Table Of Contents

Quicksilver Plug-ins

A haphazard reference by Rob McBroom qsplugins@skurfer.com with contributions from

- Henning Jungkurth
- Patrick Robertson
- Etienne Samson

Intro

I wanted to write a plug-in. I found a bit of documentation on adding Actions, but that was about it. The rest was trial and error and stabbing in the dark (with some help from looking at existing plug-ins). I hope to spare others some of this frustration by documenting what I discovered. This is mainly a collection of random notes, rather than a step-by-step guide.

I'll assume you're working on a plug-in called "Blah" when giving examples.

Getting Started

To compile most plug-ins, you'll need to build Quicksilver itself from source. (It'll put the stuff you need in `/tmp/QS`.) To build your plug-in with the Debug configuration, you'll need to first build Quicksilver with its Debug configuration. Same for the Release configuration.

To create a new plug-in:

1. Get the Xcode Project template from the GitHub repository
2. Copy `Quicksilver Plug-in.xctemplate` to `~/Library/Developer/Xcode/Templates/Application Plug-in`
3. Create a new project in Xcode. You should see "Quicksilver Plug-in" in the "Application Plug-in" category.
4. There are three types to choose from. Standard, Scripting Bridge (which is like Standard but adds the Scripting Bridge framework), and Interface which is mainly useless for now.
5. I would tell it not to create a Git repo automatically, since there will be a lot of meaningless changes right out of the gate.

Once the project has been created, there are some manual steps left to complete.

1. Delete `bltrversion` from the project, removing references only. We needed it to get copied into the project's directory, but we don't need to see it in Xcode.
2. Do a Release build of Quicksilver to get some things created in `/tmp/QS`.
3. Drag `/tmp/QS/Configuration` to the files area of your project. Uncheck "Copy itemsâ€¦" and select "Create groupsâ€¦".
4. Select the new "Configuration" folder in Xcode and bring up the Utilities panel on the right.
5. Change the Location to "Absolute Path".
6. Select the project at the top of the list of files on the left.
7. Select the project on the right where you see "PROJECT" and "TARGETS".
8. Under the Info tab, in the Configurations section, expand both Debug and Release
9. Under "Based on Configuration File", set the Debug project to "Debug" and the target to "QSPlugIn_Debug". Do the same under Release (using the configurations with "Release" in the name, of course.)
10. Switch over the the project's Build Settings tab. Select anything that's bold and hit Delete. (Bold indicates a setting in the project is overriding a setting from the configurations we just added. By "deleting" it, all you're doing is reverting it to what's in the configuration. This will prevent you from having to change you plug-in's build settings as things change in the core application.)
11. Personally, I don't like the way the files are laid out by default, so I rename some and move some around but that's a lot of work. If you have an opinion on such matters, I'll assume you don't need me to tell you how to make the changes.

12. Create a `.gitignore` file in the root directory of the project. It should contain at least:

```
.svn
.DS_Store
._.*
```

```
*.o
build
Makefile
autom4te.cache
*~
*.pbxuser
*.perspectivev3
*.mode1v3
*.mode2v3
*.pbxuser
*.tm_build_errors
*.tmproj
Developer.xcconfig
*.xcworkspace
*.xcuserdatad
```

You should now have a plug-in that will build. If you want to use Scripting Bridge, there are a few more steps.

1. Drag the application(s) you want scripting support for into your project. Don't let it copy the files.
2. Again, use the Utilities panel to change the location for each application to "Absolute Path".
3. Select the project on the left, then select the target (under TARGETS) on the right.
4. Open the "Build Phases" tab.
5. Drag the application(s) to the "Compile Sources" section and make sure it's the first thing on the list.
6. Go to the "Build Rules" tab.
7. Add a new build rule, set Process to "Source files with names matching" and enter `*.app`
8. Select "Custom script" and enter this command:

   ```
   sdef "$INPUT_FILE_PATH" | sdp -fh -o "$DERIVED_FILES_DIR" --basename "$INPUT_FILE_BASE"
   --bundleid `defaults read "$INPUT_FILE_PATH/Contents/Info" CFBundleIdentifier`
   ```

9. Add an output file with the path `$(DERIVED_FILES_DIR)/$(INPUT_FILE_BASE).h`
10. Add a line to import the resulting header file to your plug-in's `.pch` file. For example, if you added Safari, you'd enter:

    ```
    #import "Safari.h"
    ```

    This file won't exist until you build for the first time.

Many of the items in the template's `Info.plist` are named "SomethingTemplate". This prevents Quicksilver from having to process unused features. If you need to use settings in any of these sections, just rename them, removing "Template" from the end.

bltrversion

A note on this executable might be helpful. All this does is modify your project's `Info.plist` every time you build to increment the build version. There are a couple of things to keep in mind as a result.

- Git will almost always show this file as having changed. Don't commit the change unless it's actually what you want.
- You may have built the thing 250 times since the last commit, but do you really want the build number to jump that far when you were probably just testing some small changes? No. So either manually revert it to the last committed value, or increment it to something reasonable.
- You can normally install an updated plug-in by double-clicking it in Finder, but it will only replace the old one if it has a higher build number. If you lowered the number recently in order to commit something more reasonable, you might have to manually remove the old version (with a higher build number) before you can install the latest.

Documentation

Before we get into the technical stuff, make sure you **provide adequate documentation** with any plug-in you create.

This also goes for existing plug-ins that you're updating or fixing. Quicksilver plug-ins are notoriously

light on documentation and as a result, people don't discover some of their best features for years. If you're not afraid to screw with the code, don't be afraid to screw with the documentation as well.

The best place to document your plug-in is in the `Info.plist` by adding an `extendedDescription` to the `QSPlugIn` section. It can contain HTML.

Property List Overview

Throughout this document are numerous references to things that go in a plug-in's property list. Refer back here if you want to see it all in one place.

QSPlugIn

author (string) : Put your name(s) here

description (string) : A one-line description of your plug-in's purpose. (I'm not sure where this shows up in Quicksilver's UI.)

extendedDescription (string) : This is what the user will see when they click the "info" button in the Plug-ins preferences. It can be text or HTML (recommended).

icon (string) : An image or bundle ID

categories (array) : This is a list of categories you'd like the plug-in to be listed under. The full list of categories can be found under "All Plug-ins" in the Preferences.

hidden (boolean) : This is generally only used by internal plug-ins bundled with Quicksilver. It's a boolean that tells whether or not your plug-in should show up on the list in the preferences.

relatedBundles (array) : This is a list of bundle IDs for applications or plug-ins that are related

requiresBundle (array) : This is a list of bundle IDs for things that are required. Quicksilver won't load the plug-in if these things aren't on the system. For example, the Cyberduck plug-in is no use unless Cyberduck is installed, so it lists `ch.sudo.cyberduck` here.

webIcon (string) : You can provide a URL that points to an image here.

infoFile (string) : If you want to include a README (text or RTF) with your plug-in, you can specify the file name. I don't think this is accessible through Quicksilver's interface yet though.

QSActions

The QSActions section contains an array of dictionaries. The key should be an internal name for your action, like `MakeStuffBetter`. Each action can have the following children.

name (string) : The name the user will see (and search for) in the interface

description : A one-line description of what the action does

icon (string) : The image that will appear for this action in the interface

directTypes (array) : A list of types this action applies to. It won't show up unless the object in the first pane has one of the types on this list.

directFileTypes (array) : If one of your direct types is `NSFilenamesPboardType` (files), you can limit the types of files that match by providing a list of extensions here.

indirectTypes (array) : The types of objects that are allowed in the third pane (if this action uses it)

indirectOptional (boolean) : Specifies whether or not the third pane is required

actionClass (string) : The class that contains the method referred to by `actionSelector`

actionSelector (string) : The method that does the work for this action. For instance, the `MakeStuffBetter` action might refer to a method called `makeStuffBetter:`. If the action takes an argument (like the name of a cute animal) in the third pane, you specify the name of the argument here as well, as in `makeStuffBetter:usingCuteAnimal:`.

reverseArguments (boolean) : If true, the arguments will be sent to the `actionSelector` in the opposite order. Using the example above, an action that allowed cute animals in the first pane and something to make better in the third pane could reuse the `makeStuffBetter:usingCuteAnimal:` method you've already written.

alternateAction (string) : The name of the alternate action as it appears in your property list under `QSActions`

validatesObjects (boolean) : This tells Quicksilver whether or not to run `validActionsForDirectObject:`. That method gives you much more fine-grained control in cases where simply checking the type isn't sufficient to decide whether or not an action should be available.

displaysResult (boolean) : If true, the Quicksilver interface will reappear after your action runs. You would normally only use this if you action returns a `QSObject`.

enabled (boolean) : Whether or not this action should be available by default when your plug in is installed for the first time

precedence (float) : A number between 0.0 and 4.0. You should generally only use this if the action applies to a new type created by your plug-in, and not for any of the types Quicksilver knows of by default. Most of the built-in default actions have a low precedence and you can very easily overpower them here. Users would not appreciate this action suddenly becoming the default for files or text.

runInMainThread (boolean) : If true, this forces the action to be run in the main thread (for timers or actions that calculate a delay)

hidden (boolean) : Whether or not the action should be directly usable by the user. You might want to set this for an alternate action.

QSRegistration

This section gives Quicksilver additional information about your plug-in.

QSTypeDefinitions

This allows your custom type(s) to appear in the Actions section of the preferences. Any new types provided by your plug-in should be described here. The key is the name of a type as defined in your code. Each entry should have two children: `name` and `icon`.

QSBundleChildHandlers

This lists bundles that should have children provided by your plug-in. (Generally this is used to allow right-arrowing into an existing application like Mail or Address Book.) The key should be the bundle ID, and the value should be the name of a class that contains a `loadChildrenForObject:` method.

QSApplicationActions

These are actions that only appear when a specific application is in the first pane. The key should be the bundle ID of the application. Under this is a dictionary of actions. I'm not entirely sure how to set this up. See the iTunes module for an example.

QSProxies

This defines proxy objects provided by the plug-in.

QSPresetAdditions

This section defines catalog entries that should be added by default under Modules in the Catalog preferences.

QSTriggerAdditions

This section defines triggers that should be added by default in the Triggers preferences.

Actions

The information in Ankur's tutorial is pretty good, but there are some things it doesn't mention.

Direct and Indirect Objects

You may see references to direct and indirect objects. Basically, direct objects are the things in the first pane of Quicksilver's interface and indirect objects are the thing in the third pane.

If you're looking at existing code, these are usually referred to as `dObject` and `iObject`. You don't have to use these names, but it will save you some typing if you plan to copy and paste a lot from examples.

Displaying Results

Something to note about the `displaysResult` option in `Info.plist`: This only means that Quicksilver will pop the interface back up *if* your action returns a `QSObject` to it. If you have an action that may or may not return something to the interface, it appears to be safe to enable this.

Also note that if your action returns a `QSObject` to Quicksilver, that *will* become the thing in the first pane, whether the UI is re-displayed or not. If not, the user will see the object you returned the next time he invokes Quicksilver.

Alternate Actions

Alternate actions are a very powerful feature of Quicksilver. When a user selects an action in the second pane, they can hit â†© to run it, or they can hit âŒ˜â†© to run an alternate (if one is defined). Generally an alternate action will be similar to the main action, but modified in some way.

You create alternate actions just like any other action by adding a section for it in `Info.plist` and adding code in `BlahAction.m` for it. The main thing to note is that an action doesn't need to be active (checked in the preferences) to work as an alternate. You can define which of the actions in your plug-in are enabled or disabled by default, so an alternate could be something you never intend for the user to see/use directly or it could be one of the "normal" actions that you want to be conveniently accessible. Users can, of course, enable or disable actions themselves after your plug-in is installed.

Alternate actions should be able to take the same number of arguments as the main action. You can't define an alternate that requires something in the third pane for an action that doesn't use the third pane.

To define an alternate for an action, add an item named `alternateAction` to your action's properties in `Info.plist`. The value for `alternateAction` should be the name of another action as it appears under "QSActions" in the same plist.

Reverse Actions

Some actions are reversible. For instance, web searches can be either

```
Search Engine â†’ Search Forâ€¦ â†’ [search terms]
```

or

```
[search terms] â†’ Find Withâ€¦ â†’ Search Engine
```

Reverse actions are added to the plist just like any other action. They should be more or less identical to the "forward" version of the action, with two key changes:

1. The name of the action should be different.
2. There should be a boolean called `reverseArguments` set to TRUE

The Comma Trick

If you want your actions to support the comma trick (where a user can select several things in the

first pane and act on them), you'll have to write actions to loop through all the objects and do the appropriate thing. If you expect to get several objects of the same type, the easiest technique involves getting a quick representation of each. `QSObject`s have several properties but you can specify which one should be associated with a specific type when the object is created using `setObject:forType:`. Assuming you set these as strings, you can get at them like this:

```
for (NSString *userSelectedThing in [dObject arrayForType:QSBlahType])
{
    // code
}
```

Note that the above code will also work if the user just passes a single item to your action.

If you need to get the full details of the `QSObject`s passed in, you can do something like the following in your action's code:

```
for (QSObject *individual in [dObject splitObjects])
{
    NSString *thisGuy = [individual name];
    NSString *somethingWeNeed = [individual objectForMeta:@"key"];
}
```

`splitObjects` is safe to use on single `QSObject`s, so it will work for any case, but sometimes you need to know which you have. The best way to tell if you're dealing with single vs. multiple objects is to look at the count:

```
if ([dObject count] == 1)
{
    // single object
} else {
    // multiple objects
}
```

For actions that won't support the comma trick, you should use the object validation process to prevent those actions from ever showing up, rather than checking for multiple objects in the action itself.

Validating Objects

validActionsForDirectObject

Example: `- (NSArray *)validActionsForDirectObject:(QSObject *)dObject indirectObject:(QSObject *)iObject`

If you set `validatesObjects` to TRUE on an action in `Info.plist`, that action will never appear unless you add some corresponding code. This method must exist in `BlahAction.m` and return the name of any actions that you think will be safe to run based on whatever validation steps you've gone through. It should return an array of action names as strings. Perhaps you don't want the actions to work with strings longer than 100 characters or something. You could accomplish that with:

```
NSString *myString=[[dObject arrayForType:QSBlahType] lastObject];
NSMutableArray *newActions=[NSMutableArray arrayWithCapacity:1];
if ([myString length] < 100) {
    [newActions addObject:@"FirstAction"];
    [newActions addObject:@"SecondAction"];
}
return newActions;
```

Keep in mind that Quicksilver has to run this validation code on every object that ever gets selected in the first pane in order to determine if this action is applicable or not before it can display the actions in the second pane. And not just for your plug-in. It's every validation method from every active plug-in every time a user selects something. It should go without saying at this point, but make sure your `validActionsForDirectObject:` method returns quickly!

validIndirectObjectsForAction

Example: `- (NSArray *)validIndirectObjectsForAction:(NSString *)action directObject:(QSObject *)dObject`

This method gets called when going to the third pane (for actions that provide one). It should return an array of `QSObject`s that are valid for that action. For instance, selecting a file and choosing the "Move Toâ€¦" action will only show folders in the third pane.

Personally, I use it to make sure the third pane comes up in text entry mode instead of regular "search" mode. To accomplish that, do something like this:

```
return [NSArray arrayWithObject:[QSObject textProxyObjectWithDefaultValue:@""]];
```

You can set the default text to whatever and that text will appear (selected) in the third pane. The above example just makes it blank.

If the third pane will ask for some sort of search text, you should stick to the convention of other plug-ins (and Mac OS X in general) and set the default text to be whatever the user last searched for. To accomplish that:

```
NSString *searchString=[[NSPasteboard pasteboardWithName:NSFindPboard] stringForType:NSStringPboardType];
return [NSArray arrayWithObject:[QSObject textProxyObjectWithDefaultValue:searchString]];
```

This method doesn't seem to be affected by `validatesObjects` in the plist.

AppleScript Actions

It's possible to define actions that exist only as AppleScript and not as Objective-C. These actions will look a little bit different in your `Info.plist`.

1. `actionClass` should be set to "QSAppleScriptActions".
2. `actionScript` should be set to the name of an AppleScript file, like "Blah.scpt". This script will need to be copied into your plug-in's `Resources` folder. A single AppleScript file can handle multiple actions (by using different `actionHandler`s).

3. `actionHandler` should be the name of a function in the script, like "do*this*thing" (assuming the script has something like this in it).

    ```
    on do_this_thing(objectFromQuicksilver)
        tell application "XYZ" to lick the face of objectFromQuicksilver
    end do_this_thing
    ```

You'd omit the `actionSelector` for AppleScript actions.

I'm not sure what happens if you send multiple objects (via the comma trick) to an AppleScript action.

Application Actions

Applications Actions are actions that only appear in the list of actions when a specific application if selected in the first pane of Quicksilver. Examples of this are the actions "Next Song", "Play - Pause", "Previous Song" that are provided by the iTunes plugin and only appear if iTunes is selected in the first pane. Or the "Get New Mail" and "Open New Mail" actions for Mail.app.

These actions are defined in the `Info.plist` of the plugin, in `QSRegistration` â†' `QSApplicationActions` â†' bundle identifier â†' action identifier. The same pattern can be used as in the normal actions (inside `QSActions`) **OR** a `QSCommand`. For an example of how `QSApplicationActions` are defined, see the Info.plist of the Apple Mail Plugin.

Commands

A `QSCommand` is something that contains a direct object, an action, and an indirect object. In other words, it encapsulates something you would normally put together in Quicksilver's interface.

Commands can be created by users (using the âŒƒâ†© "encapsulate" keystroke), by code (using methods from the `QSCommand` class), and in a plug-ins `Info.plist` (Under `QSRegistration` â†' `QSCommands`). Triggers typically (always?) refer to commands for obvious reasons.

One thing to note is that commands don't require either a direct or indirect object. For example, the iTunes plug-in comes with a number of predefined triggers like "Play", "Increase Volume", "Increase Rating", etc. These always do the same thing. There's no need to specify an object for them to act

on. As a result, the commands these triggers refer to contain only an action.

When defining commands in the property list, the contents will vary greatly depending on the `actionID`. The `actionID` can refer to any action, but there are only a few that make sense. Below are examples of three common types.

For commands that call an Objective-C method somewhere, use “QSObjCSendMessageAction”:

```
<key>QSiTunesMute</key>
<dict>
  <key>command</key>
  <dict>
    <key>actionID</key>
    <string>QSObjCSendMessageAction</string>
    <key>directArchive</key>
    <dict>
      <key>data</key>
      <dict>
        <key>qs.action</key>
        <dict>
          <key>actionClass</key>
          <string>QSiTunesControlProvider</string>
          <key>actionSelector</key>
          <string>volumeMute</string>
          <key>icon</key>
          <string>iTunesIcon</string>
          <key>name</key>
          <string>Mute</string>
        </dict>
      </dict>
    </dict>
    <key>directID</key>
    <string>QSiTunesVolumeMute</string>
  </dict>
</dict>
```

Be sure the key and `directID` are not the same.

For commands that should invoke Quicksilver, and allow searching a specific subset of objects, use “QSObjectSearchChildrenAction”:

```
<key>QSiTunesSearchArtists</key>
<dict>
  <key>command</key>
  <dict>
    <key>actionID</key>
    <string>QSObjectSearchChildrenAction</string>
    <key>directArchive</key>
    <dict>
      <key>data</key>
      <dict>
        <key>com.apple.itunes.qsbrowsercriteria</key>
        <dict>
          <key>Result</key>
          <string>Artist</string>
          <key>Type</key>
          <string>Artist</string>
        </dict>
      </dict>
    </dict>
  </dict>
  <key>name</key>
  <string>Search Artists</string>
</dict>
```

For commands that are handled by an AppleScript file somewhere, use “AppleScriptRunAction”:

```
<key>QSiTunesMute</key>
<dict>
  <key>command</key>
  <dict>
    <key>actionID</key>
    <string>AppleScriptRunAction</string>
    <key>directResource</key>
    <dict>
```

```
      <key>bundle</key>
      <string>com.blacktree.Quicksilver.QSiTunesPlugIn</string>
      <key>path</key>
      <string>Contents/Resources/Scripts/Mute.scpt</string>
    </dict>
  </dict>
</dict>
```

Trigger Presets

Your plug-in can provide pre-defined triggers. These go under `QSTriggerAdditions` and look something like this:

```
<key>ID</key>
<string>QSiTunesNextSongTrigger</string>
<key>command</key>
<string>QSiTunesNextSongCommand</string>
<key>defaults</key>
<dict>
  <key>characters</key>
  <string>ïœƒ</string>
  <key>keyCode</key>
  <integer>124</integer>
  <key>modifiers</key>
  <integer>9961768</integer>
  <key>type</key>
  <string>QSHotKeyTrigger</string>
</dict>
<key>name</key>
<string>Next Song</string>
<key>set</key>
<string>iTunes</string>
```

Only the type is required under `defaults` if you don't want to specify a shortcut. The `set` refers to the name of a group in the Trigger preferences. To create a new group for your plug-in's triggers, add something like this under `QSRegistration` â†' `QSTriggerSets`:

```
<key>QSBlah</key>
<dict>
  <key>icon</key>
  <string>GenericDocument</string>
  <key>name</key>
  <string>Blah</string>
</dict>
```

Adding to the Catalog

Info.plist

There are two sections in `Info.plist` you should know about.

`QSPresetAdditions` is where you can list catalog entries that should be present by default when your plug-in is loaded. (These are the things you see in the "Modules" section of the Catalog.) These can be normal existing sources, like "QSFileSystemObjectSource" (a.k.a. File & Folder Scanner), or they can refer to a new source you define yourself.

If you want to see what these look like, open `~/Library/Application Support/Quicksilver/Catalog.plist` and look at the `customEntries` section. In fact, one easy way to create a catalog preset for your plug-in is to create it as a custom entry in Quicksilver, then copy/paste it from `Catalog.plist`. The only thing you may want to add by hand is an `icon`, which can go right under `name`. You'd also change `sources` to something like `QSBlahSource` if you're defining your own.

The other important section in the `Info.plist` is `QSRegistration`. If you're creating your own source instead of using one of the built-ins, you'll need to add it here so Quicksilver will use it. You'll want to make sure there's a dictionary called `QSObjectSources`. Create an item under this and set both the name and value to the name of the class in `BlahSource.m` that provides the source, such as `QSBlahSource`.

Also under `QSRegistration` are some settings you'll need if you want users to be able to "right

arrow" into things (which can also be done with '/'). As far as I can tell, you use `QSBundleChildHandlers` to allow right arrowing into things that don't "belong" to your plug-in and you use `QSObjectHandlers` for things your plug-in will create.

The thing you use `QSBundleChildHandlers` for is generally some application that's already on the system and related to your plug-in, but has no direct knowledge of, or support for Quicksilver. The left hand side should be a bundle identifier, like `com.apple.Finder` and the right hand side should be the name of a class in your plug-in that contains a `loadChildrenForObject:` method.

`QSObjectHandlers` is how you can allow users to "right arrow" into things you create. On the left should be a type of object in Quicksilver and on the right should again be the name of a class in your plug-in that contains a `loadChildrenForObject:` method. This definitely works with types you invent (see Custom Types below) and I assume it will work for existing types, but haven't tried it. This setting must be present for the `setQuickIconForObject:` to get called as well, even if you won't be defining children. It might look like this:

```
<key>QSObjectHandlers</key>
<dict>
    <key>QSBlahType</key>
    <string>QSBlahSource</string>
    <key>QSBlahAnotherType</key>
    <string>QSBlahSource</string>
</dict>
```

Note that you can use the same "source" class for all types, or create a separate class for each one. If you use the same source, the `loadChildrenForObject:` method will just have to figure out what to set as children based on the type of object passed to it.

Code

There are some methods you'll want to define in `BlahSource.m` and Quicksilver will call them at the appropriate time. This is not a complete list of available methods, just the ones I've run across and (I think) figured out. You can look at existing plug-ins for some example code. Note that many of these methods are only called by Quicksilver if you define something for `QSObjectHandlers` in `Info.plist`.

The information for an entry (from `Info.plist`) can be pulled in like this:

```
NSMutableDictionary *settings = [theEntry objectForKey:kItemSettings];
```

Again, existing plug-ins can give you some examples of how to use this info.

objectsForEntry

Example: `- (NSArray *)objectsForEntry:(NSDictionary *)theEntry`

This method does the work of adding your objects to the catalog. It can do whatever you want it to do (parse files, get information over the network, query a database, etc.) as long as it returns an array of `QSObject`s.

There are several ways to create a `QSObject`. Here are a couple:

```
QSObject *myObject = [QSObject objectWithString:string];
QSObject *myObject = [QSObject objectWithName:name];
QSObject *myObject = [QSObject URLObjectWithURL:url];
```

There is a full list of all the methods available for creating objects in the Creating section of this manual.

All of the available ways and the differences between them isn't entirely clear, so experiment and see what works for you. There are also things you may want to do to your objects prior to adding them to the array to be returned, like `setIdentifier:`, `setLabel:`, `setDetails:`, `setName:`, `setObject:forType:` etc. There's a whole section in this document on `QSObject` that provides more details.

indexIsValidFromDate

Example - `(BOOL)indexIsValidFromDate:(NSDate *)indexDate forEntry:(NSDictionary`

```
*)theEntry
```

Quicksilver runs this during scheduled catalog updates (every 10 minutes by default) to ask “Is this catalog entry still up to date?” How you determine the answer to this question is up to you. If this method returns YES/TRUE, Quicksilver moves along without doing anything. If this method returns NO/FALSE, Quicksilver will attempt to update the entry by calling the `objectsForEntry:` method.

You can have this method do nothing but return TRUE of FALSE unconditionally, and that’s exactly what it does for many plug-ins. Just keep in mind that this will either cause your source to never get updated (except manually) or cause it to get updated on every single rescan (which may be a performance concern, depending on what it does).

`indexDate` and `theEntry` are passed in by Quicksilver. I believe the `indexDate` tells you when the entry was last updated in the catalog. `theEntry` contains information about the entry that you might need (file paths, URLs, etc.).

NOTE: This method is not consulted when your plug-in is first installed or when a user selects the entry in the catalog and hits the “rescan” button manually. In other words, if you’re checking for errors with your source here to avoid unnecessary rescans, that’s good, but you still need to check for errors in `objectsForEntry:` as well, because it will be called at least once.

setQuickIconForObject

Example: `- (void)setQuickIconForObject:(QSObject *)object`

If you call `setIcon:` when adding objects to the catalog, they will show up at first but quickly disappear. Calling `setIcon:` from this method instead will work more reliably. Example:

```
[object setIcon:[QSResourceManager imageNamed:@"GenericNetworkIcon"]];
```

Quicksilver calls this in real-time as objects need to be displayed, which is why it appears more reliably. (It’s probably more efficient that way too, compared to storing an icon to everything in the catalog whether it gets used or not.)

This method does not seem to be called for `QSObject`s created using `objectWithString:`. That method makes a call to `loadIcon:` internally, which must tell Quicksilver that it already has an icon and doesn’t need to go look for one.

loadChildrenForObject

Example: `- (BOOL)loadChildrenForObject:(QSObject *)object`

This is very similar to `objectsForEntry:`, but instead of adding things to the catalog, it loads them on the fly when you right arrow into the parent object. You just need to create an array of `QSObject`s, but instead of returning them, you assign them to the parent object like this:

```
[object setChildren:children];
return TRUE;
```

These can be newly created objects or objects that are already in the catalog. If the children have already been added to the catalog, the most efficient thing to do is to retrieve them instead of recreating them. (See the later sections on getting specific objects by ID and by type.)

Read up on identifiers in the QSObject section before creating/using objects as children.

objectHasChildren

Example: `- (BOOL)objectHasChildren:(QSObject *)object`

This method is used to indicate to the user that an object is browsable (can be right arrowed into). If it returns `YES`, a little arrow is displayed on the very right of the object.

If you used `loadChildrenForObject:` so the user can right arrow into an object, you should indicate that to the user by having this method return `YES`.

NOTE: Not returning `YES` will hide the little arrow indicator, but the user will still be able to arrow into the object, if you provided children for the object by using `loadChildrenForObject:`.

isVisibleSource

Example: `- (BOOL)isVisibleSource`

This method determines whether or not your source shows up on the drop-down list for adding new custom catalog entries. As far as I can tell, the name users see will be something ugly like "QSBlahSource". The mapping that generates pretty names seems to be embedded with Quicksilver itself.

Return FALSE to keep this off of the drop-down. Return TRUE if you want users to be able to add custom entries of this type to their catalog. Keep in mind that you will also need to provide them with some sort of interface to do so.

Custom Types

You may want to add a new type of thing to Quicksilver's catalog that doesn't conform to the existing types Quicksilver knows about. The main reason you'd want to do this is to make your actions apply only to relevant entries. For instance, you may want "SSH to host" as the default for remote hosts. Though hostnames and FQDNs are just strings and can be added to Quicksilver easily, simply adding them to the catalog as strings would cause the default action to be "Open URL" or "Large Type" in most cases. You would either have to choose the SSH action manually every time, or move it up in priority making it the default for all text and URLs, which probably isn't what you want.

Source

To use a custom type when adding objects to the catalog, the first thing to do is add something like this to `BlahSource.h`:

```
#define QSBlahType @"QSBlahType"
```

Then, in your `objectsForEntry:` method in `BlahSource.m`, you would do something like this on each QSObject before adding it to the array:

```
[myObject setObject:someIdentifier forType:QSBlahType];
```

Now, when you're adding actions to the `Info.plist`, they can use this as a value under `directTypes` or `indirectTypes`. You will almost certainly want to assign other types in addition to the one you've created. See "Types" in the QSObject section for details.

Actions

Speaking of actions, if your action(s) need to refer to your custom type, you should add something like this to `BlahAction.h`:

```
#define QSBlahType @"QSBlahType"
```

And then in the various action methods in `BlahAction.m`, you can test for objects of that type with something like this:

```
if ([dObject containsType:QSBlahType]) {}
```

QSObject

Your plug-in will communicate with Quicksilver pretty much entirely through `QSObject` objects. That is, Quicksilver will send `QSObject`s to your methods, and if you need to send something back to Quicksilver, you'll return either a single `QSObject` or an array of them. Here's what I know about creating them and using those previously created.

Creating

You'll need to create `QSObject`s if adding things to the catalog and also if your action needs to send something back to Quicksilver (to end up in the first pane of the UI). This section is largely written with adding to the catalog in mind, but the information should be useful for returning objects for display as well (because it's so dead simple by comparison). There seem to be several methods for creating new `QSObject`s. Here are a couple I've run across.

- objectWithString
- objectWithName
- makeObjectWithIdentifier
- objectWithURL
- fileObjectWithPath

`objectWithName:` simply creates an empty object and stores the name in the object's dictionary of metadata. `objectWithString:` will use the string to set the name, but it also sets the type to `QSTextType` and then does some fancy analysis (using QSObject (StringHandling)'s `sniffString` method) to figure out if the string should be treated differently in any way. This is how strings can be turned into URL objects or file objects automagically. Also of note: The Mac OS X spell checker has no problem with "automagically".)

Most of the time, you probably want to take advantage of Quicksilver's "smarts" and use `objectWithString:` but if for some reason you just want it to take your string and not mess with it, use `objectWithName:`.

In my experience, you don't need to do anything more than the above to create a usable object, but you will probably want to add more details such asâ€¦

Identifier

An identifier for your `QSObject` doesn't seem to be required (or perhaps Quicksilver will pick one internally if you don't explicitly set one) but I recommend always setting one. In any case, it's important to understand how they're used. You set one like this:

```
[myObject setIdentifier:someString];
```

In addition to actually storing the identifier string in your object, this causes Quicksilver to add the object to an internal registry.

If you choose an identifier that is already in use, you will **replace the existing object** with your new one in the registry. This has a couple of implications. First, you should obviously pick a unique string to use as an identifier. Second, if you end up "recreating" various objects (either to appear in the third pane, or to act as children for another object) be careful what you do with them. Here are some tips:

1. For objects that just serve some temporary purpose in the UI, don't even set an identifier.
2. Wherever possible, use existing objects instead of recreating identical ones.
3. If you want to use an existing object, but modify it in some way, create a new empty `QSObject` and assign it attributes from the existing object as needed.

Name

If you've created an object using on of QSObject's more 'basic' methods such as `objectWithIdentifier:` or even `[[QSObject alloc] init]` it is important to set the object's name using the `setName:@"some name"` method.

This is the most prominent string in the UI (unless you set a label), and Quicksilver matches against it when a user is typing.

Type

To set a type for an object, do something like this:

```
[myObject setObject:someObject forType:QSBlahType];
```

Each `QSObject` has an array of types, so you can call this method multiple times, each with a different type, if you want. That deserves some further discussion. Most actions in Quicksilver only work with certain types. You will probably want to make built-in actions or actions defined in other plug-ins do something useful with your objects, which may be of a new type Quicksilver has never heard of. The "Paste" and "Large Type" actions are prime examples. You can't very well expect various other plug-ins (or Quicksilver itself) to add support for your new type. This is where you want to use an existing type to get things working.

Hopefully, by now, you've figured out what the object is supposed to be for each type you assign. It

will be something that actions supporting that type can make use of. I'll use "Paste" as an example. The "Paste" action works with a few types I'm sure, but the simplest one is `QSTextType`. The thing you want to paste in that case will be a string. Your `QSObject` probably has many strings (name, label, details, etc.) so how can you tell Quicksilver which to spit out when running the "Paste" action? Like this:

```
[myObject setObject:thisString forType:QSTextType];
```

Now, when a user selects this object and uses an action that works with `QSTextType`, it will use `thisString`.

If you set multiple types, you should also set one as primary.

```
[myObject setPrimaryType:QSBlahType];
```

I don't think you can use `setPrimaryType:` alone. You must also call `setObject:forType:` using the same type at some point.

Here are all the types Quicksilver declares in `QSTypes.h`. I don't know what might be special about each one, but examples of their creation and use are probably available in existing code.

```
QSFilePathType
QSTextType
QSAliasDataType
QSAliasFilePathType
QSURLType
QSEmailAddressType
QSContactEmailType
QSContactPhoneType
QSContactAddressType
QSFormulaType
QSActionType
QSProcessType
QSMachineAddressType
QSABPersonType
QSNumericType
QSIMAccountType
QSIMMultiAccountType
QSPasteboardDataType
QSCommandType
QSHandledType
```

Label

This is optional, but if you want the text that appears most prominently in the user interface to be something other than the name, you can set this using `setLabel:@"some string"`.

This is also the string that Quicksilver matches against when a user is typing, so keep that in mind. You can make things easier for users to find in the catalog by messing with the label. (The name will still be searchable as well, and will appear instead when matched.)

Setting the label will not change the appearance of the search result list, only the main QS panes. In the result list, the label wont be displayed. Instead, the name will still be the most prominent thing displayed.

Details

The "details" string is the text that appears smaller underneath the label in most Quicksilver interfaces. Set it using `setDetails:@"some string"`.

Setting this explicitly is optional but if you don't, Quicksilver will pick something. In some cases, it uses the identifier. Keep this in mind if your identifiers aren't very nice to look at.

Icon

You can set an icon for a `QSObject` using the `setIcon:` method. I'm not even sure what it takes (an `NSImage` probably?), but Quicksilver makes it easy for you with `QSResourceManager`. Just do something like this:

```
[myObject setIcon:[QSResourceManager imageNamed:@"GenericNetworkIcon"]];
```

I don't recommend using the `setIcon:` method when adding things to the catalog. (See the note on `setQuickIconForObject:` for an explanation.) However, using this method directly from one of your actions does work if you want to set the icon temporarily for something you're sending back to the Quicksilver UI. (That too will go away, but this is only an issue if the user leaves the object you return up in the first pane.)

FYI, if you want to use one of the standard system icons for something, many can be found in `/System/Library/CoreServices/CoreTypes.bundle/Contents/Resources`. You can refer to them using only the file's name (no path and no ".icns" extension). To use the icon for an application, use its bundle identifier, like `com.apple.Mail`. Finally, you can also provide the full path to an image file. Here are some examples:

```
[QSResourceManager imageNamed:@"com.apple.Mail"]
[QSResourceManager imageNamed:@"/Users/me/Pictures/Some.icns"]
```

You can also use `QSResourceManager` to get specific icons from inside a bundle's Resources folder, but that requires a bit of extra work. For example, to use the bookmark menu icon in Safari's bundle, you need to define a section in the property list called `QSResourceAdditions`, then define different icons under it, such as:

```
<key>SafariBookmarkMenuIcon</key>
<dict>
  <key>bundle</key>
  <string>com.apple.Safari</string>
  <key>resource</key>
  <string>tiny_menu.tiff</string>
</dict>
```

That will allow you to refer to the icon in your code with something like

```
[QSResourceManager imageNamed:@"SafariBookmarkMenuIcon"]
```

Finally, if you want to use the icon for a particular file type, here's one method using the extension.

```
[myObject setIcon:[[NSWorkspace sharedWorkspace] iconForFileType:@"pdf"]];
```

Children

Usually, you'll load and set children on-the-fly using `loadChildrenForObject:`, but it does seem to be possible to set them when an object is first created as well. Assuming you have created an array of other `QSObject`s called "children", you can set them like this:

```
[myObject setChildren:children];
```

These child objects will appear in the UI if a user select your object and hits â†' or /.

Parent

If you have set children for an object so you can right arrow into them, these children should also know about their parent object. This builds a proper hierarchy and helps QS to figure out what to do when you arrow back out of the list of children objects (using â†).

So, before using `[myObject setChildren:children]` to set children for an object, for each of the children `QSObject`s, you should set the parent ID to the identifier of the parent `QSObject`. For example by doing the following for each `QSObject` in the `children` array:

```
[childObject setParentID:[myObject identifier]];
```

Then, once the user arrows back out of the list of children, the correct object will be selected in the main QS pane and also be highlighted in the result list. If this isn't set, the selected child object will remain in the QS pane and the first result will be highlighted in the result list. This can lead to much confusion, so it should be avoided.

Arbitrary Metadata

In addition to the standard things like name, label, icon, and type, `QSObject`s provide an `NSMutableDictionary`. You can store just about anything in here, so the possibilities for your plug-in

get really interesting with this. If your objects are files, perhaps you want to store their size here. Perhaps you want to store width, height, and resolution for an image. Maybe you want to store a thumbnail image for the object. (It doesn't have to be a string.) Perhaps you just like to waste memory on other people's computers and want to store the lyrics to your favorite song.

Dictionaries are just lists of key:value pairs, so to add your custom thing to the object, you would do something like:

```
[myObject setObject:value forMeta:@"key name"];
```

The key name is just a string you'll use to refer to this thing in the dictionary later. The value can by any type of Cocoa object.

There'll be more on this in the next section. As a real-world example, one use I found for this was customizing icons for things in the catalog. As mentioned elsewhere, using `setIcon:` when things are added to the catalog is the wrong approach. You could probably store the icon as metadata, but this is inefficient. A better approach is to just store some identifying string as metadata, and move the logic that uses that data over to the `setQuickIconForObject:` method.

Using

When Quicksilver passes `QSObject`s to your actions, here are some of the ways to get information out of them.

String Values

To get a quick string representation of an object for whatever reason:

```
NSString *blah = [dObject stringValue];
```

This is probably best for objects you didn't create and don't know the contents of. The `stringValue` method tries do do some smart things, but if none that pans out, it will call `displayName`. The `displayName` method will return the label if set, otherwise it returns the name.

If you're familiar with what an object contains and you want to get the name or label specifically, you can use `[dObject name]` or `[dObject label]`.

Metadata

To retrieve metadata that you may have set when adding objects to the catalog:

```
NSString *value = [dObject objectForMeta:@"key name"];
```

Of course you would only use `NSString` if that was the type of object you stored.

This metadata has many possibilities, but two of the more obvious are:

1. Checking to see which actions will work on this object (using `validActionsForDirectObject`)
2. Directing the behavior of your actions

"combined objects"

If a user sends multiple things to your action using the comma trick, then `[dObject stringValue]` will return "combined objects". I give examples of how to loop through one of these combined objects in the Actions section.

Getting Specific Objects by ID

The `objectWithIdentifier:` method will return an existing `QSObject` if you know what it's ID was set to (probably because you set it yourself). It's a class method, so you don't call it on any particular object.

```
QSObject *thisGuy = [QSObject objectWithIdentifier:uuid];
```

The objects are looked up and returned from the object registry. I'm not sure exactly what the relationship is between the object registry and the catalog. Looking at the Quicksilver source indicates that they're independent, but in practice, I haven't been able to request an object by

identifier unless it is in the catalog. If that's true, then the object your requesting must have been added to the catalog by `objectsForEntry:` at some prior point.

Getting Specific Objects by Type

You can use `QSLib` to retrieve objects from Quicksilver's catalog by type. The most common uses for this are to restrict the objects that appear in the third pane, and to build up a list of children for another object.

To simply get everything of a certain type, call this:

```
NSArray *objects = [QSLib arrayForType:@"QSBlahType"];
```

To get the same list but sort by rank (typically determined by how often a user accesses the object):

```
NSArray *objects=[QSLib scoredArrayForString:nil inSet:[QSLib arrayForType:@"QSBlahType"];
```

More Information

A lot can be figured out by looking through `QSObject.m`. If you really want to track down *everything* `QSObject` provides, search for "@implementation QSObject" in the Quicksilver source and stare in horror at the number of results.

Building and Testing

When testing a newly built plug-in, if things don't work as expected, don't go back to the code looking for the problem right away. It may be that you're testing against cached data from older code. This process seems to ensure that you're actually testing your latest build.

1. Install the newly built plug-in
2. Relaunch Quicksilver
3. Rescan the catalog entries for your plug-in
4. Relaunch Quicksilver again

If you're trying to work on one of the existing plug-ins and you can't get it to build, here are a couple of things that have worked for me.

1. Build Quicksilver from source (Release, not Debug). The process puts a lot of things in `/tmp/QS/build/release/` that the source for the plug-ins expect to find there.
2. Tell Xcode to do a "Release" build of the plug-in instead of "Debug". (I'm not sure what all the ramifications are with this, but it seems to work.)

Debugging Plugins

To be able to debug plugins using an actual debugger instead of just NSLog() statements, there are a few steps that need to be done. Most of this information is from an old thread on the QS mailinglist. Some of these steps are already described in the **Getting Started** section in the beginning of this guide, just check again to make sure everything is set up correctly.

1. Make a backup of everything Quicksilver.
2. Disable "Load symbols lazily" in the XCode preferences, under the Debugging tab.
3. Build the debug version of Quicksilver
4. Make sure you have the Configuration folder from the Quicksilver source included in you project.
5. Make the build settings of you project and target "based on" the settings from the Configuration folder:
    1. Go into the Project settings (Project -> Edit Project Settings), into the Build tab
    2. On the top left, select *Debug* from the "Configuration:" drop-down.
    3. Then, on the bottom right, where it say "Based On:" select *Debug* from the drop-down.
    4. Delete everything that shows up in bold from the list of settings ("Show:" *Settings Defined at This Level* shows all of these settings).
    5. Go into the Options for the target (âŒ¥âŒ˜E), into the build settings again.
    6. Select for "Configuration:" *Debug* again.
    7. Select for "Based On:" *QSPlugIn_Debug*
    8. Again, delete everything that shows up in bold, **except** the "Product Name". Set the

"Product Name" to something meaningful (like the plugin's name :-))
9. You might want to do the same steps for the *Release* configuration, but you wont need to just for debugging.

6. Build the debug version of the plugin. You should end up with a *.qsplugin in /tmp/QS/build/Debug/.
7. Install the newly build plugin (by double-clicking on the .qsplugin file). This ensures that your newly build plugin is loaded in Quicksilver, not a different/older/not-debug-version.
8. Quit Quicksilver if running.
9. Go into the Quicksilver project in XCode and start debugging (âŒ¥âŒˆY).
10. Once Quicksilver has started, go into the debugger console and check if the debugger is still running. If not, there should be a message like `Debugger stopped. Program exited with status value:0`. In that case, quit Quicksilver again and start debugging again. This time the debugger should stay active even after Quicksilver finished starting.
11. Now, any breakpoints you set in the plugin project should stay blue (instead of turning yellow) and should break when reach.

This sounds quite complicated, but once you have set it up correctly, debugging is pretty straight forward: Build plugin, install plugin, debug Quicksilver.

Links

Some things that helped me.

Official Docs - Almost worth your time. Almost.

The Existing Plug-ins - Loads and loads of undocumented, uncommented examples that you may or may not be able to make sense of.

The Remote Hosts Module - Not to plug my own work, but it's the only frakking plug-in I've seen with a single frakking comment to explain what's going on, so it might be worth a look.

Ankur Kothari's nice tutorial - This really helps get you started, but doesn't document anything other than creating basic actions.

- Creating a plug-in
- Info.plist
- Actions

Some information on setting up a project for Scripting Bridge