Progressive Downloader Plug-in Implementation Guide

Introduction

Hello and welcome to the guide. You will learn how to make a simple PD plug-in and how to use the SDK. To achieve our goal you will need:

- Mac OS X 10.6 or later
- XCode 4.2 or later
- Progressive Downloader 1.0.4 or later.
- The latest version of the SDK (see the links section)
- Your knowledge of Objective-C

I believe all the points from the list above are checked and you are ready to start development.

New Project

Since we're going to do it from blank, the first thing to do is to create a new project. Launch XCode. Click New Project. Find the Cocoa Bundle template (*Fig.1*) and click Next.

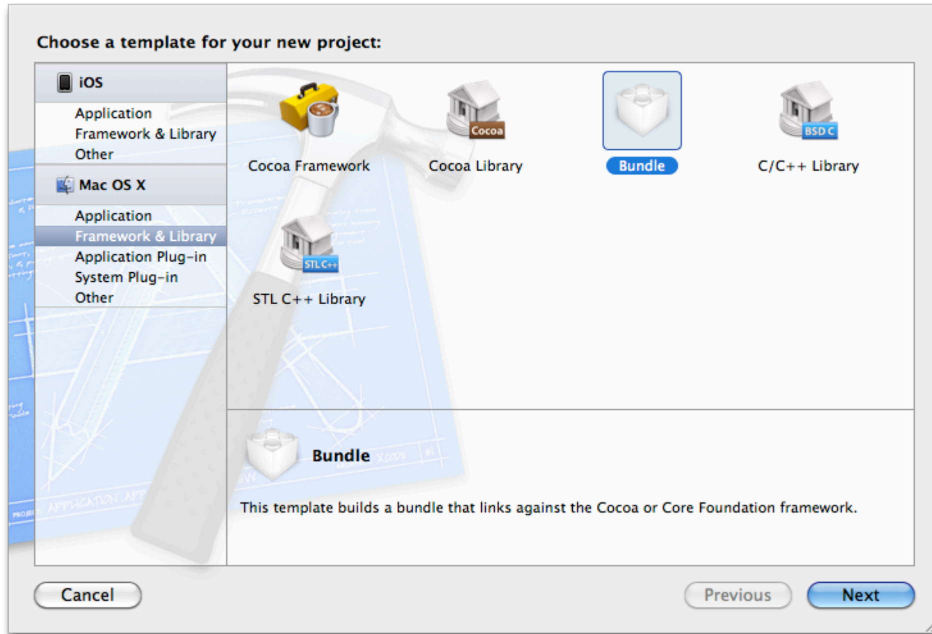

Fig.1 Selection of the Cocoa Bundle template.

Name your plug-in and input your personal company identifier. Let's use the ones from *Fig.2* for this guide.

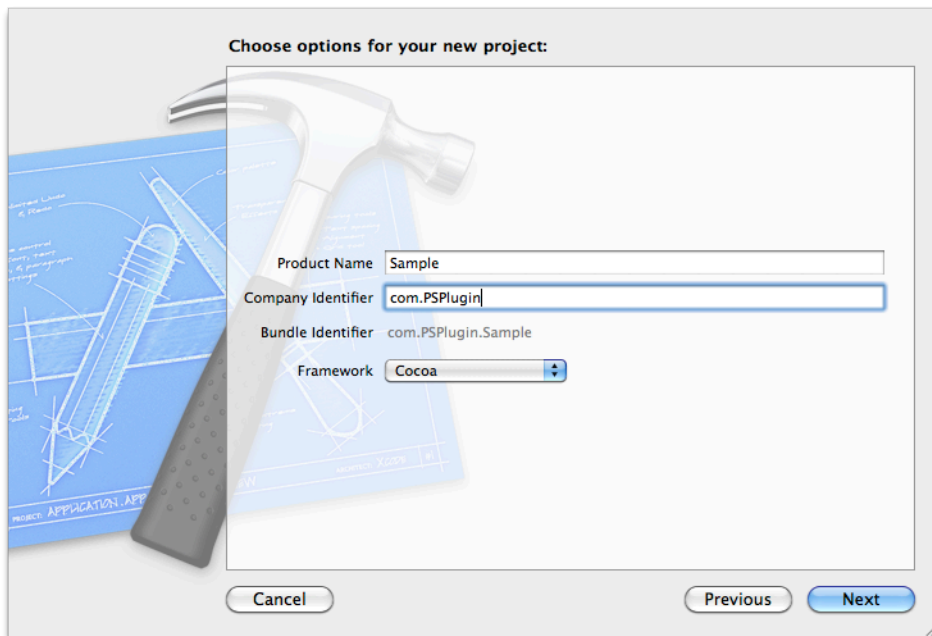

Fig. 2 New project name dialog.

Click "Next" and save the project anywhere you want. Now we need to copy the plug-in packing utility to the folder of our new project. Find SDK.zip you should have downloaded and

unpack it. Inside the SDK folder you find a file named "pipack". Copy it to the new project folder and bring XCode back to front.

We've got two steps remaining: add a script build phase and add SDK source files to the project. I hope there's no need teach you how to add those files to the project (drag'n'drop them ;) so let's assume there's only one step left – add the phase. Choose the only build target we have for now and select the Build Phases tab. Now click Add Build Phase and select Add Run Script. You should see a new phase has just added to the build line, expand it and copy the following script.

```
rm -fr "${BUILT_PRODUCTS_DIR}/.*"
cd "${SRCROOT}"
./pipack --sourcedir "${BUILT_PRODUCTS_DIR}" --output "${TARGET_NAME}.psplugin"
```

We are ready to try and build a plug-in package. Click Build or Cmd+B and check the project folder for a file named "Sample.psplugin". It's a plug-in package that could be loaded by Progressive Downloader.

In the next chapter we will add some logic and ask Progressive Downloader to load our plug-in.

Acquaintance with the Plug-in SDK

Each PD plug-in must include a class that implements `PSPluginProtocol` declared in "PSPluginProtocol.h". Let's create one. Remember that it must be a Cocoa Objective-C class and it must have a unique name. Open the new file wizard (Cmd+N), select Cocoa from the group selection, Objective-C class and go the next step by clicking "Next".

Here's the step where we must name our new class. There's the Apple's class naming rule: one must name classes with a two-letter prefix and the prefix must not be NS. PD supplements the rule with one more prefix – PS. Try to avoid using this prefix in your plug-ins.

We know that we shouldn't use the PS prefix in our plug-in but here we've going to make an exception. It's a test project anyway. Let's name the new class `PSPrincipalClass` and hit "Next" (*Fig. 3*).

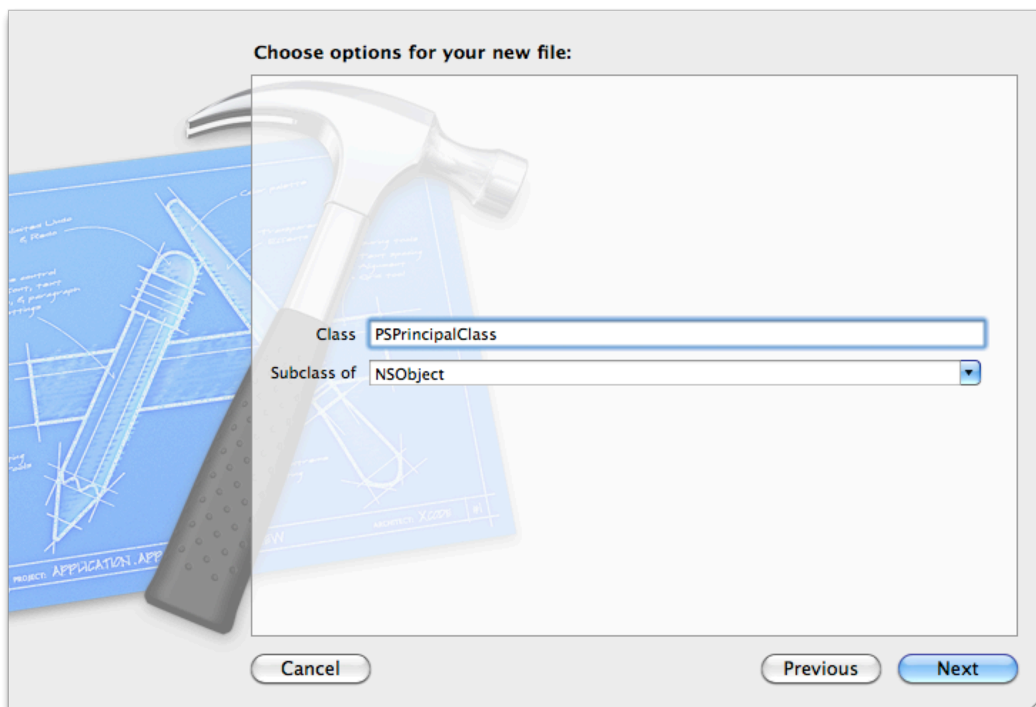

Fig. 3 Class naming dialog

The new class, we've just created. Now we need to let PD that it's the class we're going to expose. Open the "Sample-Info.plist" file and input `PSPrincipalClass` as the value for Principal class property. The class needs implementation. Open its header and add link "PSPluginProtocol.h" with an import directive. After it, declare that the new class conforms `PSPluginProtocol`. Your "PSPrincipalClass.h" should now look like this:

```
//
//  PSPrincipalClass.h
//
//

#import <Foundation/Foundation.h>

#import "PSPluginProtocol.h"

@interface PSPrincipalClass : NSObject
    <PSPluginProtocol>
```

```
@end
```

The time has come to implement some logic. `PSPluginProtocol` requires that we implement at least two main methods, which are: `initialize` and `clean`. `initialize` is the method to be called when our plug-in loads up. This could be the moment we one has just installed the plug-in or, if the plug-in is already installed, on application launch. `clean` will be called only once on plug-in uninstallation. So you need to place all the clean up procedures there.

Let's write some code that will add an application menu item click on which will bring a dialog box. I believe you already noticed those "PSPluginAPI.*" files. That's the moment when we need to use their potential. Include "PSPluginAPI.h", add static declaration of the API instance variable and prepare `initialize` the method body for our code. The file should now look as the following:

```
//
//  PSPrincipalClass.m
//

#import "PSPrincipalClass.h"

#import "PSPluginAPI.h"

@implementation PSPrincipalClass

// API instance
static NSObject <PSPluginAPIProtocol> *apiInst = nil;

- (BOOL)initialize
{

}

@end
```

In order to work with PD instance we must acquire a plug-in SDK instance first. The `GetPSPluginAPI` function is the one we need. It has just one parameter, which is the id of the calling bundle. Keep in mind that our class name is unique and you are able to get the id starting with `NSBundle`'s `bundleForClass`. Now you got the API instance and ready to add your own functionality to Progressive Downloader.

```
// get id of this bundle
NSString *bundleId = [[NSBundle bundleForClass: [self class]] bundleIdentifier];
NSAssert(bundleId,@"!!! Bundle ID is not initialized!");

// get API instance
apiInst = [GetPSPluginAPI(bundleId) retain];
NSAssert(apiInst,@"!!! Something went wrong! Check what you forgot to do!");
```

Using API's `addMainMenuItemWithTitle:action:target:keyEquivalent:` we can add an application menu item. Its syntax in almost similar to `NSMenu`'s `addItemWithTitle:action:keyEquivalent:`.

```
// add our menu item
[apiInst addMainMenuItemWithTitle: @"Our test item"
                           action: @selector(testItemClicked:)
                           target: self
                    keyEquivalent: @""];
```

The last two strokes and we're done. First is to implement the action method for the menu item. Declare it and add an API's `showAlert:…` call there. And second is to declare the `clean` method with simple return. The full file list should look like this:

```
//
//  PSPrincipalClass.m
//

#import "PSPrincipalClass.h"

#import "PSPluginAPI.h"

@implementation PSPrincipalClass

// API instance
static NSObject <PSPluginAPIProtocol> *apiInst = nil;

- (IBAction)testItemClicked: (id)sender
{
    // HELLO, WORLD!
    [apiInst showAlert: @"Message!"
         defaultButton: @"OK"
       alternateButton: nil
           otherButton: nil
                   msg: @"Hello, World!"
               warning: NO];
}

- (BOOL)initialize
{
    // get id of this bundle
    NSString *bundleId = [[NSBundle bundleForClass: [self class]] bundleIdentifier];
    NSAssert(bundleId,@"!!! Bundle ID is not initialized!");

    // get API instance
    apiInst = [GetPSPluginAPI(bundleId) retain];
    NSAssert(apiInst,@"!!! Something went wrong! Check what you forgot to do!");

    // add our menu item
    [apiInst addMainMenuItemWithTitle: @"Our test item"
                               action: @selector(testItemClicked:)
                               target: self
                        keyEquivalent: @""];

    return YES;
}

- (BOOL)clean
{
    return YES;
}

@end
```

Now build the project and double-click the plug-in file to install what we wrote.

NOTE: Always remove old plug-in from the application before giving another try with new version. The plug-in installation process checks version before installation and it's the same as the one you have installed, it will do nothing.

Task Control

In this chapter we will create a menu item that deletes all completed tasks. First off, add another menu item in the `initialize` method and prepare an empty action method. The API method we're going to use to get list of completed tasks is `tasksWithState:`. It returns tasks with the state you specified as its parameter (use the `tasks` method to get the full list).

```
// get all competed tasks
NSArray *cmpList = [apiInst tasksWithState: kPSPluginAPITaskStateDone];
```

Once we got the list we must ask user whether he wants to remove them or not. Again we're going to use the `showAlert:` method we used once in previous chapter but this time we need to use two buttons and method result.

```
if (NSAlertAlternateReturn == [apiInst showAlert: @"Confirmation!"
                                        defaultButton: @"No"
                                      alternateButton: @"Yes"
                                          otherButton: nil
                                                  msg: @"Are you sure you want to delete
all completed tasks?"
                                              warning: YES])
        {

        }
```

User has confirmed the dialog and now we need to clear the tasks up.

```
for (NSObject <PSPluginTaskProtocol> *task in cmpList)
   [task remove];
```

Looks simple, doesn't it? The whole action code listing:

```
// get all competed tasks
NSArray *cmpList = [apiInst tasksWithState: kPSPluginAPITaskStateDone];

// check if there any complete tasks
if ([cmpList count]>0)
{
    if (NSAlertAlternateReturn == [apiInst showAlert: @"Confirmation!"
                                           defaultButton: @"No"
                                         alternateButton: @"Yes"
                                             otherButton: nil
                                                     msg: @"Are you sure you want to delete all
completed tasks?"
                                                 warning: YES])
    {
        NSUInteger error = [cmpList count];
        for (NSObject <PSPluginTaskProtocol> *task in cmpList)
            error -= [task remove];
        if (error)
        {
            // it doen't happen too frequently but let's treat this case anyway
            NSString *message = [NSString stringWithFormat: @"Unable to remove %d
tasks!",error];
            [apiInst showAlert: @"Error!"
                 defaultButton: @"OK"
               alternateButton: nil
                   otherButton: nil
                           msg: message
                       warning: YES];
        }
        else
            [apiInst showAlert: @"Information!"
                 defaultButton: @"OK"
               alternateButton: nil
```

```
                        otherButton: nil
                                msg: @"All tasks successfully deleted!"
                            warning: NO];
    }
}
else
    [apiInst showAlert: @"Information!"
         defaultButton: @"OK"
       alternateButton: nil
           otherButton: nil
                   msg: @"No completed tasks found!"
               warning: NO];
```

Now let's do something more complex. For instance, we can create a trigger that removes task once it's done. To do it we must add `PSPluginTaskDelegateProtocol` to the protocol list of our principal class and move the import "PSPluginAPI.h" directive to the header file. Now back to the implementation file and declare the `taskCompleted:` delegate method like it's declared in the protocol. You already know that to remove task you need to call its method `remove`. Let's do it. The listing should look like the following:

```
- (void)taskCompleted: (NSObject <PSPluginTaskProtocol> *)task
{
    // the only thing to do is to remove the task
    [task remove];
}
```

Next step is to say the API instance that our class instance is ready to be a delegate. Add the following line after the line where we create the API instance.

```
[apiInst setDelegate: self];
```

The only thing left to do is to update the clean method where we'll release the API instance. If we won't do it application may crash because when user removes our plug-in and continues using the application without restart the API instance still count our principal class instance as a delegate when it's already released.

```
- (BOOL)clean
{
    [apiInst release];
    return YES;
}
```

Time to check what you've done.

File-system Operations

Starting off version 1.1 Progressive Downloader runs in the sandbox environment (Mac App Store version only) and you should limit usage of file-system by using application data directory (`applicationDataPath`) and temporary directory (`generateTempFileName`). There are cases when you need to expand the list of accessible directories, you can do it by using the `acquireAccessToFolder:`[1] method. The method asks user to allow directory access in the sandbox and immediately returns YES in normal environment. It's a good practice to call this method before each file operation.

1. This method is deprecated in Plug-in API version 1.6. Please check the appropriate chapter for a replacement.

New in 1.3

Plug-in API gets three new methods in 1.3. First two methods are needed if your plug-in should interact with other application windows and must be called using the main API instance.

The `mainWindow` method returns main application window instance. It's similar to [`NSApp mainwindow`] call with one "but" - it returns correct instance even when the window is hidden or closed.

The `beginSheet:modalDelegate:didEndSelector:contextInfo:` method will be found helpful in case you need to open a sheet but application doesn't have any visible window. Use it to be sure your panel will be opened as a sheet rather than a window.

The last method must be called using a task instance. It adds a record to the task log (the one you can read in the bottom of the main window when one task is selected). There are two versions of the `addLogRecordOfType:` method. One of them gets static string value as argument and another supports formatting similar to [`NSString stringWithFormat:`].

New in 1.4

There's only one new method in this API update. The method is an extended version of `showAlert:` that supports optional checkbox control (suppressionButton). It can be helpful in case you need user to apply the same dialog answer to all items in list.

New in 1.6

The `acquireAccessToFolder:` methods have been deprecated, two new methods are here to replace them: `startAccessToFolder:withMessage:` and `endAccess:`. Plug-in must start access session with `startAccessToFolder:withMessage:` each time it's going to commit a file operation. Once the operation is finished, the plug-in must close the access session with an `endAccess:` call. It's very important to strike a balance between calls of these two functions so system could free bookmark resources when they aren't needed any more. For short, here's how these methods should be used:

1. Start a session (method returns session identifier on success and nil on failure).
2. Do file operations within the folder you asked to start session in.
3. End the session using its identifier.

Another new method helps you watch over bookmarks usage: `accessStats`. It returns an `NSArray` where each item is an `NSArray` with two objects. First object is session identifier (read "path") and second is usage count. In most cases you'll need it only to check you're doing right with the previous two methods. There's no reason to reinvent the wheel so just install the LogReader plug-in (check the links section) that already exposes such functionality.

Three new methods were introduced to help with implementation of file client behavior:

1. `download:toPath:extra:handler:`
2. `upload:toPath:extra:handler:`
3. `listingOf:extra:error:`

Check a neat and quick example for the first method below:

```
__block BOOL abortProcess = NO; // set it to YES to abort the download
// we can test with the latest development version disk image, knowing it's always there
NSURL *source = [NSURL URLWithString: @"http://www.macpsd.net/update/development/PSD.dmg"];
// destination is the app's temorary folder thus result filename is destination +
@"PSD.dmg"
NSURL *destination = [NSURL fileURLWithPath: [[_api generateTempFileName]
stringByDeletingLastPathComponent]];

[_api download: source toPath: destination extra: nil handler: ^BOOL(off_t downloaded,
off_t total, NSError *error){
      if (!error) {
                 // when error is not assigned, the process is on its way
                 // indicator is an NSProgressIndicator instance
                 [indicator setDoubleValue: downloaded];
                 [indicator setMaxValue: total];
      } else {
                 // bump a rock on the road, let user know of it
                 [_api showAlert: @"Error!"
                   defaultButton: @"OK"
                 alternateButton: nil
                     otherButton: nil
                             msg: [error localizedDescription]
                         warning: YES];
      }

      return abortProcess;
}];
```

Since `setApplicationImage:forKey:` generates app-wide notification to repaint all the icons, it's must more preferable to set all images at once. Here comes one more new method – `setApplicationImages:`. It takes one argument, an `NSDictionary,` where values are `NSImages` for appropriate keys (full list of keys could be returned with the `applicationImageDictionary` method).

The new API gets a unified method get statistics of task count and downloaded bytes (functionality of the method will be extended in future releases) – `statisticsFor:`. It's a good alternative to the `tasksWithState:` method when you just need to get count of tasks with some state. At the release moment, the method is only able to return statistics for the following keys:

`KPSPluginAPIStatRunningTaskCountKey` – count of currently running tasks.
`kPSPluginAPIStatCompletedTaskCountKey` – count of completed tasks.
`kPSPluginAPIStatSuspendedTaskCountKey` – count of suspended tasks.
`kPSPluginAPIStatScheduledTaskCountKey` – count of scheduled tasks.
`kPSPluginAPIStatIncompleteTaskCountKey` – count of incomplete tasks.
`kPSPluginAPIStatTotalTaskCountKey` – total task count.
`kPSPluginAPIStatDownloadedBytesKey` –count of downloaded bytes.

The last but not least is the method that's capable to display an HTML document – `showDocumentAt:` and `showDocumentAt:modal:`. It would come useful to show a guide or an FAQ article from your own plug-in.

Links

Progressive Downloader Plug-in SDK – http://www.macpsd.net/developer/SDK.zip

Plug-in Sample – http://www.macpsd.net/developer/PluginSample.zip

LogReader – psplugin://com.ps.LogReader