Services Implementation Guide

Data Management: Event Handling

2009-05-18

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction

Note: This document was previously titled *System Services*.

Services are features exported by your application for the benefit of other applications. Services let you share the resources and capabilities of your application with other applications in the system.

Users access services through the Services menu that's found in every application's application menu. An application does not need to know in advance what operations are available; the application merely needs to indicate the types of data it uses. The Services menu will make available the operations that apply to those types when they apply.

This document describes how Mac OS X services work, shows some typical Services menus, and provides instructions on how you can use services in your application.

Who Should Read This Document?

You should read this document if you are a Cocoa application developer and want to provide your application's services to other applications or make services from other applications available to your application.

Prerequisites

Before you read this document, you should be familiar with information property lists. You need to know what they are and how to add properties to a list. For more information, see Information Property List Files in *Runtime Configuration Guidelines*.

For guidelines on naming menu items and for designing the interface for a services application, see *Apple Human Interface Guidelines*.

Organization of This Document

Read the first three chapters to learn how services work, see examples of services in applications, and learn which properties you use to provide and use services in your applications. The remaining two chapters describe in detail how to provide and use services in your applications.

Changes for Mac OS X v10.6

The Services feature was updated in Mac OS X version 10.6 with the following changes and additions to properties:

- A slash is no longer treated as specifying a submenu with `NSMenuItem`.
- `NSSendTypes` and `NSReturnTypes` no longer need to be specified.
- There are three new properties: `NSSendFileTypes`, `NSServiceDescription`, and `NSRequiredContext`.

Services Overview

Services allow a user to access the functionality of one application from within another application. An application that provides a service advertises the operations it can perform on a particular type of data—for example, encryption of text, optical character recognition of a bitmapped image, or generating text such as a message of the day. When the user is manipulating that particular type of data in some application, the user can choose the appropriate item in the Services menu to operate on the current data selection (or merely insert new data into the document).

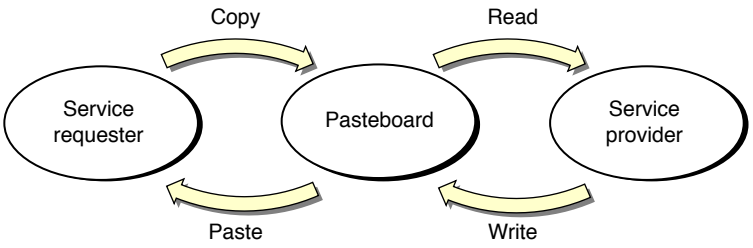
This chapter discusses how services are processed and describes some sample services.

How Service Requests Work

Services are performed by transferring data back and forth between applications through a shared pasteboard. Note that the two applications—service requester and service provider—are completely separate; they do not run in a shared memory space. The pasteboard holding the data is specific to the service request and does not normally interfere with the standard Copy/Paste pasteboard.

When the user chooses a Services menu item, data flows as shown in Figure 1. The current selection is copied to a pasteboard which is then passed to the service provider application. If the service provider is not currently running, it is automatically launched. The service provider reads the contents of the pasteboard and operates on it. The service provider writes new data back to the pasteboard and the pasteboard is returned to the original application. The original application then pastes the pasteboard's contents into the document, replacing the current selection, if there is one. The service provider application does not automatically quit at the end of the service request.

Figure 1 Data flow in a service request



Not all services both receive and provide data. Some services only receive data and others only provide data. In these cases only one of the copy and paste steps is performed. Services can thus be divided into two groups:

- **Processor.** This type of service acts on data. A processor service acts on the current selection and then sends it to the service. For example, if a user selects an email address in a TextEdit document, and then chooses Send Selection from the Services menu, TextEdit copies the person's address to the pasteboard, the Mail application launches, and Mail pastes the address into the Send field of a new email message.

- **Provider.** This type of service gives data to the calling application. For example, if a user chooses Capture Full Screen from the Services menu, the Grab application opens, takes a screen shot, then returns the screen shot (TIFF data in this case) to the calling application. The calling application (such as TextEdit) is responsible for pasting the data into the active document.

A service falls into both categories if it processes the current selection and then provides a replacement value. For example, a text encryption service takes the current text selection, encrypts it, and then returns the encrypted text to the service requester to replace the current selection.

Sample Services

The following figures show services in action. Figure 2 shows the Services menu from the TextEdit application. Make New Sticky Note is an example of a processor service. The Make New Sticky Note command takes the current selection in the TextEdit document, opens a new Stickies document, and then pastes the selection into the Stickies document. For more convenient use, a keyboard shortcut (Command-Shift-Y) is defined for this service.

Figure 2 Make New Sticky Note is a processor service

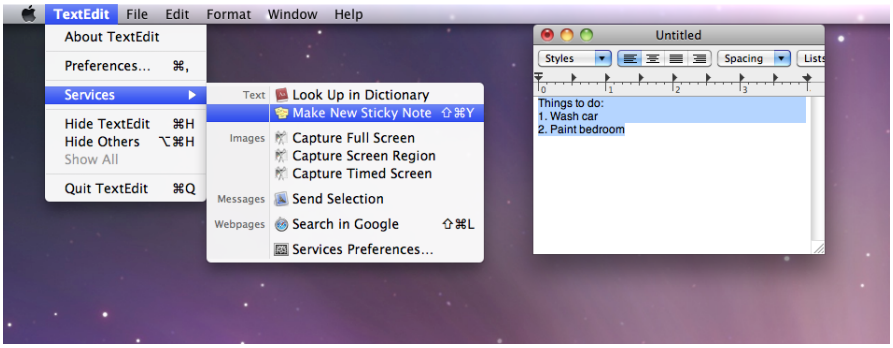
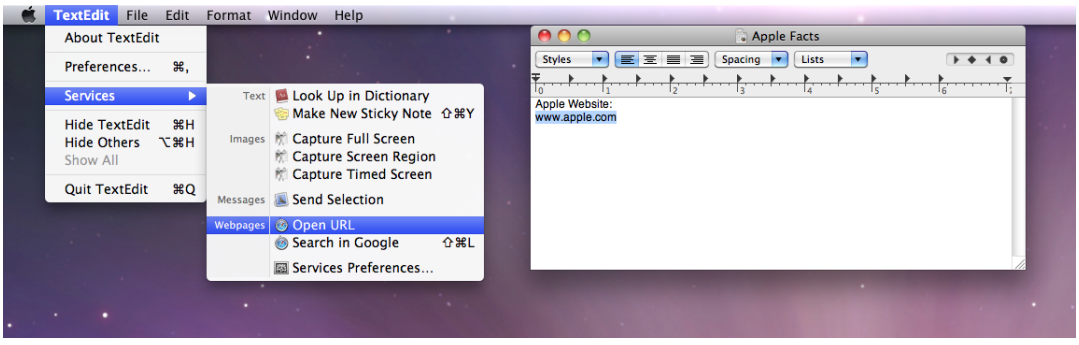


Figure 3 shows another example of a processor service. In this case, the Open URL command copies the selected text, launches a Web browser, pastes the selected text into the browser's location field, and then tries to connect to that location.

Figure 3 Open URL is a processor service



Capture Full Screen is a provider service. Figure 4 shows the Apple Facts document before Capture Full Screen is invoked.

Figure 4 Capture Full Screen is a provider service

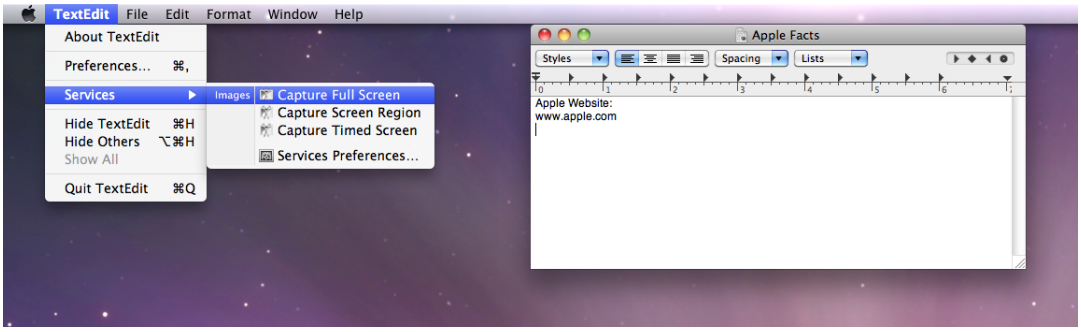
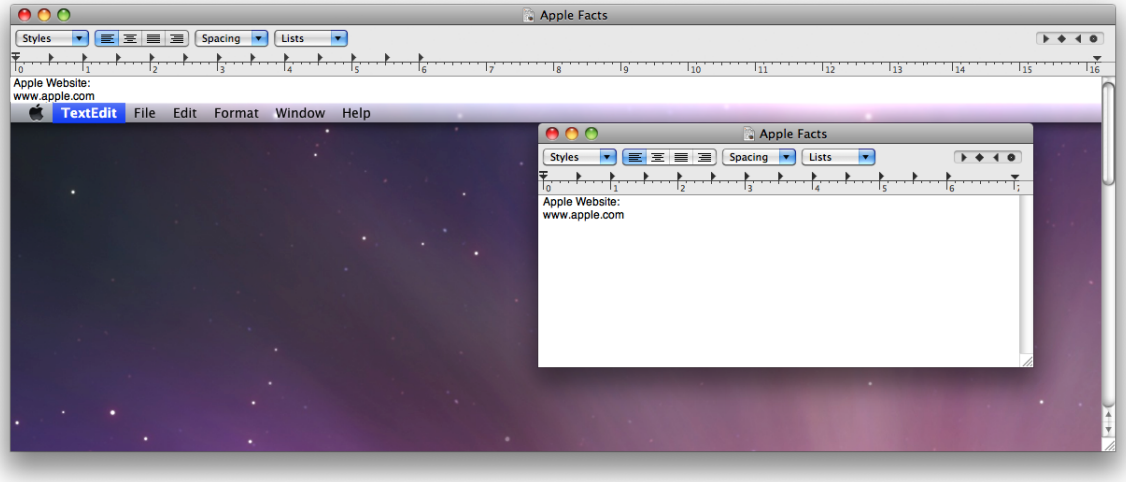


Figure 5 shows the Apple Facts document after Grab has taken a shot of the current screen and returned the data to the TextEdit application. Recall that it is the responsibility of TextEdit to do something with the returned data. In this example, TextEdit simply pastes the TIFF into the current document at the insertion point.

Figure 5 The Apple Facts document after a screen shot has been inserted



Items in the Services Menu

Applications that provide services may be installed anywhere on the system. The applications' information property lists declare the services the applications provide (see "Services Properties" (page 15)). Mac OS X collects the property list information and uses it to populate the items in the Services menu based on the particular data types supported by each application.

The Services menu is included in the default nib file created by Xcode and Interface Builder for Cocoa applications. If the application's menu is instead created programmatically, you need to designate a Services menu using the `NSApplication` method `setServicesMenu:`. If an application registers for services (see "Using Services" (page 23)), the appropriate items are automatically available in the Services menu.

The items in the Services menu are commands categorized by what type of data they operate on. The application icon of the application providing a service appears to the left of the service's name in the menu. If two or more applications provide a service with an identical name, the name of the application providing each one is appended in parentheses after the name of the service in the menu.

The Services menu is populated when the menu is opened. Choosing the Services menu causes the current responder chain to be searched for objects that can provide or receive data of the types used by each service listed in the Services menu. If an object is found that can use a given service, the service's menu item is shown in the menu. Menu items for which no suitable object is found are not shown in the menu.

Services Properties

Any application that has one or more services to provide must advertise the type of data its services can handle. Services are advertised through the `NSServices` property of the application's information property list (`Info.plist`) file.

Note: The information property list (`Info.plist`) contains key-value pairs that specify an application's properties that are of interest to the Finder and other applications. Although the `Info.plist` is a text file that uses XML (Extensible Markup Language) format, you should not modify the XML directly unless you are very familiar with XML syntax. Instead, use Xcode or the Property List Editor application provided with Mac OS X to modify the `Info.plist` file. You can find more information on property lists in *Runtime Configuration Guidelines*.

Property Definitions

`NSServices` is a property whose value is an array of dictionaries that specifies the services provided by the application. Keys for each dictionary entry, are as follows:

- `NSMessage` indicates the instance method to invoke. Its value is used to construct an Objective-C method of the form *`messageName`*`:userData:error:`. This message is sent to the application's service provider object.
- `NSPortName` is the name of the port on which the application should listen for service requests. Its value depends on how the service provider application is registered. In most cases, this is the application name. This property is ignored for Automator workflows being used as services.
- `NSMenuItem` is a dictionary that specifies the text of the Services menu item. Only one entry should be in this dictionary, and its key should be `default`. Its string value is used as the menu item's text. There are no submenus in the Services menu, so the text preceding and including any slash present is discarded. (In Mac OS X version 10.5 and earlier, you could use a slash to specify a submenu. For example, `Mail/Send Selection` appears in the Services menu as a submenu named Mail with an item named Send Selection.)

  Any services with identical names are disambiguated by adding the application name providing each service in parentheses after its name. (In Mac OS X version 10.5 and earlier, `NSMenuItem` must be unique, as only one is used in the Services menu if there are duplicates.)

  To localize the string specified, create a `ServicesMenu.strings` file for each localization in your bundle with the above `default` menu item string as the lookup key. For example, create a `.strings` file that has `Send Selection` as the key and the localized text as its value. (See *Resource Programming Guide* for details on localized strings files.) If a localized string is not found, the `default` text is used.
- `NSKeyEquivalent` is an optional dictionary that specifies the keyboard equivalent for invoking the menu command. Like `NSMenuItem`, the only entry in the dictionary should have the key `default` with a string value that can be localized in the `ServicesMenu.strings` file. The string value must be a single character. The keyboard shortcut is this one character, with the Command key modifier. If the character is uppercase, the Shift modifier is also used.

Use key equivalents sparingly. Remember that your shortcuts are being added to the collection of shortcuts defined by each application as well as defined by the other services. When an application already has a shortcut with that key equivalent, the application's shortcut wins. If multiple services define the same shortcut, which one gets invoked is undefined.

- `NSSendTypes` is an optional array that contains data type names. Send types are the types sent from the application requesting a service. *NSPasteboard Class Reference* lists several common data types. Additionally, in Mac OS X version 10.5 and later, Uniform Type Identifiers may be used. (See *Uniform Type Identifiers Overview* for more information on Uniform Type Identifiers.) This key is not required. (In Mac OS X version 10.5 and earlier, an application that provides a service must specify `NSSendTypes`, `NSReturnTypes`, or both.)
- `NSReturnTypes` is an optional array that contains data type names. Return types are the data types returned to the application requesting a service. The NSPasteboard class description lists several common data types. Additionally, in Mac OS X version 10.5 and later, Uniform Type Identifiers may be used. (See *Uniform Type Identifiers Overview* for more information on Uniform Type Identifiers.) This key is not required. (In Mac OS X version 10.5 and earlier, an application that provides a service must specify `NSSendTypes`, `NSReturnTypes`, or both.)
- `NSUserData` is an optional string that contains a value of your choice. You can use this string to customize the behavior of your service. For example, if your application provides several similar services, you can have the same `NSMessage` value for all of them (each service invokes the same method) and use different `NSUserData` values to distinguish between them. This entry is also useful for applications that provide open-ended, or add-on, services. This property is ignored for Automator workflows being used as services.
- `NSTimeout` is an optional numerical string that indicates the number of milliseconds that services should wait for a response from the application providing a service when a response is required. If the wait time exceeds the timeout value, the application aborts the service request and continues without interruption. If you don't specify this entry, the timeout value is 30000 milliseconds (30 seconds).

  Users may also press the Escape key or type Command-period to cancel.
- `NSSendFileTypes` is an array that contains file type names. Only Uniform Type Identifiers are allowed; pasteboard types are not permitted. (See *Uniform Type Identifiers Overview* for more information on Uniform Type Identifiers.) By assigning a value to this key, your service declares that it can operate on files whose type conforms to one or more of the given file types. Your service will receive a pasteboard from which you can read file URLs. You may specify values for both `NSSendTypes` and `NSSendFileTypes` if your service can operate on both pasteboard data and files.
- `NSServiceDescription` is a string that contains a description of your service that is suitable for presentation to users. It can be localized via the `ServicesMenu.strings` file.

  The description may be long. If your description is long, the value of `NSServiceDescription` should be a short token, such as `SERVICE_DESCRIPTION`. The full text should be given in the `ServicesMenu.strings` file with that token as its key.
- `NSRequiredContext` is a dictionary that can be used to limit when the service appears. Through judicious use of `NSRequiredContext`, you can ensure that your service appears only when it applies and does not clutter the Services menu when it is not applicable.

  `NSRequiredContext` may be a dictionary that contains any of the following keys, all of which are optional. It may also be an array of such dictionaries, in which case the service is enabled if any of the given contexts is satisfied.

  ```
  NSApplicationIdentifier
  NSTextScript
  NSTextLanguage
  NSWordLimit
  ```

```
NSTextContent
```

These keys have the following significance:

- `NSApplicationIdentifier` is a bundle ID as a string or an array of such IDs. Your service will appear only if the bundle ID of the current application matches one of the given bundle IDs. For example, you could use this to limit a service to appear only in Xcode or in the Finder.
- `NSTextScript` is a string containing a standard four-letter script tag, such as `Latn` or `Cyrl`. It may also be an array of such strings. The service will only appear if the dominant script of the selected text is text corresponding to one of the given script tags. This key is relevant only for services that accept text.
- `NSTextLanguage` is a string containing the required overall language of the selected text as a BCP-47 tag. It may also be an array of such strings. The service appears only if the BCP-47 tag of the overall language of the selected text matches one of the given tags.

  The matching is performed via the default language range matching scheme, which is a prefix-matching scheme. For example, the `NSTextLanguage` string "zh" matches text whose language is "zh-Hant". This key is relevant only for services that accept text.
- `NSWordLimit` is an integer representing the maximum number of selected words that the service operates on. For example, a service to look up a stock by ticker symbol might have an `NSWordLimit` of 1 because ticker symbols cannot contain spaces. It may not be an array. This key is relevant only for services that accept text.
- `NSTextContent` is a string specifying a data type that the text must contain, or an array of such strings. The valid values are as follows:

  ```
  URL
  Date
  Address
  Email
  FilePath
  ```

  The service is available only if the selected text contains one of the specified data types. All text selected is provided to the service-vending application, not just the parts found to contain the given data types.

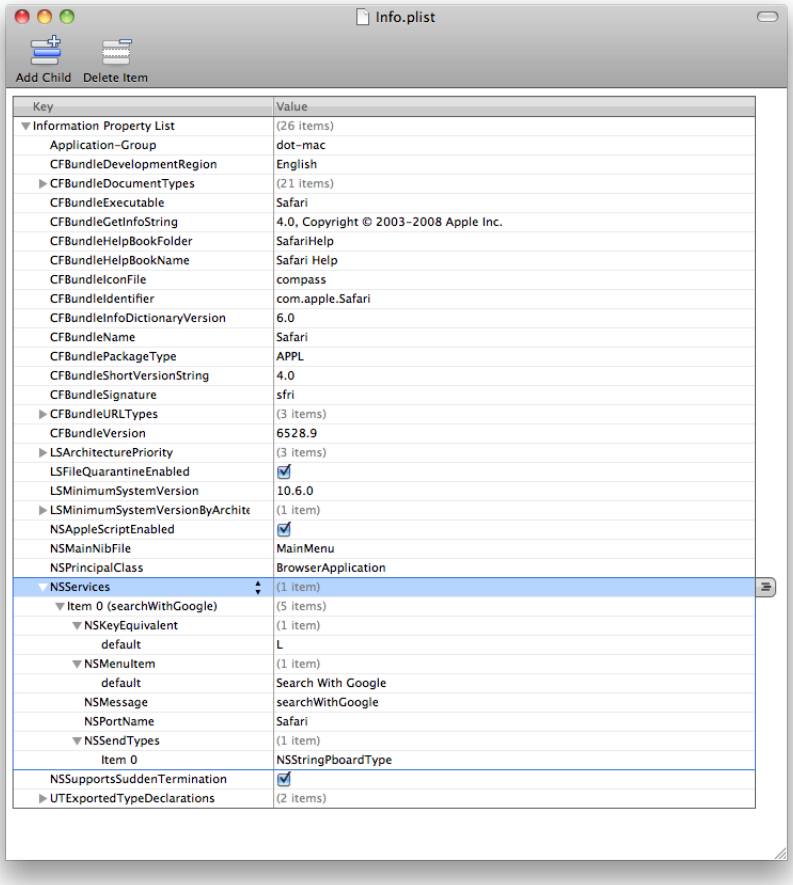
Add-On Services

You typically define services when you create your application and advertise them in the `Info.plist` file of the application's bundle. The Services facility also allows you to advertise services outside of the application bundle, enabling you to create "add-on" services after the fact. This is where the `NSUserData` entry becomes truly useful: you can define a single message in your application that performs actions based on the user data provided, such as running the user data string as a UNIX command or treating it as a special argument in addition to the selected data that gets sent through the pasteboard. To define an add-on service, you create a bundle with a `.service` extension that contains an `Info.plist` file, which in turn contains the add-on service's `NSServices` property. The property uses the application's `NSMessage` and `NSPortName` values.

Sample Property List

The `NSServices` property for Safari is shown in Figure 1 as it appears in the Property List Editor application.

Figure 1 The `NSServices` property for Safari



The `NSServices` property has one entry that represents the only service offered by Safari: “Search with Google.” Note that, for this entry, the port name is Safari. As mentioned, the port name is usually the application name.

The entry has one return type, `NSStringPboardType`. An application can have more than one return type per entry, and the return types don’t necessarily need to be the same for each entry. Both Uniform Type Identifiers and pasteboard types are valid here. (For more information on Uniform Type Identifiers, see *Uniform Type Identifiers Overview*.)

The entry has a key equivalent of L, which means that Command-L can be used to invoke the service.

Providing a Service

Providing a service consists of the following steps:

1. Deciding what the service will do
2. Implementing the service method
3. Registering the service provider
4. Advertising the service by adding it to your application's property list file
5. Installing the service

Deciding on a Service

For the purposes of this chapter, suppose you have decided to work on a program to read USENET news and have an object with a method to encrypt and decrypt articles, such as the one in Listing 1. News articles containing offensive material are often encrypted with this algorithm, called ROT13, in which letters are shifted halfway through the alphabet. Since this feature is generally useful as a simple encryption scheme, it can be exported to other applications.

Listing 1 Text encryption method

```
- (NSString *)rotateLettersInString:(NSString *)aString {
    NSString *newString;
    unsigned length;
    unichar *buf;
    unsigned i;

    length = [aString length];
    buf = malloc( (length + 1) * sizeof(unichar) );
    [aString getCharacters:buf];
    buf[length] = (unichar)0; // not really needed....
    for (i = 0; i < length; i++) {
        if (buf[i] >= (unichar)'a' && buf[i] <= (unichar) 'z') {
            buf[i] += 13;
            if (buf[i] > 'z') buf[i] -= 26;
        } else if (buf[i] >= (unichar)'A' && buf[i] <= (unichar) 'Z') {
            buf[i] += 13;
            if (buf[i] > 'Z') buf[i] -= 26;
        }
    }
    newString = [NSString stringWithCharacters:buf length:length];
    free(buf);
    return newString;
}
```

Implementing the Service Method

To offer the encryption facility as a service, write a method such as the one in Listing 2.

Listing 2 Service method

```
- (void)simpleEncrypt:(NSPasteboard *)pboard
    userData:(NSString *)userData error:(NSString **)error {

     // Test for strings on the pasteboard.
     NSArray *classes = [NSArray arrayWithObject:[NSString class]];
     NSDictionary *options = [NSDictionary dictionary];

     if (![pboard canReadObjectForClasses:classes options:options]) {
          *error = NSLocalizedString(@"Error: couldn't encrypt text.",
               @"pboard couldn't give string.");
          return;
     }

     // Get and encrypt the string.
     NSString *pboardString = [pboard stringForType:NSPasteboardTypeString];
     NSString *newString = [self rotateLettersInString:pboardString];
     if (!newString) {
          *error = NSLocalizedString(@"Error: couldn't encrypt text.",
               @"self couldn't rotate letters.");
          return;
     }

     // Write the encrypted string onto the pasteboard.
     [pboard clearContents];
     [pboard writeObjects:[NSArray arrayWithObject:newString]];
}
```

The method providing the service is of the form *messageName*`:userData:error:` and takes the values shown in Listing 2. The method itself takes data from the pasteboard as needed, operates on it, and writes any results back to the pasteboard. In case of an error, the method simply sets the pointer given by the error argument to a non-nil `NSString` and returns. The error message is logged to the console. The *`userData`* parameter is not used here.

Now you have an object with methods that allow it to perform a service for another application. Next, you need to register the object at run time so the services facility knows which object to have perform the service.

Registering the Service Provider

You create and register your object in the `applicationDidFinishLaunching:` application delegate method (or equivalent) with the `setServicesProvider:` method of the `NSApplication` class. If your object is called `encryptor`, you create and register it with this code fragment:

```
EncryptoClass *encryptor;
encryptor = [[EncryptoClass alloc] init];
[NSApp setServicesProvider:encryptor];
```

If you are writing a Foundation tool, which lacks an `NSApplication` object, register the service object with the `NSRegisterServicesProvider` function. Its declaration is the following:

```
void NSRegisterServicesProvider(id provider, NSString *portName)
```

where *`provider`* is the object that provides the services, and *`portName`* is the same value you specify for the `NSPortName` entry in the `NSServices` property of the application's information property list (`Info.plist`) file. After making this function call, you must enter the run loop to respond to service requests.

You can register only one service provider per application. If you have more than one service to provide, a single object must provide the interface to all of the services.

Service requests can arrive immediately after you register the object, in some circumstances even before exiting `applicationDidFinishLaunching:`. Therefore, register your service provider only when you are completely ready to process requests.

Advertising the Service

For the system to know that your application provides a service, you must advertise that fact. You do this by adding an entry to your application project's `Info.plist` file as described in "Services Properties" (page 15). The entry you add is called the service specification. In our example, the `NSServices` property looks like this:

```
<key>NSServices</key>
<array>
    <dict>
        <key>NSKeyEquivalent</key>
        <dict>
            <key>default</key>
            <string>E</string>
        </dict>
        <key>NSMenuItem</key>
        <dict>
            <key>default</key>
            <string>Encrypt Text</string>
        </dict>
        <key>NSMessage</key>
        <string>simpleEncrypt</string>
        <key>NSPortName</key>
        <string>NewsReader</string>
        <key>NSSendTypes</key>
        <array>
            <string>NSPasteboardTypeString</string>
        </array>
        <key>NSReturnTypes</key>
        <array>
            <string>NSPasteboardTypeString</string>
        </array>
    </dict>
</array>
```

Installing the Service

A service can be offered as part of an application, such as Mail, or as a standalone service—one without a user interface that is intended for use only in the Services menu.

- To build an application that offers a service, use the extension `.app` and install it in the `Applications` folder (or a subfolder).
- To build a standalone service, use the extension `.service` and store it in `Library/Services`.

In either case, you should install it in one of the four file-system domains—System, Network, Local, and User. (See File-System Domains in *File System Overview* for details.)

The list of available services on the computer is built each time a user logs in. If your service is installed in an `Applications` directory, you need to log out and log back in before the service becomes available. If it's installed in a `Library/Services` directory, this is not necessary. You can force an update of the list of services without logging out by calling the following function:

```
void NSUpdateDynamicServices(void)
```

Testing

When you test your program, it may be useful to trigger an immediate rescan of Services, as if `NSUpdateDynamicServices` had been called. You can do this from the command line using the `pbs` tool:

```
/System/Library/CoreServices/pbs
```

It may also be useful to be sure that the system has recognized the Service. You can also use the `pbs` tool to list the registered Services by specifying the `dump_pboard` option:

```
/System/Library/CoreServices/pbs -dump_pboard
```

If your Service is recognized by `pbs`, but does not appear in the Services menu, you can use the `NSDebugServices` user default to help determine why. For example, to determine why Mail's Services appear or do not appear in TextEdit, you can launch TextEdit with the `NSDebugServices` option and pass it the bundle identifier of Mail:

```
/Applications/TextEdit.app/Contents/MacOS/TextEdit -NSDebugServices com.apple.mail
```

When the Services menu is opened, it will log information to the console about why the service does or does not appear.

Important: The `pbs` tool is for debugging purposes only. It does not have a guaranteed interface, and may even be removed in a future version of Mac OS X. You should not design any programs to depend on it.

Using Services

The default nib file created for new Cocoa applications contains a Services menu in the application menu, so there is nothing else you need to do for your application to work with the Services facility; your application automatically has access to all appropriate services provided by other applications. If you need to construct menus programmatically, you simply designate the `NSMenu` object that you want as your Services menu with the `setServicesMenu:` method of `NSApplication`.

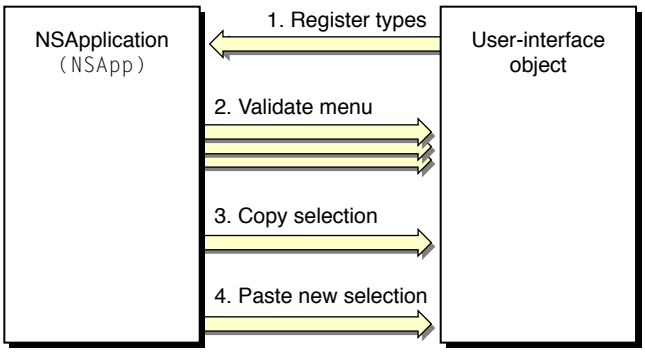
The Process

If you subclass `NSView` or `NSWindow` (or any other subclass of `NSResponder`), you need to implement it such that it interacts properly with the Services facility. Tying custom views or windows into the Services facility entails the following steps:

1. Registering your user-interface objects for services
2. Validating the Services menu items for the current selection
3. Sending the current selection to the service
4. Receiving data from the service to replace the current selection

The steps for using services are illustrated in Figure 1.

Figure 1 Using services



When a pure provider service is invoked (in other words, a service with no send types), step 3 is skipped. When a pure processor service is invoked (in other words, a service with no return types), step 4 is skipped.

The following sections cover each of these steps. A final section, "Invoking a Service Programmatically" (page 26), shows how to invoke a service in your code.

Registering Objects for Services

The Services menu does not contain every service offered by other applications. For example, in a text editor a service to invert a bitmapped image is of no use and should not be offered. Which services appear in the Services menu is determined by the data types that the objects in the application—specifically, the `NSResponder` objects—can send and receive through the pasteboard.

An `NSResponder` object registers these data types using the `NSApplication` Objective-C method `registerServicesMenuSendTypes:returnTypes:`. Objects in the AppKit framework already do this for the basic text services, but your custom `NSResponder` subclass must do this to expand the list. A convenient location is in your subclass's `initialize` class method, which is guaranteed to be invoked by the runtime before any other method of the class. All types used by instances of the class must be registered, even if they are not always available; Services menu items are either present or not present based on what is available at the moment, as described in "Validating Services Menu Items" (page 24).

An object does not have to register the same types for both sending and receiving. For example, suppose you are writing a rich text editor that can send unformatted and rich text, but can receive only unformatted text. Here is a portion of the initialization method for a text-editor's `NSView` subclass:

```
+ (void)initialize
{
    static BOOL initialized = NO;
    /* Make sure code only gets executed once. */
    if (initialized == YES) return;
    initialized = YES;

    sendTypes = [NSArray arrayWithObjects:NSStringPboardType,
                    NSRTFPboardType, nil];
    returnTypes = [NSArray arrayWithObjects:NSStringPboardType,
                    nil];
    [NSApp registerServicesMenuSendTypes:sendTypes
                    returnTypes:returnTypes];
}
```

Your `NSResponder` object can register any pasteboard data type, public or proprietary, common or rare. If it handles the public and common types, of course, it has access to more services. For a list of standard pasteboard data types, see *NSPasteboard Class Reference*.

Validating Services Menu Items

While your application is running, various types of data can be selected and available for transfer on the pasteboard. If a service does not apply to the type of the selected data, its menu item needs to be disabled. To check whether a service applies, the application object sends `validRequestorForSendType:returnType:` messages to Objective-C objects in the responder chain to see whether they have data of the type used by that service. While the Services menu is visible, this method is called frequently—typically many times per event—to ensure that the menu items for all service providers are properly enabled: it is sent for each combination of send and return types supported by each service and possibly for many objects in the responder chain. Because this method is called frequently, it must be fast so that event handling does not fall behind the user's actions.

The following example shows how this method can be implemented for an object that handles unformatted text:

```
- (id)validRequestorForSendType:(NSString *)sendType
            returnType:(NSString *)returnType
{
    if ([sendType isEqual:NSStringPboardType] &&
        [returnType isEqual:NSStringPboardType]) {
        if ([self selection] && [self isEditable]) {
            return self;
        }
    }
    return [super validRequestorForSendType:sendType returnType:returnType];
}
```

This implementation checks both the types indicated and the state of the object. The object is a valid requester if the send and return types are unformatted text or simply are not specified, and if the object has a selection and is editable (when send and return types are given). If this object cannot handle the service request in its current state, it invokes its superclass's implementation.

The `validRequestorForSendType:returnType:` message is sent along an abridged responder chain, comprising only the responder chain for the key window and the application object. The main window is excluded.

Sending Data to the Service

When the user chooses a Services menu command, the responder chain is checked with `validRequestorForSendType:returnType:`. The first object that returns a value other than `nil` is called upon to handle the service request by providing data (if any is required) with a `writeSelectionToPasteboard:types:` message. You can implement this method to provide the data immediately or to provide the data only when it is actually requested. Here is an implementation for an object that writes unformatted text immediately:

```
- (BOOL)writeSelectionToPasteboard:(NSPasteboard *)pboard
types:(NSArray *)types
{
    NSArray *typesDeclared;

    if ([types containsObject:NSStringPboardType] == NO) {
        return NO;
    }
    typesDeclared = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:typesDeclared owner:nil];
    return [pboard setString:[self selection]
                     forType:NSStringPboardType];
}
```

This method returns `YES` if it successfully writes or declares any data and `NO` if it fails. If you have large amounts of data or you can provide the data in many formats, you should provide the data only on demand. You declare the available types as above, but with an owner object that responds to the message `pasteboard:provideDataForType:`. For more details, see *NSPasteboard Class Reference*.

Receiving Data from the Service

After the service requester writes data to the pasteboard, it waits for a response as the service provider is invoked to perform the operation; if the service does not return data, of course, the requesting application continues running and none of the following applies. The service provider reads the data from the pasteboard, works on it, and then returns the result. At this point the service requester is sent a `readSelectionFromPasteboard:` message telling it to replace the selection with whatever data came back. The simple text object can implement this method as follows:

```
- (BOOL)readSelectionFromPasteboard:(NSPasteboard *)pboard
{
    NSArray *types;
    NSString *theText;

    types = [pboard types];
    if ( [types containsObject:NSStringPboardType] == NO ) {
        return NO;
    }
    theText = [pboard stringForType:NSStringPboardType];
    [self replaceSelectionWithString:theText];
    return YES;
}
```

This method returns `YES` if it successfully reads the data from the pasteboard; otherwise, it returns `NO`.

Invoking a Service Programmatically

Though the user typically invokes a standard service by choosing an item in the Services menu, you can invoke it in code using this function:

```
BOOL NSPerformService(NSString *serviceItem, NSPasteboard *pboard)
```

This function returns `YES` if the service is successfully performed; otherwise, it returns `NO`. The name of a Services menu item (in any language) is contained in *`serviceItem`*. It must be the full name of the service; for example, "`Search in Google`." The parameter *`pboard`* contains the data to be used for the service, and, when the function returns, it contains the data returned from the service. You can then do with the data what you wish.

Document Revision History

This table describes the changes to *Services Implementation Guide*.

Date	Notes
2009-05-18	Updated for Mac OS X v10.6; changed the title from "System Services."
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.