calls the `application:openURL:sourceApplication:annotation:` method of its app delegate. If your app is in the foreground, you should use this method to open the file and display it to the user. If your app is in the background, you might decide only to note that the file is there so that you can open it later. Because files transferred via AirDrop are encrypted using data protection, you cannot open files unless the device is currently unlocked.

Files transferred to your app using AirDrop are placed in your app's `Documents/Inbox` directory. Your app has permission to read and delete files in this directory but it does not have permission to write to files. If you plan to modify the file, you must move it out of the `Inbox` directory before doing so. It is recommended that you delete files from the `Inbox` directory when you no longer need them.

For more information about supporting document types in your app, see *Document-Based App Programming Guide for iOS*.

Using URL Schemes to Communicate with Apps

A URL scheme lets you communicate with other apps through a protocol that you define. To communicate with an app that implements such a scheme, you must create an appropriately formatted URL and ask the system to open it. To implement support for a custom scheme, you must declare support for the scheme and handle incoming URLs that use the scheme.

> **Note:** Apple provides built-in support for the `http`, `mailto`, `tel`, and `sms` URL schemes among others. It also supports `http`-based URLs targeted at the Maps, YouTube, and iPod apps. The handlers for these schemes are fixed and cannot be changed. If your URL type includes a scheme that is identical to one defined by Apple, the Apple-provided app is launched instead of your app. For information about the schemes supported by apple, see *Apple URL Scheme Reference*.

Sending a URL to Another App

When you want to send data to an app that implements a custom URL scheme, create an appropriately formatted URL and call the `openURL:` method of the app object. The `openURL:` method launches the app with the registered scheme and passes your URL to it. At that point, control passes to the new app.

The following code fragment illustrates how one app can request the services of another app ("todolist" in this example is a hypothetical custom scheme registered by an app):

```
NSURL *myURL = [NSURL URLWithString:@"todolist://www.acme.com?
Quarterly%20Report#200806231300"];

[[UIApplication sharedApplication] openURL:myURL];
```

If your app defines a custom URL scheme, it should implement a handler for that scheme as described in Implementing Custom URL Schemes. For more information about the system-supported URL schemes, including information about how to format the URLs, see *Apple URL Scheme Reference*.

Implementing Custom URL Schemes

If your app can receive specially formatted URLs, you should register the corresponding URL schemes with the system. Apps often use custom URL schemes to vend services to other apps. For example, the Maps app supports URLs for displaying specific map locations.

Registering Custom URL Schemes

To register a URL type for your app, include the `CFBundleURLTypes` key in your app's `Info.plist` file. The `CFBundleURLTypes` key contains an array of dictionaries, each of which defines a URL scheme the app supports. Table 6-1 describes the keys and values to include in each dictionary.

Table 6-1 Keys and values of the `CFBundleURLTypes` property

Key	Value
`CFBundleURLName`	A string containing the abstract name of the URL scheme. To ensure uniqueness, it is recommended that you specify a reverse-DNS style of identifier, for example, `com.acme.myscheme`. The string you specify is also used as a key in your app's `InfoPlist.strings` file. The value of the key is the human-readable scheme name.
`CFBundleURLSchemes`	An array of strings containing the URL scheme names—for example, `http`, `mailto`, `tel`, and `sms`.

Note: If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme.

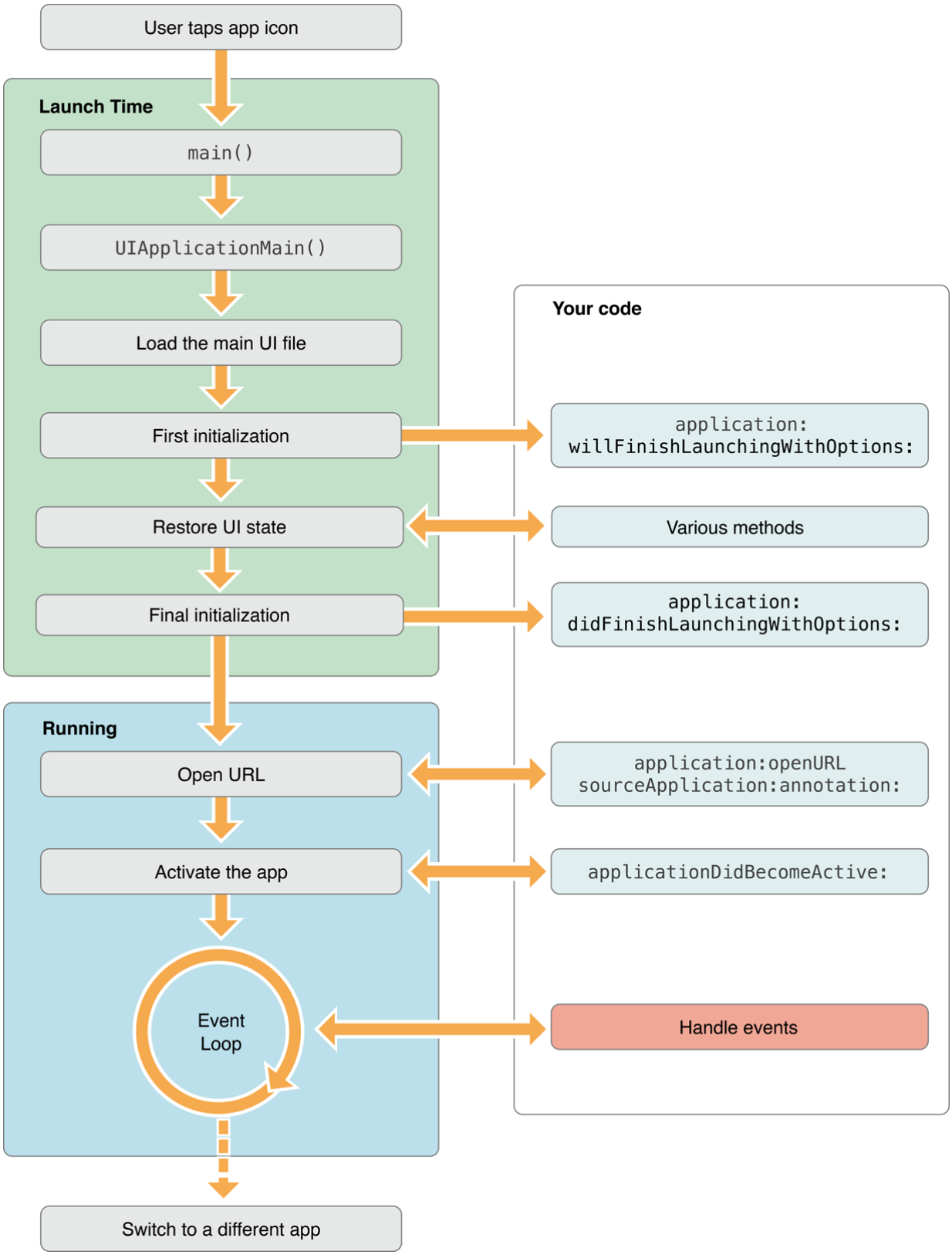
Handling URL Requests

An app that has its own custom URL scheme must be able to handle URLs passed to it. All URLs are passed to your app delegate, either at launch time or while your app is running or in the background. To handle incoming URLs, your delegate should implement the following methods:

- Use the `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods to retrieve information about the URL and decide whether you want to open it. If either method returns `NO`, your app's URL handling code is not called.
- Use the `application:openURL:sourceApplication:annotation:` method to open the file.

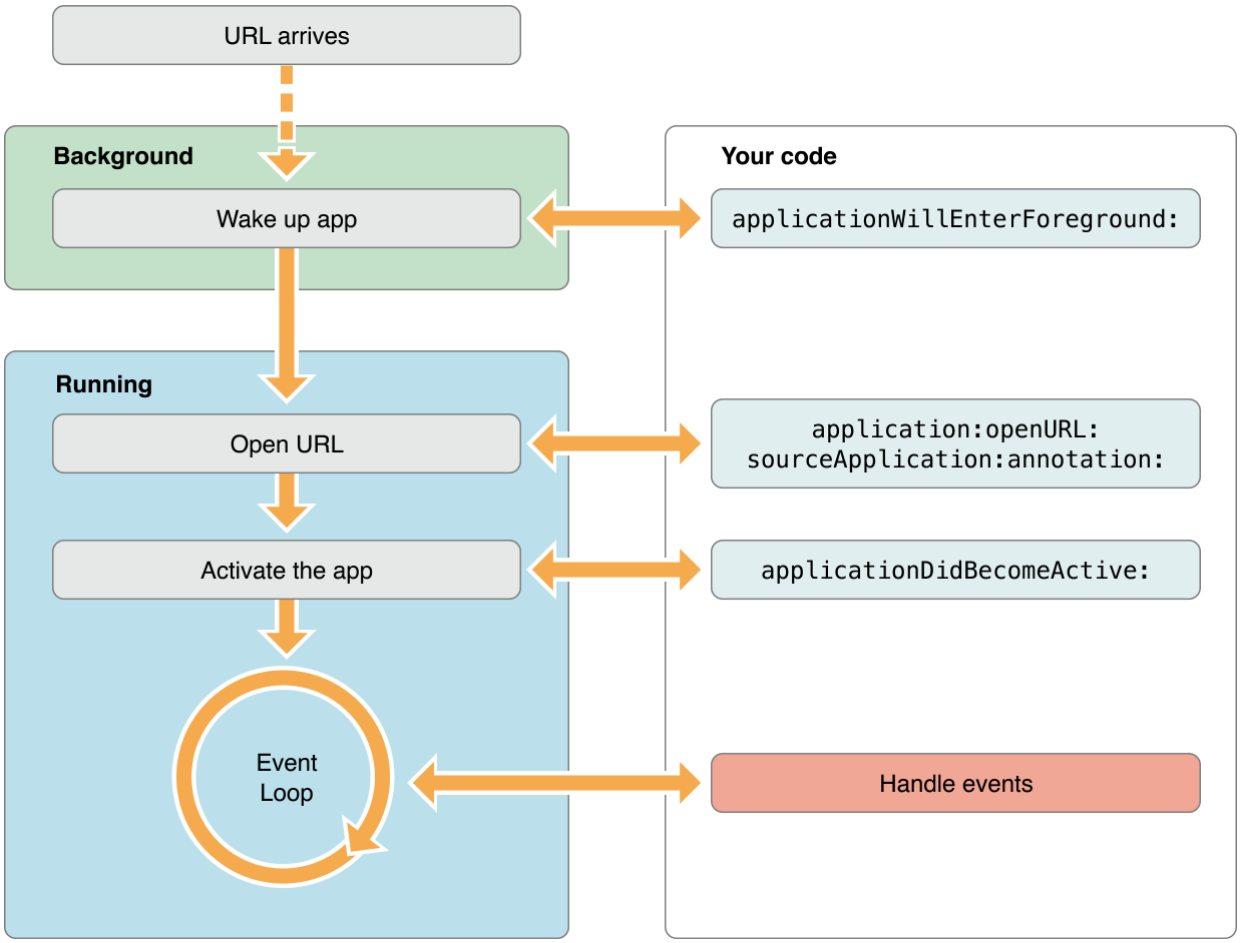
If your app is not running when a URL request arrives, it is launched and moved to the foreground so that it can open the URL. The implementation of your `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` method should retrieve the URL from its options dictionary and determine whether the app can open it. If it can, return `YES` and let your `application:openURL:sourceApplication:annotation:` (or `application:handleOpenURL:`) method handle the actual opening of the URL. (If you implement both methods, both must return `YES` before the URL can be opened.) Figure 6-1 shows the modified launch sequence for an app that is asked to open a URL.

Figure 6-1 Launching an app to open a URL



If your app is running but is in the background or suspended when a URL request arrives, it is moved to the foreground to open the URL. Shortly thereafter, the system calls the delegate's `application:openURL:sourceApplication:annotation:` to check the URL and open it. Figure 6-2 shows the modified process for moving an app to the foreground to open a URL.

Figure 6-2 Waking a background app to open a URL



Note: Apps that support custom URL schemes can specify different launch images to be displayed when launching the app to handle a URL. For more information about how to specify these launch images, see Displaying a Custom Launch Image When a URL is Opened.

All URLs are passed to your app in an `NSURL` object. It is up to you to define the format of the URL, but the `NSURL` class conforms to the RFC 1808 specification and therefore supports most URL formatting conventions. Specifically, the class includes methods that return the various parts of a URL as defined by RFC 1808, including the user, password, query, fragment, and parameter strings. The "protocol" for your custom scheme can use these URL parts for conveying various kinds of information.

In the implementation of `application:openURL:sourceApplication:annotation:` shown in Listing 6-2, the passed-in URL object conveys app-specific information in its query and fragment parts. The delegate extracts this information—in this case, the name of a to-do task and the date the task is due—and with it creates a model object of the app. This example assumes that the user is using a Gregorian calendar. If your app supports non-Gregorian calendars, you need to design your URL scheme accordingly and be prepared to handle those other calendar types in your code.

Listing 6-2 Handling a URL request based on a custom scheme

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
        sourceApplication:(NSString *)sourceApplication annotation:(id)annotation {
    if ([[url scheme] isEqualToString:@"todolist"]) {
        ToDoItem *item = [[ToDoItem alloc] init];
        NSString *taskName = [url query];
        if (!taskName || ![self isValidTaskString:taskName]) { // must have a task
name
            return NO;
        }
```

```
        taskName = [taskName
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];


        item.toDoTask = taskName;
        NSString *dateString = [url fragment];
        if (!dateString || [dateString isEqualToString:@"today"]) {
            item.dateDue = [NSDate date];
        } else {
            if (![self isValidDateString:dateString]) {
                return NO;
            }
            // format: yyyymmddhhmm (24-hour clock)
            NSString *curStr = [dateString substringWithRange:NSMakeRange(0, 4)];
            NSInteger yeardigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(4, 2)];
            NSInteger monthdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(6, 2)];
            NSInteger daydigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(8, 2)];
            NSInteger hourdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(10, 2)];
            NSInteger minutedigit = [curStr integerValue];

            NSDateComponents *dateComps = [[NSDateComponents alloc] init];
            [dateComps setYear:yeardigit];
            [dateComps setMonth:monthdigit];
            [dateComps setDay:daydigit];
            [dateComps setHour:hourdigit];
            [dateComps setMinute:minutedigit];
            NSCalendar *calendar = [s[NSCalendar alloc]
initWithCalendarIdentifier:NSGregorianCalendar];
            NSDate *itemDate = [calendar dateFromComponents:dateComps];
            if (!itemDate) {
                return NO;
            }
            item.dateDue = itemDate;
        }


        [(NSMutableArray *)self.list addObject:item];
        return YES;
    }
    return NO;
}
```

Be sure to validate the input you get from URLs passed to your app; see Validating Input and Interprocess Communication in *Secure Coding Guide* to find out how to avoid problems related to URL handling. To learn about URL schemes defined by Apple, see *Apple URL Scheme Reference*.

Displaying a Custom Launch Image When a URL is Opened

Apps that support custom URL schemes can provide a custom launch image for each scheme. When the system launches your app to handle a URL and no relevant snapshot is available, it displays the launch image you specify. To specify a launch image, provide a PNG image whose name uses the following naming conventions:

<basename>`-`*<url_scheme>*`<other_modifiers>.png`

In this naming convention, basename represents the base image name specified by the `UILaunchImageFile` key in your app's `Info.plist` file. If you do not specify a custom base name, use the string `Default`. The *<url_scheme>* portion of the name is your URL scheme name. To specify a generic launch image for the `myapp` URL scheme, you would include an image file with the name `Default-myapp@2x.png` in the app's bundle. (The @2x modifier signifies that the image is intended for Retina displays. If your app also supports standard resolution displays, you would also provide a `Default-myapp.png` image.)

For information about the other modifiers you can include in launch image names, see the description of the `UILaunchImageFile` name key in *Information Property List Key Reference*.
