Technical Note TN2017

Using Launch Services for discovering document binding and launching applications

CONTENTS

Launch Services is the new API for launching applications in Mac OS X.

Launch Services is responsible for binding documents to applications and provides the system-level functionality of launching applications. The Desktop Database falls short in identifying bundles and file extensions; Launch Services provides the appropriate functionality to identify and launch applications.

This Note is directed at application developers who have relied on Desktop Manager services in the past or require application launching from within their code.

Updated: [Apr 17 2001]

About Launch Services

Launch Services will be covered in detail within Tech Pubs documentation; however, it is important that you adopt Launch Services as soon as possible. While we don't have the full documentation, this note shows how to do some of the more important things. Another valuable resource is the headerdoc information in LaunchServices.h itself.

The primary purpose of Launch Services is to launch applications. Launch Services serves as the single, medium-level interface to starting applications of various formats, opening documents either with a specific application or using the default application, and opening URLs. Launch Services is responsible for grouping documents associated with the same application together before issuing launch requests in order to minimize the number of launch requests. None of the launching APIs are preemptive or re-entrant safe for Mac OS X.

With Mac OS X and application bundles, the Desktop Manager has lost its usefulness. Since application bundles may be composed strictly of data files and properties of the bundle listed in its `plist` file Launch Services was created to register and maintain information about which documents should be launched by which applications.

If you use the Desktop DB APIs you'll notice they ignore bundled applications. Details of how to locate applications within the Desktop Database can be found in "DTS Technote 1002, On Launching an App with a Document"

Back to top

Benefits of using Launch Services

The API for finding and launching applications has changed with Mac OS X, but this is a good thing - it's easier now. In the past, launching applications with documents in a way similar to the Finder required knowledge of several API sets including the Desktop Manager, File Manager, Translation Manager, Internet Config, Process Manager, and Apple Events. Now all this functionality has been wrapped up into one easy-to-use set of APIs.

Launch Services was created specifically to avoid the common need for third parties to ask the Finder to open an application or document for them. The Finder previously had implicit knowledge of the Translation Manager, the Desktop Database, and other aspects of determining the correct application not available elsewhere. Launch Services solves this problem by being the sole place clients should go to launch applications or open documents. The Mac OS X Finder does no additional processing beyond calling Launch Services, so any client using Launch Services will behave the same as the Finder with respect to

opening applications and documents.

Back to top

Document Binding Criteria

Locating the application is based on a number of criteria primarily set up within the application's `plist`. The user may override some information by specifying which applications are designated to open certain files, files of certain types, and extensions. Matching documents with applications is based on a set of priorities. Note that file types and creators of `'????'` are treated the same as 0.

1. Has the user specified the application for this document?
2. If a creator was provided, look for applications with the creator.
3. If an extension was provided, look for applications that claim they can open files with the provided extension (comparison of extensions is case insensitive).
4. If a file type was provided, look for applications that claim they can open files with the provided type.
5. In any case, if the located application does not claim to handle documents like the supplied inputs, it is not considered eligible and is skipped.
6. Select the first application from the collection giving preference first to native applications, then Classic applications.
7. Break any ties of the same application by choosing the latest version number.

These precedence rules produce the following examples:

File Name	Type	Creator	Results
Read Me.txt	TEXT	ttxt	Classic SimpleText because creator was specified
Read Me.txt	????	????	TextEdit.app because no creator, TextEdit claims .txt, is native, is local
Doc1.pdf	PDF_	CARO	Classic Acrobat Reader because creator was specified and no native Acrobat Reader
Doc2.pdf	TEXT	????	TextEdit.app because no creator, TextEdit claims pdf, is native, and is local
Index.html	TEXT	ttxt	Classic SimpleText because creator was specified
Index.html	TEXT	????	IE-X because no creator, IE-X claims .html, is native, and is local
Index.html	????	????	IE-X because no creator, IE-X claims .html, is native, and is local
My Doc	TEXT	MSWD	Classic Microsoft Word because creator was specified
My Doc	????	????	No binding - requests the user to select an application a
My Doc	TEXT	????	TextEdit.app because no creator, TextEdit claims OSType TEXT, is native, and is loc

Launch Services is also tasked with maintaining the collection of recently opened applications and documents. Only those applications successfully launched and only those documents successfully opened are added to the recent items.

Back to top

Locating the Default Application

Currently Launch Services is only available on Mac OS X to both CFM and Mach-O applications. The APIs are available within <LaunchServices.h> and you should check the availability of Launch Services by doing a runtime T-Vector check of an API, shown in Listing 1, as the Launch Services functionality may be brought back to Mac OS 8/9 through CarbonLib at a later date. Looking at <LaunchServices.h> you'll notice parallel versions of the APIs accepting either `FSRefs` or CFURLs.

If you use the Desktop DB APIs, you'll notice they ignore bundled applications. It used to be primarily responsible for locating which single binary application is associated with a given type and creator pair. Details of how to locate applications within the Desktop Database can be found in " DTS Technote 1002, On Launching an App with a Document"

The following code demonstrates how to locate the default application given a files OSType.

```
if ( (UInt32)LSGetApplicationForInfo !=
        (UInt32)kUnresolvedCFragSymbolAddress ) {
    err = LSGetApplicationForInfo(kLSUnknownType,
        'MSIE', nil, kLSRolesAll,
        &outAppRef, &outAppURL);
    if ( err != noErr ) return( err );
}
```

Listing 1. A source code listing to check if Launch Services is available by checking its T-Vector and then locates the default application given an applications creator, `'MSIE'`.

Likewise, you can also locate the appropriate application based on a number of other criteria including OSType, file extension, role, etc. For example, in the listing below the routine LSGetApplicationForInfo() is being called to locate the appropriate application for viewing a document based on the document's file name extension.

```
err = LSGetApplicationForInfo(
    kLSUnknownType,
    kLSUnknownCreator,
    CFSTR("html"),
    kLSRolesViewer,
    &outAppRef,
    &outAppURL
);
```

Listing 2. A source code listing locates the designated application to view ".html" files.

Back to top

Launching an Application with a Document

There are typically two scenarios in which one launches an application with a document. The simplest and most common case is launching the default application to open the specified document. (This is similar to double-clicking a file from the Finder.) This can be done by calling LSOpenFSRef().

```
EXTERN_API( OSStatus )
LSOpenFSRef(
  const FSRef *  inRef,             /* Item to open */
  FSRef *        outLaunchedRef);   /* can be NULL */
```

Listing 3. The following example shows how to open a file with the default application given an FSRef to the file you wish to open.

```
LSLaunchFSRefSpec inLaunchSpec;
    /* app to use, can be NULL*/
inLaunchSpec.appRef = &outAppRef;
    /* items to open/print, can be NULL*/
inLaunchSpec.numDocs = numDocuments;
    /* array of FSRefs*/
inLaunchSpec.itemRefs = documentRefArray;
    /* passed untouched to application as optional parameter*/
inLaunchSpec.passThruParams = nil;
    /* default = open, async, use Info.plist, start Classic*/
inLaunchSpec.launchFlags = kLSLaunchDefaults;
    /* used if you register for app birth/death notification*/
inLaunchSpec.asyncRefCon = nil;

err = LSOpenFromRefSpec( &inLaunchSpec, &outLaunchedRef );
```

Listing 4. The following example demonstrates the more interesting case of launching a chosen application to open the specified documents.

Back to top

Navigation Services Filter Proc

You may want to create a Navigation Services filter proc if you want to display only the files an application can open.

The important part of writing a Navigation Services filter proc is determining whether each item should be enabled for selection or not. To make this decision an application must gather certain information about the item and determine whether the item can be opened by the developer's application. For instance if your text editing application has a `plist` describing that it can open files of type `'TEXT'`, or files with a file name extension of "txt", or "text", this generic filter proc will query Launch Services to automatically display only files your application is capable of opening.

```
pascal Boolean NavLaunchServicesFilterProc(
                    AEDesc* theItem,
                    void* info,
                    NavCallBackUserData ioUserData,
                    NavFilterModes filterMode)
{
#pragma unused( info, ioUserData )
LSItemInfoRecord  infoRec;
FSRef  fsRef;
OSStatus  err    = noErr;
Boolean  showItem= false;
if ( filterMode == kNavFilteringBrowserList )
{
err= GetFSRefFromAEDesc( &fsRef, theItem );
if ( err != noErr ) goto BailWithError;

            /* Ask LaunchServices for information about
            package-ness and type and creator. */
        err = LSCopyItemInfoForRef( &fsRef, kLSRequestAllInfo,
                    &infoRec );
if ( (err != noErr) && (err != kLSApplicationNotFoundErr) )
goto BailWithError;

            /* Determine if we can open this item. If the
            item is a folder (and not a package) then we
            always enable it so the user can see the contents.
            Boolean isPkg =
                ( (infoRec.flags & kLSItemInfoIsPackage) != 0 );
            Boolean isApp =
                ( (infoRec.flags & kLSItemInfoIsApplication) != 0 );
            */

if ( (infoRec.flags & kLSItemInfoIsAliasFile) != 0 )
{
            /* We could resolve the alias and decide
            to select the item or not based on the
            original.  Left to the reader as an exercise. */
}

if ( (infoRec.flags & kLSItemInfoIsContainer) != 0 )
{
showItem = true;
}
else
{
Boolean canViewItem = false;

err= LSCanRefAcceptItem( &fsRef, &gApplicationFSRef,
kLSRolesViewer, kLSAcceptDefault, &canViewItem );

if ( err == noErr )
showItem = canViewItem;
}
}
else
{
showItem = true;
}
return( showItem );
BailWithError:
return( false );
}
```

```
OSStatusGetFSRefFromAEDesc( FSRef *fsRef, AEDesc* theItem )
{
OSStatus  err  = noErr;
AEDesc  coerceDesc= { NULL, NULL };
/* If the AEDesc isn't already an FSSpec,
convert it to one... */
if ( theItem->descriptorType != typeFSRef )
{
err = AECoerceDesc( theItem, typeFSRef, &coerceDesc );
if ( err == noErr )
theItem = &coerceDesc;
}
/* Get the FSRef out of the AEDesc */
if ( err == noErr )
err = AEGetDescData( theItem, fsRef, sizeof(*fsRef) );
AEDisposeDesc( &coerceDesc );
/* If we have could not get an FSRef try getting an FSSpec and
make an FSRef out of it. */
if ( err != noErr )
{
FSSpec fsSpec;
AEDesc coerceDesc2 = {NULL, NULL};

/* If the AEDesc isn't already an FSSpec, convert it to one... */
if ( theItem->descriptorType != typeFSS )
{
err = AECoerceDesc( theItem, typeFSS, &coerceDesc2 );
theItem = &coerceDesc2;
}

/* Get the FSSpec out of the AEDesc and convert it to an FSRef... */
if ( err == noErr )
err = AEGetDescData( theItem, &fsSpec, sizeof(fsSpec) );
AEDisposeDesc( &coerceDesc2 );
if ( err == noErr )
err = FSpMakeFSRef( &fsSpec, fsRef );
}
return( err );
}
```

Listing 5. A source code listing of a NavigationServices filter proc which only displays items which Launch Services decides your application can open. This becomes very important, for example, if your text editing application should open files without a file type, but with an filename extension. Download the sample code.

Back to top

Other Common Routines

`LSCopyItemInfoForRef`, used in code Listing 5, and its related functions are used to gather important information about packages. Whether it is a Carbon or Classic application, a package, on application, the type and creator, and other info. As noted in the header file, LaunchServices.h, some variants of this routine are thread safe.

Developers often want to show the same "kind" strings as displayed within the Finders list view. For example File, Folder, Alias, CodeWarrior text file, etc. `LSCopyKindStringForRef` and `LSCopyKindStringForURL` return a `CFStringRef` containing the file kind.

Back to top

References

IMPORTANT:
Be sure that you read the DTS Technote 2013, The 'plst' Resource to set up your applications `plist` correctly.

Also read DTS Technote 1002, On Launching an App with a Document for information about the Desktop Database.

Back to top

Downloadables

	Acrobat version of this Note (68K)	Download
	tn2017.hqx, an application using the Navigation Services filter code in Listing 5 (28 K)	Download

Back to top
