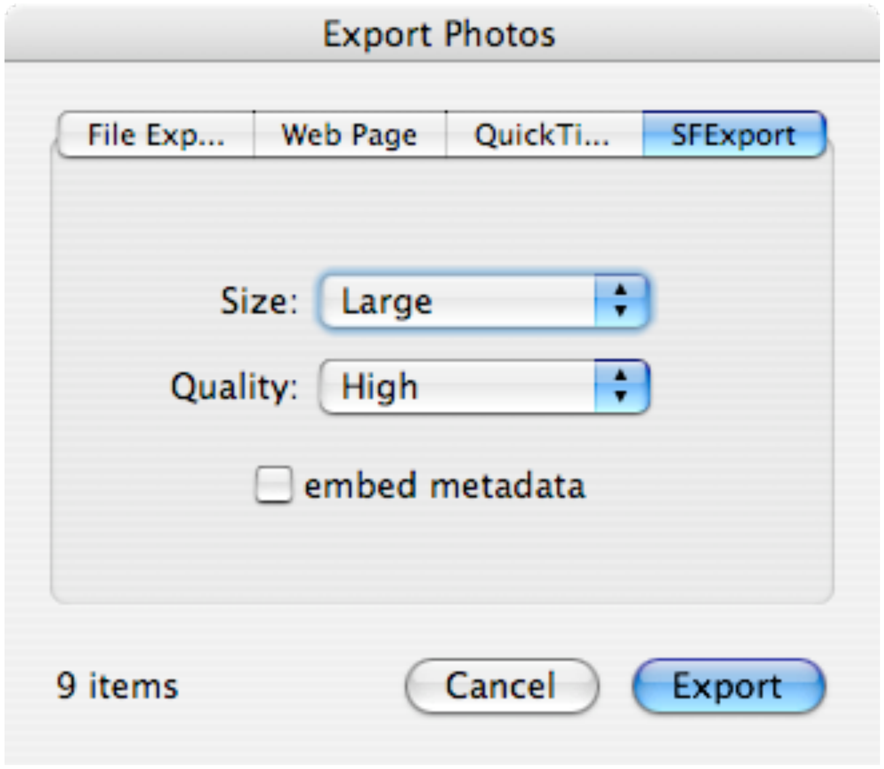
Making an iPhoto Export Plugin

Introduction

In addition to the built in exporters, iPhoto supports third party exporters in the form of export plugins. This guide will familiarize the reader with the plugin making process by walking through the steps for making a "Simple File Exporter" export plugin using iPhoto's Export Plugin API.

The plugin will allow users to set export image size, quality, and whether or not to embed metadata in the output image. The completed plugin will look like the screenshot below.



Size: Large

Quality: High

embed metadata

9 items

Figure 1. Completed plugin

The Export Plugin API consists of three protocols defined respectively in ExportImageProtocol.h, ExportPluginProtocol.h, and ExportPluginBoxProtocol.h. Please refer to the API documentation for more information on the protocols.

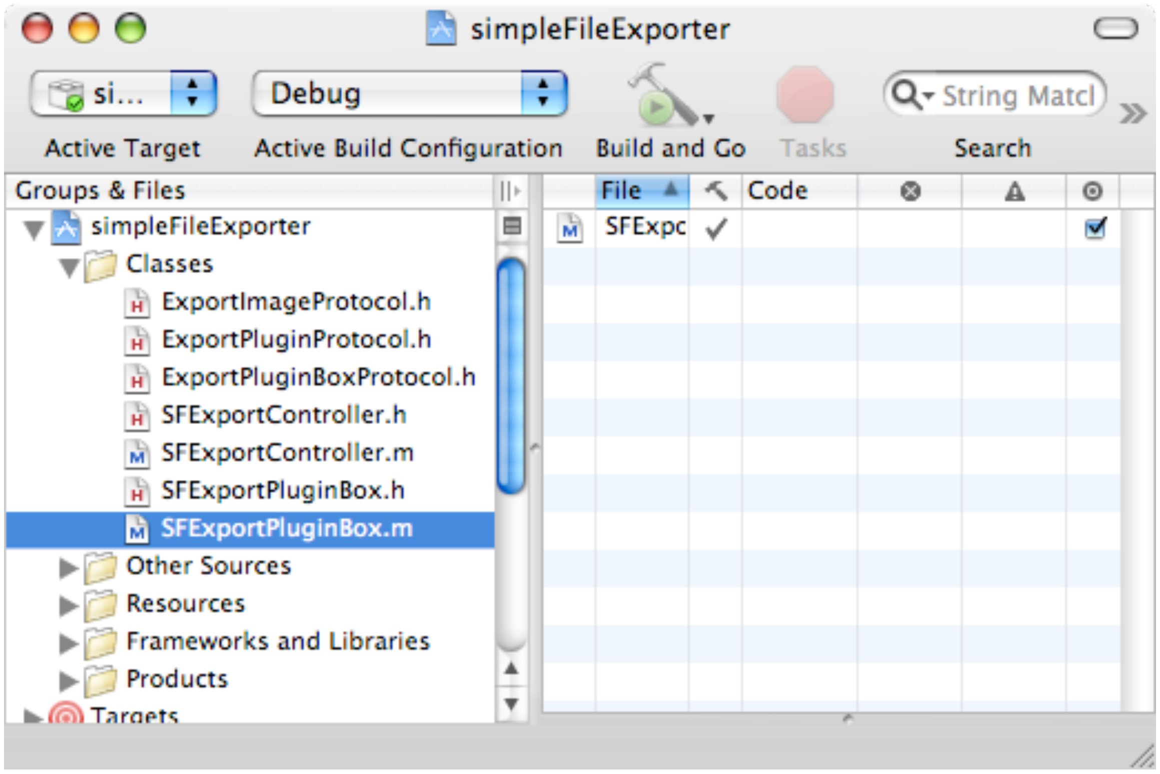
The API documentation, API header files, and source code for the "Simple File Exporter" are all included in this SDK package.

Creating the Project

The first step in creating a plugin is to start a Cocoa Bundle project. Open Xcode and select File > New Project. Chose Cocoa Bundle as the project type. Name the project “simpleFileExporter”.

An export plugin needs at least two classes: a custom NSBox class that adopts the ExportPluginBoxProtocol and a plugin controller class that adopts the ExplortPluginProtocol. Create these two classes by selecting File > New File in Xcode. Name them “SFExportPluginBox” and “SFExportController”.

After the two classes (and their header files) are created, add the three Export Plugin API header files to the project by dragging them to Xcode. Make sure to check “Copy items to destination group’s folder.”



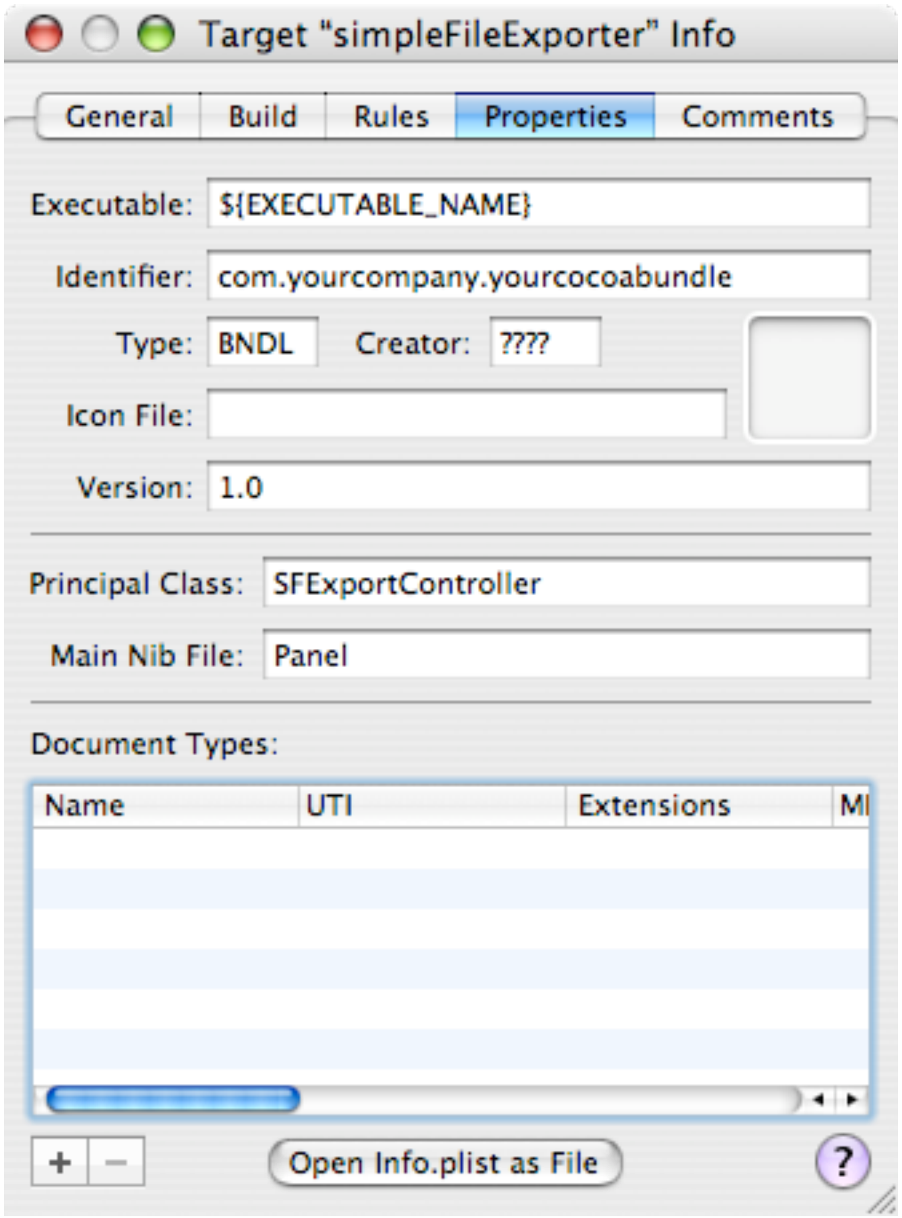
```
simpleFileExporter
  Classes
    ExportImageProtocol.h
    ExportPluginProtocol.h
    ExportPluginBoxProtocol.h
    SFExportController.h
    SFExportController.m
    SFExportPluginBox.h
    SFExportPluginBox.m
  Other Sources
  Resources
  Frameworks and Libraries
  Products
Targets
```

Figure 2. Project files

Editing Project Property List

The property list for the project must be edited so that iPhoto can find and load the plugin.

With the project open in Xcode, select Project > Edit Active Target. Select the build tab and change the Wrapper Extension to “iPhotoExport.” Selected the Properties tab and set the Principal Class to be the plugin controller class. In our case this is “SFExportController.” Set the Main Nib File to be the filename of the Nib file that will contain the UI of the plugin. For now, this file will be called “Panel”.



Executable: ${EXECUTABLE_NAME}
Identifier: com.yourcompany.yourcocoabundle
Type: BNDL Creator: ????
Icon File:
Version: 1.0
Principal Class: SFExportController
Main Nib File: Panel

Figure 3. Properties tab

The Bundle Identifier should be set to a unique string and will be used by iPhoto as a way to identify the plugin.

Plugin Box

Now is the time to add code to the plugin box class. This class will adopt the ExportPluginBoxProtocol. The plugin box will act as the super view for all other plugin views, iPhoto’s export controller will use it to layout the plugin UI in the export window. Below is the header

```
//
//  SFExportPluginBox.h
//  simpleFileExporter
//

#import <Cocoa/Cocoa.h>
#import "ExportPluginProtocol.h"
#import "ExportPluginBoxProtocol.h"

@interface SFExportPluginBox : NSBox <ExportPluginBoxProtocol> {
    IBOutlet id <ExportPluginProtocol> mPlugin;
}

- (BOOL)performKeyEquivalent:(NSEvent *)anEvent;

@end
```

and implementation files for the plugin box.

```
//
//  SFExportPluginBox.m
//  simpleFileExporter
//

#import "SFExportPluginBox.h"

@implementation SFExportPluginBox

- (BOOL)performKeyEquivalent:(NSEvent *)anEvent
{
    NSString *keyString = [anEvent charactersIgnoringModifiers];
    unichar keyChar = [keyString characterAtIndex:0];

    switch (keyChar)
    {
        case NSFormFeedCharacter:
        case NSNewlineCharacter:
        case NSCarriageReturnCharacter:
        case NSEnterCharacter:
        {
            [mPlugin clickExport];
            return(YES);
        }
        default:
            break;
    }

    return([super performKeyEquivalent:anEvent]);
}

@end
```

mPlugin points to the plugin controller object. performKeyEquivalent: is used to handle the “enter” key so that pressing “enter” initiates the export. The plugin box class should be similar to this one in any export plugin.

Creating the nib File

Once the plugin box class is complete, we can make the UI for the plugin. In Interface Builder, select File > New and pick Cocoa > Empty as the starting point.

Create a new NSWindow object and put an NSBox object inside it. Drag the plugin box header file (SFExportPluginBox.h) and drop it off at the doc window.

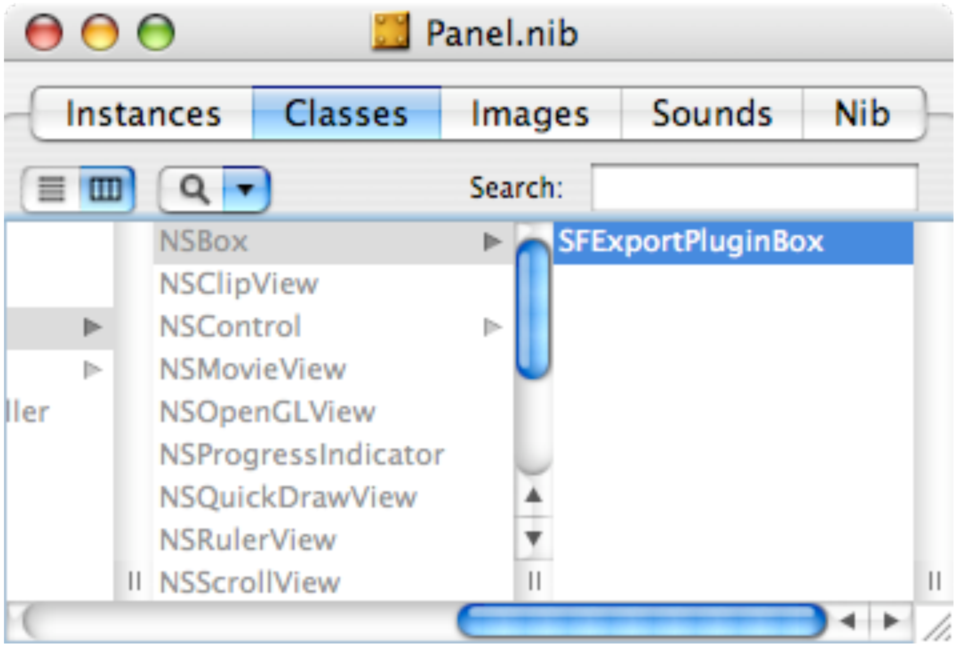


Figure 4. The doc window after dropping off SFExportPluginBox.h

Select the NSBox and open Inspector. Change the title of the box to “SFExport”. This will be the name displayed on the tab view of the export window. Under the Custom Class tab, select SFExportPluginBox.

Drop two NSPopUpButton into the SFExportPluginBox object. They will correspond to the controls for image size and quality. Now add a check box button and label it “embed metadata”. It will indicate whether the user wants to embed metadata. Finally, add two system size text and change their text to “Size:” and “Quality:”.

After everything has been placed, the layout should look like this:

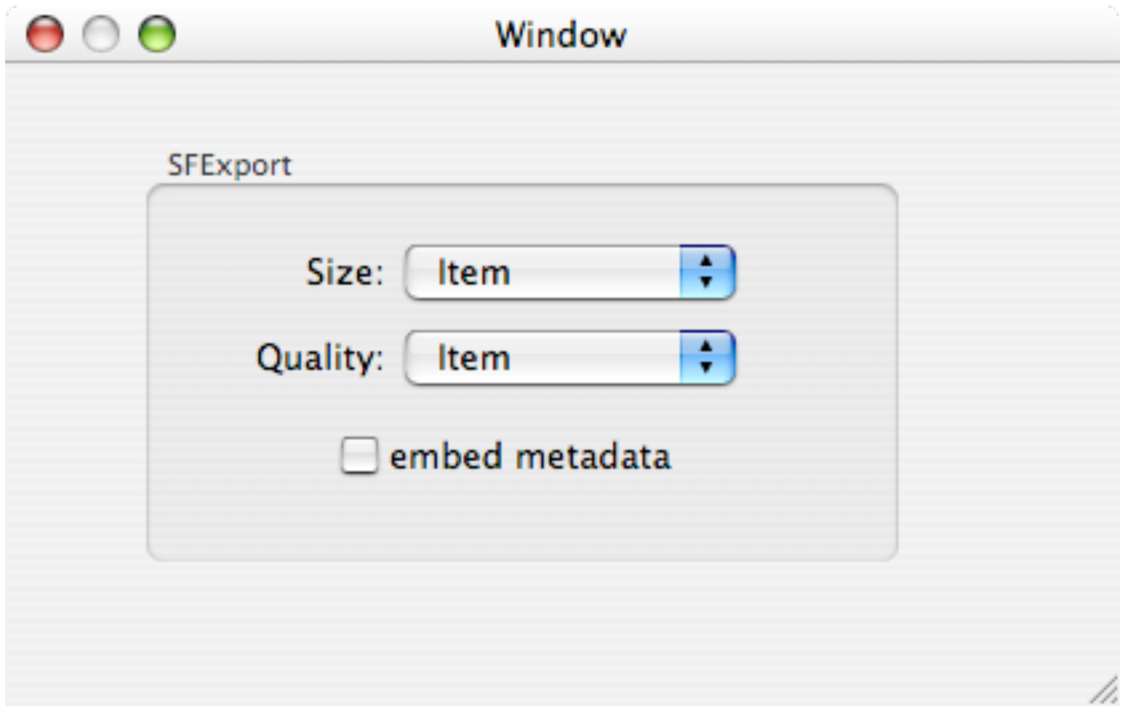


Figure 5. Plugin UI layout

Don’t worry about the size of the Window, only SFExportPluginBox and its contents are drawn in the iPhoto export window.

Now we need to add menu items to the two NSPopUpButtons. Double click the NSPopUpButton for size and add or delete menu items until there are four. From top to bottom, give them titles “Small”, “Medium”, “Large”, and “Full Size” and tag them respectively 0 to 3. Do the same with the NSPopUpButton for quality but with titles “Low”, “Medium”, “High”, and “Maximum”.

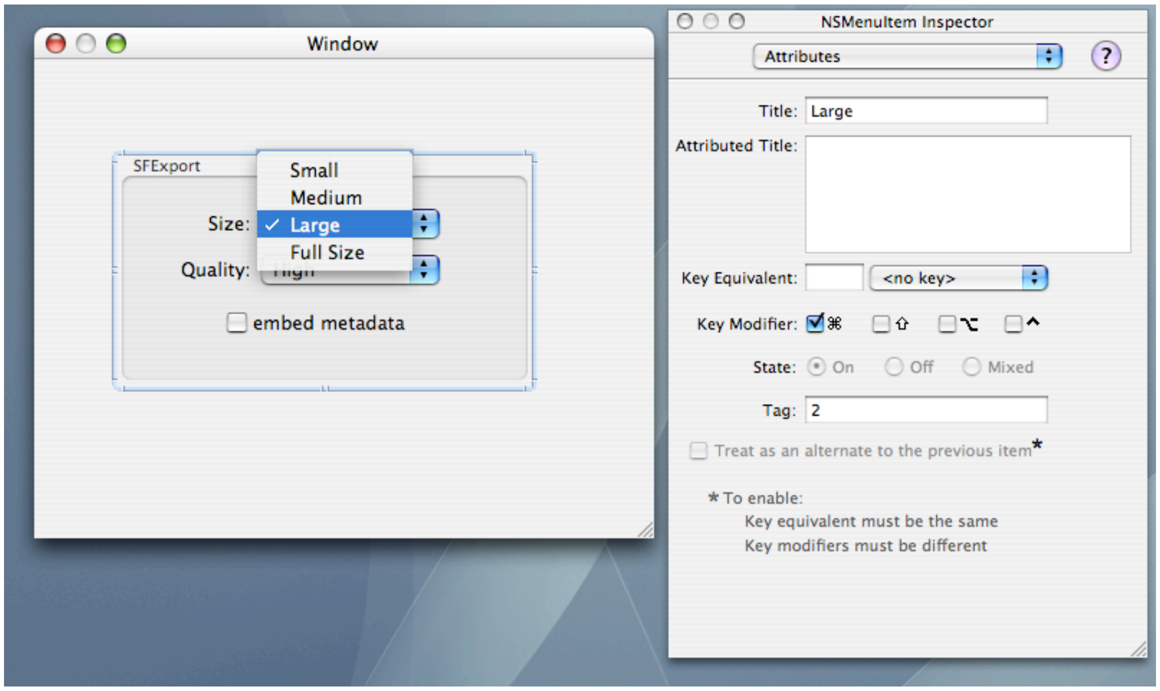


Figure 6. Editing menu items

Now save the file with the name “Panel” (same name as the one we set for plist) and add it to the simpleFileExporter project.

Plugin Controller Interface

Most of the code for the plugin is in the plugin controller. The plugin controller is responsible for communicating with the UI and performing export. The plugin controller must adopt the ExportPluginProtocol.

```
//
//  SFExportController.h
//  simpleFileExporter
//

#import <Cocoa/Cocoa.h>
#import "ExportPluginProtocol.h"

@interface SFExportController : NSObject <ExportPluginProtocol> {
    id <ExportImageProtocol> mExportMgr;

    IBOutlet NSBox <ExportPluginBoxProtocol> *mSettingsBox;
    IBOutlet NSControl *mFirstView;

    IBOutlet NSPopUpButton  *mSizePopUp;
    IBOutlet NSPopUpButton  *mQualityPopUp;
    IBOutlet NSButton       *mMetadataButton;

    NSString *mExportDir;
    int mSize;
    int mQuality;
    int mMetadata;

    ExportPluginProgress mProgress;
    NSLock *mProgressLock;
    BOOL mCancelExport;
}

// overrides
- (void)awakeFromNib;
- (void)dealloc;

// getters/setters
- (NSString *)exportDir;
- (void)setExportDir:(NSString *)dir;
- (int)size;
- (void)setSize:(int)size;
- (int)quality;
- (void)setQuality:(int)quality;
- (int)metadata;
- (void)setMetadata:(int)metadata;

@end
```

The instance variable mExportMgr stores the export manager which conforms to the ExportImageProtocol and is passed to the plugin when it is initialized by iPhoto. The export manager provides information on the user selected images and have methods that performs resizing and adding metadata. See the protocol documentation for details.

The variables mSettingsBox and mFirstView store objects the iPhoto export controller will ask for. They will be returned by their respective "get" methods defined in the ExportPluginProtocol.

mSizePopUp, mQualityPopUp, and mMetadataButton are outlets connecting to the corresponding NSButtons in the UI.

mExportDir, mSize, mQuality, and mMetadata will store the settings chosen by the user.

mProgress stores the progress struct that will control the behavior of the progress panel when plugin is performing the export. mProgressLock is the mutex lock for the struct.

Finally, mCancelExport will indicate whether to cancel the export, either due to user clicking cancel or error.

Aside from the methods declared in ExportPluginProtocol, the plugin controller will override awakeFromNib and dealloc. It will also have getter and setter methods for mExportDir, mSize, mQuality, and mMetadata.

Making Connections

Before implementing the plugin controller class, we will connect its outlets in Interface Builder. Open Panel.nib in Interface Builder and drag SFExportController.h to the doc window.

Select the File's Owner object under the Instances tab and open its inspector. Under Custom Classes, select SFExportController.

Control drag from File's Owner to the NSPopUpButton for size. Connect the mFirstView and mSizePopUp outlets. mFirstView will tell iPhoto's export controller which view will be the first responder when the plugin becomes "active."

Control drag from File's Owner to the NSPopUpButton for quality. Connect the mQualityPopUp outlet.

Control drag from File's Owner to the check box for embed metadata and connect the mMetadata outlet.

Control drag from File's Owner to the SFExportPluginBox and connect the mSettingsBox outlet.

Finally, control drag from the SFExportPluginBox to the File's Owner and connect the mPlugin outlet.

Implementing Plugin Controller

The plugin controller will implement the methods declared in ExportPluginProtocol and SFExportController.h. We start off with the following.

```
//
//  SFExportController.m
//  simpleFileExporter
```

```
//

#import "SFExportController.h"
#import <QuickTime/QuickTime.h>

@implementation SFExportController

@end
```

First implement the initialization and deallocing methods.

```
- (void)awakeFromNib
{
    [mSizePopUp selectItemWithTag:2];
    [mQualityPopUp selectItemWithTag:2];
    [mMetadataButton setState:NSOffState];
}

- (id)initWithExportImageObj:(id <ExportImageProtocol>)obj
{
    if(self = [super init])
    {
        mExportMgr = obj;
        mProgress.message = nil;
        mProgressLock = [[NSLock alloc] init];
    }
    return self;
}

- (void)dealloc
{
    [mExportDir release];
    [mProgressLock release];
    [mProgress.message release];

    [super dealloc];
}
```

awakeFromNib is called when Panel.nib is loaded. There, we set the default settings to be displayed. The default settings are “high” quality, “large” size, and no metadata.

initWithExportImageObj: initializes the plugin controller and is where the plugin gets the export manager.

Next implement the getter and setter methods for the member variables. These methods are fairly straightforward.

```
- (NSString *)exportDir
{
    return mExportDir;
}

- (void)setExportDir:(NSString *)dir
{
    [mExportDir release];
    mExportDir = [dir retain];
}

- (int)size
{
    return mSize;
}

- (void)setSize:(int)size
{
    mSize = size;
}

- (int)quality
{
    return mQuality;
}

- (void)setQuality:(int)quality
{
    mQuality = quality;
}

- (int)metadata
{
    return mMetadata;
}

- (void)setMetadata:(int)metadata
{
    mMetadata = metadata;
}
```

Now implement the rest of the ExportPluginProtocol.

```
- (NSView <ExportPluginBoxProtocol> *)settingsView
{
    return mSettingsBox;
}
```

```
- (NSControl *)firstView
{
    return mFirstView;
}
```

settingsView and firstView are straightforward getter methods for their corresponding member variable.

```
- (void)viewWillBeActivated
{

}

- (void)viewWillBeDeactivated
{

}
```

viewWillBeActivated and viewWillBeDeactivated will be called when the plugin becomes or is no longer “active.” Since for this simple exporter there is nothing we need to do in these situations, these methods are left blank.

```
- (NSString *)requiredFileType
{
    if([mExportMgr imageCount] > 1)
        return @"";
    else
        return @"jpg";
}

- (BOOL)wantsDestinationPrompt
{
    return YES;
}

- (NSString*)getDestinationPath
{
    return @"";
}

- (NSString *)defaultFileName
{
    if([mExportMgr imageCount] > 1)
        return @"";
    else
```

```
        return @"sfe-0";
}

- (NSString *)defaultDirectory
{
    return @"~/Pictures/";
}
```

requiredFileType tells iPhoto what file type the plugin exports to. We send the imageCount message to the export manager to ask how many images the user selected. If more than one image is selected, an empty string will be returned indicating a directory. Otherwise return “jpg”.

The export manager implements many helpful methods for the plugin to use. These methods are declared in ExportImageProtocol. See the protocol documentation for details on each method.

The plugin will let the user pick the export directory, so wantsDestinationPrompt returns YES. Since the user will supply the destination path, the plugin does not provide a built in path; so getDestinationPath returns an empty string.

defaultFileName returns the filename to be displayed in the save panel when iPhoto prompts the user for the export path. defaultDirectory is the directory whose contents are displayed in the save prompt when it first opens.

```
- (BOOL)treatSingleSelectionDifferently
{
    return YES;
}

- (BOOL)handlesMovieFiles
{
    return NO;
}

- (BOOL)validateUserCreatedPath:(NSString*)path
{
    return NO;
}
```

Since we want the user to be able to pick a filename if he’s exporting a single image and pick a directory when he’s exporting multiple images, treatSingleSelectionDifferently returns YES.

The plugin does not support exporting movie files, so handlesMovieFiles returns NO. Also it does not allow the user to enter export directories that does not exist, so validateUserCreatedPath: returns NO.

```
- (void)clickExport
{
    [mExportMgr clickExport];
}
```

SFExportPluginBox calls clickExport when the user presses “enter.” Its only job is to call the clickExport method of the export manager. This method should be the same for any export plugin.

```
- (void)startExport:(NSString *)path
{
    NSFileManager *fileMgr = [NSFileManager defaultManager];

    [self setSize:[mSizePopUp selectedTag]];
    [self setQuality:[mQualityPopUp selectedTag]];
    [self setMetadata:[mMetadataButton state]];

    int count = [mExportMgr imageCount];

    // check for conflicting file names
    if(count == 1)
        [mExportMgr startExport];
    else
    {
        int i;
        for(i=0; i<count; i++)
        {
            NSString *fileName =
                [NSString stringWithFormat:@"sfe-%d.jpg",i];
            if([fileMgr fileExistsAtPath:
                    [path stringByAppendingPathComponent:fileName]])
                break;
        }
        if(i != count)
        {
            if (NSRunCriticalAlertPanel(@"File exists",
                    @"One or more images already exist in directory.",
                    @"Replace", nil, @"Cancel")==NSAlertDefaultReturn)
                [mExportMgr startExport];
            else
                return;
        }
        else
            [mExportMgr startExport];
    }
}
```

After the user clicks “export” and picked an export path, iPhoto will call the plugin’s startExport: method with the path the user selected (or the path returned by getDestinationPath if wantsDestinationPrompt returned NO). The startExport: method should do last minute preparations before proceeding with the export (or prompt an alert and cancel the export).

For this plugin, the startExport: method will grab the user settings and check for filename conflicts at the export path. After everything’s ready, it will send the startExport message to the export manager. If something goes wrong, it simply returns without sending startExport.

When checking filename conflicts, the plugin first checks whether the user is exporting a single file or multiple files. If a single file is to be exported, then the save panel will already have checked for naming conflicts and the plugin can simply proceed by sending startExport to the export manager (this is not the case if wantsDestinationPrompt returned NO because the save panel will not be presented).

If multiple files are to be exported, then check for naming conflicts by looping through all the image names until a conflict is found or no image is left to check. If a conflict is found, open an alert panel and ask for user decision.

```
- (void)performExport:(NSString *)path
{
    NSLog(@"performExport path: %@", path);
    int count = [mExportMgr imageCount];
    BOOL succeeded = YES;
    mCancelExport = NO;

    [self setExportDir:path];

    // set export options
    ImageExportOptions imageOptions;
    imageOptions.format = kQTFileTypeJPEG;
    switch([self quality])
    {
        case 0: imageOptions.quality = EQualityLow; break;
        case 1: imageOptions.quality = EQualityMed; break;
        case 2: imageOptions.quality = EQualityHigh; break;
        case 3: imageOptions.quality = EQualityMax; break;
        default: imageOptions.quality = EQualityHigh; break;
    }
    imageOptions.rotation = 0.0;
```

```
switch([self size])
{
    case 0:
        imageOptions.width = 320;
        imageOptions.height = 320;
        break;
    case 1:
        imageOptions.width = 640;
        imageOptions.height = 640;
        break;
    case 2:
        imageOptions.width = 1280;
        imageOptions.height = 1280;
        break;
    case 3:
        imageOptions.width = 99999;
        imageOptions.height = 99999;
        break;
    default:
        imageOptions.width = 1280;
        imageOptions.height = 1280;
        break;
}
if([self metadata] == NSOnState)
    imageOptions.metadata = EMBoth;
else
    imageOptions.metadata = NO;

// Do the export
[self lockProgress];
mProgress.indeterminateProgress = NO;
mProgress.totalItems = count - 1;
[mProgress.message autorelease];
mProgress.message = @"Exporting";
[self unlockProgress];

NSString *dest;

if(count > 1)
{
    int i;
    for(i=0; mCancelExport==NO && succeeded==YES && i<count; i++)
    {
        [self lockProgress];
        mProgress.currentItem = i;
        [mProgress.message autorelease];
        mProgress.message = [[NSString
            stringWithFormat:@"Image %d of %d", i + 1, count]
```

```
                retain];
            [self unlockProgress];

            dest = [[self exportDir] stringByAppendingPathComponent:
                [NSString stringWithFormat:@"sfe-%d.jpg", i]];

            succeeded = [mExportMgr exportImageAtIndex:i
                                                  dest:dest
                                               options:&imageOptions];
        }
    }
    else
    {
        [self lockProgress];
        mProgress.currentItem = 0;
        [mProgress.message autorelease];
        mProgress.message = @"Image 1 of 1";
        [self unlockProgress];

        dest = [self exportDir];
        succeeded = [mExportMgr exportImageAtIndex:0
                                              dest:dest
                                           options:&imageOptions];
    }

    // Handle failure
    if (!succeeded) {
        [self lockProgress];
        [mProgress.message autorelease];
        mProgress.message = [[NSString
            stringWithFormat:@"Unable to create %@", dest] retain];
        [self cancelExport];
        mProgress.shouldCancel = YES;
        [self unlockProgress];
        return;
    }

    [self lockProgress];
    [mProgress.message autorelease];
    mProgress.message = nil;
    mProgress.shouldStop = YES;
    [self unlockProgress];
}
```

If startExport message is sent to export manager, iPhoto will call performExport: to actually do the export. The Simple File Exporter will use export manager’s exportImageAtIndex:dest:options: method to perform resizing, compression, and embedding metadata.

Before using exportImageAtIndex:dest:options:, all variables of the options struct must be set. The export format for this plugin will be JPEG, so format is set to kQTFileTypeJPEG. This constant is defined in QuickTime/QuickTime.h.

The width and height options define the maximum width and height of the image without changing its original aspect ratio. If the user selects “Full Size” as the image size, then an arbitrarily large number is used because the exported image can never be larger than its original size.

Before export, the progress struct is set to update progress status. The previous progress message is autoreleased and a new message is set. The message is autoreleased instead of released because the message may be accessed from another thread at a time when the previous message is released but the new message is not yet assigned.

If multiple images are to be exported, the images are exported one at a time in a loop. The loop stops prematurely if the user clicked “Cancel” or an export failed.

The export directory is the directory the user selected if multiple files are to be exported. So when exporting multiple images, the file name must be appended onto the export directory. If only a single image is to be exported, then the export directory is the full path to the destination file and no filename needs to be appended.

If an error occurred during export, the progress message will be updated to notify the user. After everything’s done, close the progress panel by setting shouldStop to YES.

```
- (ExportPluginProgress *)progress
{
    return &mProgress;
}

- (void)lockProgress
{
    [mProgressLock lock];
}

- (void)unlockProgress
{
    [mProgressLock unlock];
}
```

These three methods are called by iPhoto to access the progress struct. They should also be the same in any export plugin.

```
- (void)cancelExport
{
    mCancelExport = YES;
}
```

cancelExport is called when the user clicks “Cancel”.

```
- (NSString *)name
{
    return @"Simple File Exporter";
}
```

Finally, return the name of the plugin.

Installing the Plugin

Once the plugin is built without errors the sample Xcode project “simpleFileExporter” automatically installs the plugin into iPhoto’s external plugin folder at –

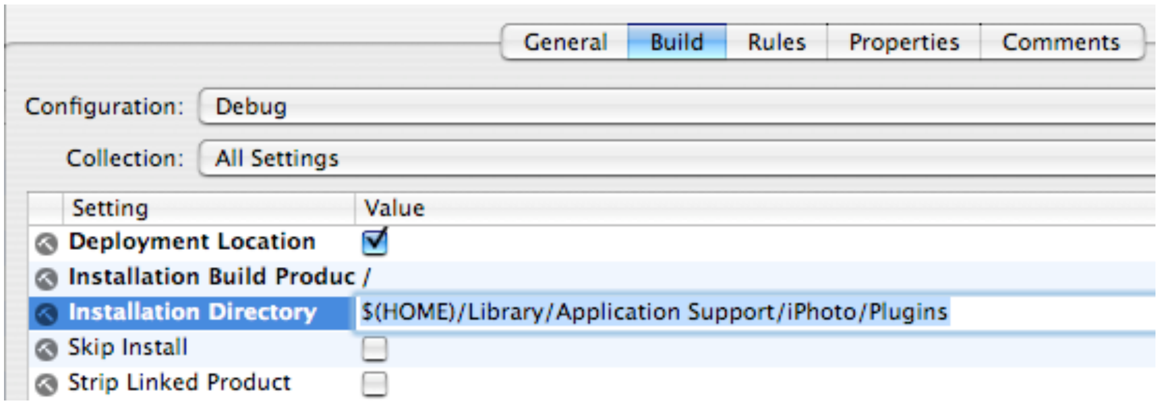
```
      ~/Library/Application Support/iPhoto/Plugins/
```

There are two places iPhoto looks for external plugins –

```
      ~/Library/Application Support/iPhoto/Plugins/
      /Library/Application Support/iPhoto/Plugins/
```

If the plugin is in either place, it should be available in the export panel the next time iPhoto starts. To uninstall the plugin, simply delete it from its folder.

The installation settings in your Xcode project’s target look like this –



Configuration: Debug

Collection: All Settings

Setting	Value
Deployment Location	✓
Installation Build Produc	/
Installation Directory	$(HOME)/Library/Application Support/iPhoto/Plugins
Skip Install	☐
Strip Linked Product	☐

Debugging the Plugin

The “simpleFileExporter” Xcode project already has a custom executable mapped to the iPhoto application so you can readily debug your plugin from within Xcode. If you create a new project on your own, you will need to add your own custom executable that will launch iPhoto.