Danny Greg

Building an Upload Plugin for RMUploadKit

© Realmac Software 2009-2010

Requirements

This document assumes a knowledge of programming in Objective-C and with the Cocoa frameworks. Experience in working with frameworks and plugins, although not required, will certainly help.

In order to build plugins for the RMUploadKit framework you will need the following:

- A 64-bit Intel Mac
- Mac OS X Snow Leopard
- Xcode Developer Tools 3.2 or later
  - This is the build found on the Snow Leopard install disk
- The RMUploadKit SDK

Uploader plugins are 64 bit only and require Garbage Collection.

This documentation has been put together using Xcode 3.2 therefore if using a later version screenshots may differ from your version of Xcode.

Framework Assumptions

The upload kit framework makes two critical assumptions.

1. Your bundle identifier must never, ever change.
2. Your preset, credentials and plugin classes all reside within that bundle.
3. Your preset and credential class names are persisted, and expected to be found in future revisions of your bundle.

Bundle Format

Your plugin bundle **must** have the file extension ‘uploader’.

Your plugin bundle identifier (the value for `CFBundleIdentifier` in the Info.plist) must be unique, and **must not** be prefixed ‘com.realmacsoftware’; that prefix is reserved.

Your bundle’s Info.plist file **must** also have a ‘RMUploadPluginBundleMinimumFrameworkBundleVersion’ key, the value for this key must be <string>1</string> or the plugin will not be loaded.

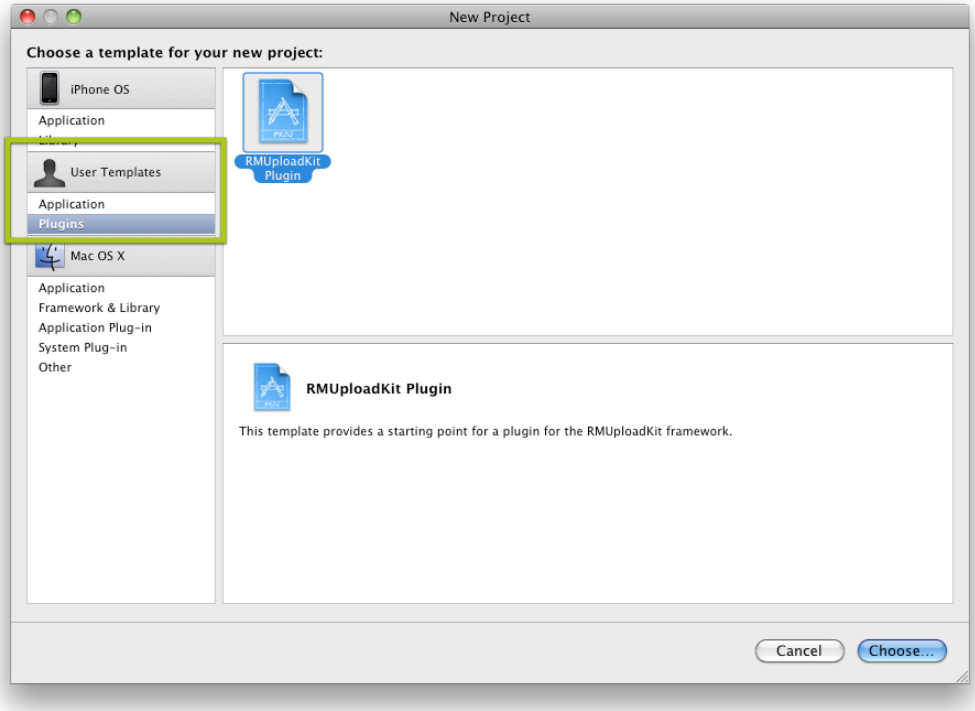
© Realmac Software 2009-2010

Getting Started

To install the SDK open the installer package.

This installs the SDK in `~/Library/Application Support/Developer/Shared/Xcode/SDKs/RMUploadKit.sdk`. You *can* put the SDK anywhere, though the 'RMUploadKit Plugin' project template is configured to look. If you do move it, adjust the `ADDITIONAL_SDKS` build setting accordingly.

You should now be able to find the project template under "User Templates" in Xcode's New Project window:



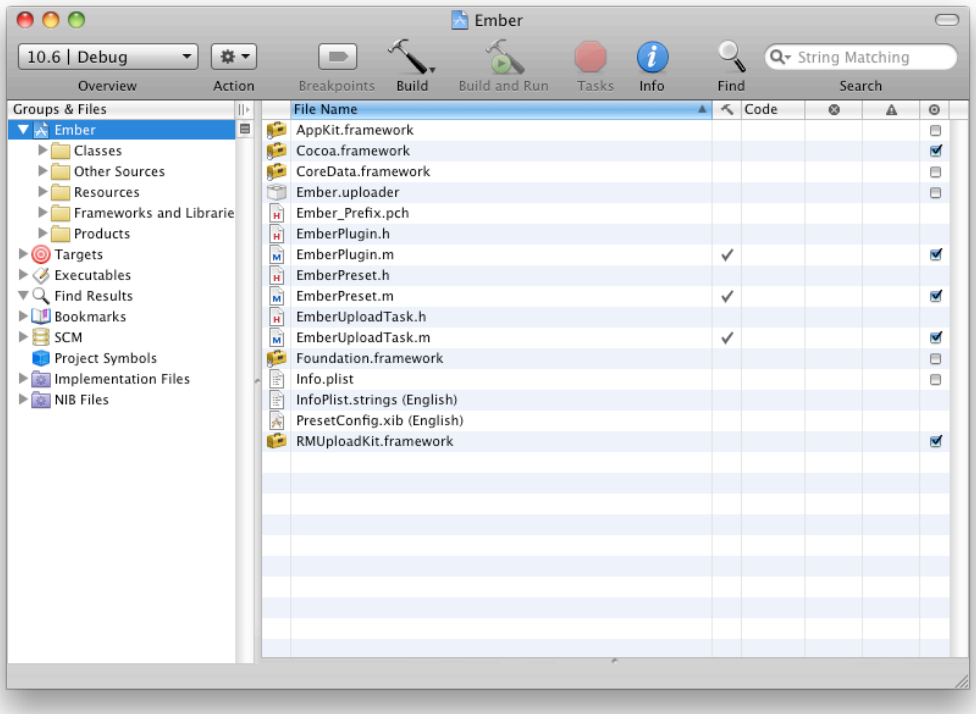
The template should build out of the box. It includes a plugin, preset and upload task, as well as configuring build settings for 64 bit Intel and Garbage Collection.

It also sets the `RMUploadPluginDestinationTypes` key for your template preset and links against RMUploadKit.framework. All of these steps can, of course, be done manually.

Your first build may be slower than usual, Xcode first creates a Composite SDK from the Base SDK (your platform, which will be Mac OS 10.6) plus the RMUploadKit SDK, this is expected.

© Realmac Software 2009-2010

Assuming you have chosen to use the template you then choose to name your project, in this case we will be creating a plugin for Ember. You should end up with a project setup similar to this:



The project should build successfully, you are now set to start coding.

© Realmac Software 2009-2010

Plugin Structure Overview

Upload Plugin

The plugin object is the go-between for the framework and your plugin's UI. There is one instance per plugin bundle and it provides view controllers to the framework.

The plugin must be the bundle's `NSPrincipleClass`.

It is worth mentioning that one plugin bundle can contain support for as many services as you like. As long as there is one `RMUploadPreset` class for each, the plugin loader knows what your plugin contains by looking in the Info.plist for the `RMUploadPluginDestinationTypes` key, which will be discussed later in the document.

Upload Presets

Upload presets are a way of describing the intended destination of uploads in a way that can be persisted to avoid users entering details every time they wish to perform an upload. This means that depending on the service, presets can contain completely different information. For example, if we were writing a plugin that would just take a file and publish it to a location on disk, the preset would just need to store the target folder's path.

However, if we were creating a plugin for the Ember image service the preset should store what collection the user wants to upload to along with a privacy level and any categories the user wishes to upload to.

Upload Credentials

In some cases a preset will hold authentication information, however in cases where the user will benefit from being able to have multiple presets for a single account we have a credentials object.

An example to demonstrate this difference is an FTP account versus a Facebook account. With FTP there is no real benefit to having more than one preset per account so the preset can take charge of the login credentials. However, with Facebook as there could be any number of photo albums to upload an image to it is beneficial to keep the login details separate from the preset so we use a credentials object.

Upload Tasks

The framework deals with the creation and queueing of tasks, however it is the task's responsibility to know how to upload the file it has been given to the requested destination.

Unlike other background task APIs, a run loop *is* available and serviced on the thread that `upload` is called on. You are discouraged from scheduling your network connections on another thread. You **must not** recursively service the run loop yourself.

© Realmac Software 2009-2010

Building the Preset

Our Ember plugin is going to support multiple presets for each account, so we don't need to worry about authorisation in the preset, but will instead add a credentials object a bit later. What we are going to store however is a target collection, category and privacy level for any uploaded images.

`RMUploadPreset` objects are really just model objects, a bundle of properties. However they must know how to write those properties out to a plist and read them in from one.

The first step is to add some properties to the object, declaring them in the header file first:

```
#import <Cocoa/Cocoa.h>

@interface EmberPreset : RMUploadPreset

@property (assign) NSString *privacyLevel;
@property (retain) NSDictionary *targetCollection;
@property (retain) NSDictionary *targetCategory;

@end
```

You can see that we haven't declared any instance variables, this takes advantage of the modern runtime that allows us to use synthesized instance variables.

Open up the implementation file and you will notice that we have a lot of boiler plate code from the template. The `localisedName` and `uploadTaskClass` have been implemented for us, customise them if you need to.

The methods of interest here are `propertyListRepresentation` and `initWithPropertyListRepresentation:`. This is how we save and load presets respectively.

Essentially we are serialising the object but as we know that it is going into a plist we must be careful to only use objects that can be represented correctly. These are `NSString`, `NSData`, `NSNumber` and `NSDate` along with `NSDictionary` and `NSArray` as containers of those classes. Therefore the implementation of those methods for our EmberPreset would look something like this:

© Realmac Software 2009-2010

```
- (id)initWithPropertyListRepresentation:(id)values
{
	id superRepresentation = [values objectForKey:@"super"];
	self = [super initWithPropertyListRepresentation:superRepresentation];
	if (self == nil) return nil;

	self.privacyLevel = [values valueForKey:RMEmberPresetPrivacyKey];

	if ([values valueForKey:RMEmberPresetCollectionKey] != nil)
		self.targetCollection = [values valueForKey:RMEmberPresetCollectionKey];

	if ([values valueForKey:RMEmberPresetCategoryKey] != nil)
		self.targetCategory = [values valueForKey:RMEmberPresetCategoryKey];

	return self;
}

- (id)propertyListRepresentation
{
	id superRepresentation = [super propertyListRepresentation];

	NSMutableDictionary *plist = [NSMutableDictionary dictionary];
	[plist setObject:superRepresentation forKey:@"super"];
	[plist setValue:self.privacyLevel] forKey:RMEmberPresetPrivacyKey];

	if (self.targetCollection != nil)
		[plist setObject:self.targetCollection
			forKey:RMEmberPresetCollectionKey];

	if (self.targetCategory != nil)
		[plist setObject:self.targetCategory
			forKey:RMEmberPresetCategoryKey];

	return plist;
}
```

There are a few things to note in these implementations. The first is the name-spacing of the super's representation. We do not guarantee the depth of the class tree for your instance, therefore you should be sure to pass the relevant data up the class hierarchy to ensure a proper initialisation.

In this example we do that by asking for the super's plist representation and storing that keyed against "super" in the dictionary that we return.

It is also worth noting that we do not make any guarantees about the initialisation environment of the plugins. They could be initialised on any thread.

As more of a coding best-practice, there are no string literals used for keys, instead we have declared constants (not visible in the chunk above), this avoids the runtime headaches that typos can cause.

So now our `EmberPreset` object can save and load itself correctly and stores the information that we need, but we need to be able to tell it when to persist its data. Rather than having a method to save we use Key-Value Observing in the framework to trigger saving of presets. Each preset has an `RMUploadPresetDirtyKey`, when set to YES the preset will trigger a save.

The key can either be set manually or by overriding `keyPathsForValuesAffectingValueForKey:` to fake a Key-Value Observing notification on the 'dirty' property to trigger a save. For our EmberPreset it looks like this:

© Realmac Software 2009-2010

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key {
     NSMutableSet *keyPaths = [[super keyPathsForValuesAffectingValueForKey:key] mutableCopy];

     if ([key isEqualToString:RMUploadPresetDirtyKey]) {
             [keyPaths addObjectsFromArray:[NSArray arrayWithObjects:
                                            RMEmberPresetPrivacyKey,
                                            RMEmberPresetCollectionKey,
                                            RMEmberPresetCategoryKey,
                                            nil]
              ];
     }

     return keyPaths;
}
```

So any time a change is made to the privacy, collection or category the framework will observe it and the preset is saved to disk.

Using Credentials

As we have chosen to keep login details separate to the preset we will need a credentials object. Again, like presets, these are really just model objects, containers of data. Although when it comes to login credentials it is more likely that you will be storing information in the keychain or another secure location, so the credentials object just needs to know how to retrieve that information.

However, an Ember account is much more than just login credentials. We can get all sorts of information such as the user's avatar, full name, date registered etc. by also storing this information we can provide a much nicer view to the user when choosing which account to use for a new preset. So we settled with storing these properties:

```
@interface RMEmberCredentials : RMUploadCredentials

@property (copy) NSString *userName;
@property (copy) NSString *fullName;
@property (copy) NSString *token;
@property (copy) NSURL *avatarLocation;
@property (retain) NSDate *dateRegistered;
@property (assign) NSUInteger totalImagesUploaded;
@property (retain) NSArray *collections;
@property (assign) NSUInteger followerCount;

@end
```

We then implemented the property-list methods as before in the preset and the credentials class is pretty much complete.

However there is one other method that a credentials class must implement:

```
- (NSString *)userIdentifier {
     return [self userName];
}
```

You should ensure this property is should be Key-Value Observable, using dependent key paths as appropriate.

© Realmac Software 2009-2010

As we manage the credentials with no real user input the class must provide the framework with a user-displayable identifier, which will be displayed to the user when selecting from a list of already configured credential objects.

In this case we have decided that the username should be an obvious way of choosing between credentials.

The credentials class is now written but we need to tell the framework when to use it, to do this we simply implement another method on the preset class:

```
+ (Class)credentialsClass {
      return [RMEmberCredentials class];
}
```

When this method is implemented the framework will look for any stored credentials of this class and if it cannot find any prompt the user to create one before starting preset configuration.

Writing an Upload Task

Now we have our persistence set up it is time to write the core of the plugin's functionality, the code that actually deals with uploading our files.

The framework class is incredibly simple, consisting of just three methods: `initWithPreset:uploadInfo:`, `upload` and `cancel`. The lifecycle of the object is pretty much explained in those methods. The object is initialised with any information needed, we then call `upload` on the object and if the user wishes to cancel then `cancel` is called.

Throughout the lifecycle of the object it keeps the framework informed with what is happening using notifications. During the upload process the task posts `RMUploadTaskReceivedProgressNotificationName` notifications so the framework can keep the user informed of the progress of the upload. When the task has received a response from it's upload, it posts a `RMUploadTaskDidFinishTransactionNotificationName` notification. On completion, when the task has completed any bookkeeping, the task posts an `RMUploadTaskCompletedNotificationName` notification. Should any of the tasks' operations fail, such as uploading, or adding a photo to an album; the error should be returned to the framework using the `RMUploadTaskErrorKey` key in the `RMUploadTaskDidFinishTransactionNotificationName` notification.

Taking a look at task initialisation first, the job here is to store any information we are going to require for the upload. The `uploadInfo` parameter is a key-value container for any metadata being provided for the upload. There is a set of pre-defined keys that it may contain or you may inject your own by using a custom metadata view, which will be elaborated on later. The important thing to remember is that it may or may not contain any of the keys, no key's presence is guaranteed.

The predefined keys are as follows:

- `RMUploadFileURLKey` : The location of the file on disk.
- `RMUploadTitleKey` : The title the user has given for the file.
- `RMUploadOriginalDateKey` : The date the file was originally created in its current form.
- `RMUploadOriginalURLKey` : If the file represents a website, this is the URL it represents.
- `RMUploadDescriptionKey` : The description the user has given when setting metadata.
- `RMUploadTagsKey` : An array of strings which the user has tagged the image.

© Realmac Software 2009-2010

As opposed to pulling out just the information we need in this method we will store the whole `uploadInfo` and `preset` objects and worry about what we need in the `upload` method later on. This makes for easier code maintenance as if something changes in your `preset` or the `uploadInfo` you only have to make changes in the `upload` method.

Therefore our initialisation method is pretty barebones:

```
- (id)initWithPreset:(RMEmberPreset *)preset uploadInfo:(id)info
{
     self = [super initWithPreset:preset uploadInfo:info];
     if (self == nil) return nil;

     // Note: add anything else you need to setup

     return self;
}
```

© Realmac Software 2009-2010

However it means that all information is available to us when writing the `upload` method. The upload method is fairly hefty and certainly wouldn't sit well in this document, the Ember project is open source for you to take a look at however.

What I will do here is describe the structure with some pseudo code here.

```
- (void)upload
{
     // Note: pull out any keys required from the uploadInfo and build an instance of
RMUploadURLConnection or NSURLConnection
     // ... create URL connection
     newConnection.delegate = self;
     self.uploadConnection = newConnection;
     // ... create URL request
     [self.uploadConnection sendFilePostRequestWithParamaters:parameters toURL:url];
}
```

So really what we are doing is setting up an instance of `RMUploadURLConnection` to do our upload work and sending it on its way. Most of the code in the class can be found in the delegate callbacks for `RMUploadURLConnection`, parsing the returned XML from Ember and posting the relevant notifications:

```
- (void)connection:(RMUploadURLConnection *)connection uploadProgressed:(float)currentProgress
{
     NSDictionary *progressDict = [NSDictionary dictionaryWithObjectsAndKeys:
                                   [NSNumber numberWithFloat:currentProgress],
                                   RMUploadTaskProgressCurrentKey,
                                   [NSNumber numberWithFloat:1.0],
                                   RMUploadTaskProgressTotalKey,
                                   nil];
     [[NSNotificationCenter defaultCenter]
postNotificationName:RMUploadTaskDidReceiveProgressNotificationName object:self
userInfo:progressDict];
}
```

When our upload progresses we pass along a notification along with its relevant progress keys to keep the framework informed, as returns progress between 0.0 and 1.0, we say 1.0 as the total upload possible and whatever the connection just passed us as the current progress.

```
- (void)connection:(RMUploadURLConnection *)connection didFailWithError:(NSError *)error
{
     NSDictionary *userInfo = [NSDictionary dictionaryWithObject:error
forKey:RMUploadTaskErrorKey];
     [[NSNotificationCenter defaultCenter]
postNotificationName:RMUploadTaskDidFinishTransactionNotificationName object:self
userInfo:userInfo];

     [self _postCompletionNotification];
}
```

If the connection fails we set the error key in the user info and then post the `RMUploadTaskDidFinishTransactionNotificationName` notification.

© Realmac Software 2009-2010

The completion method is similar:

```
- (void)connection:(RMUploadURLConnection *)connection didCompleteWithData:(NSData *)data
{
	NSError *documentError = nil;
	NSXMLDocument *document = [[NSXMLDocument alloc] initWithData:data options:0
error:&documentError];

	if (document == nil) {
		NSDictionary *errorInfo = [NSDictionary dictionaryWithObjectsAndKeys:
									RMLocalizedStringInSelfBundle(@"The server
returned an invalid response."), NSLocalizedDescriptionKey,
								nil];
		NSError *error = [NSError errorWithDomain:RMEmberBundleIdentifier code:0
userInfo:errorInfo];

		NSDictionary *notificationInfo = [NSDictionary dictionaryWithObjectsAndKeys:
							error, RMUploadTaskErrorKey,
							nil];
		[[NSNotification defaultCenter]
postNotificationName:RMUploadTaskDidFinishTransactionNotificationName object:self
userInfo:notificationInfo];

		[self _postCompletionNotification];
		return;
	}

	// Parse XML for errors…

	NSError *locationXPathError = nil;
	NSString *imageLocation = [document stringValueForXPath:@"/response/item/permalink"
error:&locationXPathError];

	NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
	[userInfo setObject:document forKey:RMEmberXMLDocumentKey];

	if (imageLocation != nil) {
		[userInfo setObject:[NSURL URLWithString:imageLocation]
forKey:RMUploadTaskResourceLocationKey];
	}

	if (self.destination.targetCollection != nil)
		[self addToCollectionFromDocument:document];

	if (self.destination.targetCategory != nil)
		[self submitImageToCategoryFromDocument:document];

	// Completed notification is only sent when all metadata is also set.
}
```

In this code chunk we can see some basic parsing around to find the resultant location of the uploaded image, we set that against the `RMUploadTaskResourceLocationKey` key. However we also set a custom key in the user-info of the completion notification. As this is passed on to our completion view UI we will be able to access it.

We also deal with setting some additional metadata that was configured in the preset. The implementation details are unimportant as they are just simple calls to the Ember API.

At this point all we have to do is add a basic cancel method to complete the upload task:

© Realmac Software 2009-2010

```
- (void)cancel
{
      [self.uploadConnection cancel];
      [[NSNotificationCenter defaultCenter]
postNotificationName:RMUploadTaskDidCompleteNotificationName object:self];
}
```

As we have kept a reference to the connection as a property it is easy to cancel the upload and tell the framework that the cancelling has completed. You should post the completed notification after being sent `cancel` as soon as possible.

© Realmac Software 2009-2010

Adding the User Interface

The Plugin

The framework requests user interface components, in the form of `NSViewController` objects through the the `RMUploadPlugin` subclass instance. There are four possible opportunities for plugins to inject their own views into an application using the framework.

- The preset configuration view
- The credentials configuration view
- The view displayed to configure upload metadata

Your plugin is only required to provide a view for the preset configuration, unless you provide credentials in which case a credential configuration view is also required.

In our Ember plugin we will return user interface at all possible opportunities, here is the implementation of our `EmberPlugin` class:

```
@implementation EmberPlugin

- (RMUploadPresetConfigurationViewController *)
credentialsConfigurationViewControllerForCredentials:(RMUploadCredentials *)credentials; {
     RMEmberCredentialsViewController *controller = [[RMEmberCredentialsViewController alloc]
initWithNibName:@"CredentialsConfigurationView" bundle:[NSBundle
bundleWithIdentifier:RMEmberBundleIdentifier]];
     return controller;
}

- (RMUploadPresetConfigurationViewController *)presetConfigurationViewControllerForPreset:
(RMUploadPreset *)preset
{
     RMEmberPresetConfigurationViewController *controller =
[[RMEmberPresetConfigurationViewController alloc] initWithNibName:@"PresetConfigurationView"
bundle:[NSBundle bundleWithIdentifier:RMEmberBundleIdentifier]];
     return controller;
}

- (NSViewController *)additionalMetadataViewControllerForPresetClass:(Class)presetClass
{
     return [[NSViewController alloc] initWithNibName:@"MetadataView" bundle: [NSBundle
bundleWithIdentifier:RMEmberBundleIdentifier]];
}

@end
```

Although you must provide a preset, not all plugins need to provide a preset configuration view controller.

If your preset uses a credentials object, you can return nil for your preset configuration view controller, skipping any preset setup. This allows you to provide credentials with the possibility of adding preset options later.

© Realmac Software 2009-2010

As you can see we have chosen to create custom view controllers for all but one of the views and are returning them. For the last however, because the view is so simple the entire UI is handled by binding controls directly to the `representedObject`.

In each case, the relevant object is set as the `representedObject` for each view controller:

- The `RMUploadPreset` instance for the preset configuration view
- The `RMUploadCredentials` instance for the credentials configuration view
- The `uploadInfo` KVC container for the additional metadata view

It is also worth noting that the relevant objects are passed to the methods so if your plugin supports multiple services you can return the appropriate view controller for that service.

The Preset Configuration View

The preset configuration view is the UI displayed when a user is either creating or editing a new preset. This is required for any upload plugin. You are expected to provide a view and controller that allow the user to specify any information that is stored as part of a preset.

Taking our Ember preset configuration view as an example we have got quite a simple looking form, but beneath the hood there are some complexities that make the user's life extremely simple when setting up presets.

As you can return a custom view controller you have the flexibility to achieve pretty much anything you like with any of the views returned to the framework. Although, as the `representedObject` is set on the view controller for you you can get away with most of the UI work by binding controls directly to it.

The Credentials Configuration View

Similar to the preset configuration, this view is required if you support credentials in your plugin and is used to configure those credentials.

The Additional Metadata View

The framework allows plugins to request additional information about items to upload from the user. This is achieved by allowing a plugin to inject a view where the user configures the upload's metadata. Through that UI the plugin can add key/value pairs to the `uploadInfo` object. Ensure any keys you use are namespaced to your plugin to avoid conflicts.

There are a few things to keep in mind when returning your additional metadata view. Firstly, the view will be placed in a scroll view, secondly the `uploadInfo` object is the `representedObject` of the returned controller, therefore you can bind values on your interface easily. Also try and keep the view as small and simple as possible, it is going to be presented stacked with other metadata views, anything too complex will notably degrade the user experience.

© Realmac Software 2009-2010

Common Pitfalls

If your plugin fails to load, first check these points:

Check your Binary
- Check that your plugin is compiled for 64 bit Intel.
- Check that your plugin is compiling with Garbage Collection set to required.
- Confirm that your bundle extension is ‘uploader’

Check your Info.plist
- Check that the `RMUploadPluginBundleMinimumFrameworkBundleVersion` key is present and has the correct value.
- Check that the principal class is correct.
- Check that your preset class names are correct.

If your bundle still doesn’t load, please contact Realmac Software developer technical support, attaching your bundle and providing as much information as possible.

© Realmac Software 2009-2010

Appendix - Embedding Envelope Themes

NB: this is a Courier only API and doesn't affect your plugins functionality.

It's possible to embed envelope themes and autosuggest a theme on a per-preset basis. For more information on this, see the 'Courier Envelope SDK' and simply include the relevant keys in your Info.plist.

If your plugin bundle provides an envelope theme for a preset, the envelope key should be returned from `+[RMUploadPreset autoselectTrayImageKey]`.

© Realmac Software 2009-2010