XBMC Swift Documentation

Release 1.0.1

Jonathan Beluch

May 04, 2015

Contents

Welcome to the documentation for xbmcswift2. xbmcswift2 is a small framework to ease development of XBMC addons. Whether you are an experienced addon developer, or just coding your first addon, you'll find benefits to using xbmcswift2.

This documentation is divided into several parts. If you are new, you should start with the *Installation* and then move on to *Quickstart*. If you would prefer a more detailed walkthrough, try the *Tutorial*.

To get a deeper understanding of xbmcswift2, check out *URL Routing*, *Caching and Storage* and the complete *API* reference. For specific code samples, check out *Patterns*. If you are upgrading from xbmcswift, check out the *Upgrading from xbmcswift* page.

For a list of XBMC addons which use xbmcswift2, see *Addons Powered by xbmcswift2*.

CHAPTER 1

Basic Info

1.1 Installation

Note: The purpose of xbmcswift2 is to have the ability to run the addon on the command line as well as in XBMC. This means that we will have to install xbmcswift2 twice, once for the command line and once as an XBMC addon.

The XBMC version of xbmcswift2 is a specially packaged version of the main release. It excludes some CLI code and tests. It also contains XBMC required files like addon.xml.

The easiest way to get the most recent version of xbmcswift2 for XBMC is to install an addon that requires xbmcswift2. You can find a list of such addons on the *Addons Powered by xbmcswift2* page. The other options is download the current XBMC distribution from https://github.com/jbeluch/xbmcswift2-xbmc-dist/tags and unpack it into your addons folder.

Now, on to installing xbmcswift2 for use on the command line.

1.1.1 virtualenv

Virtualenv is an awesome tool that enables clean installation and removal of python libraries into a sequestered environment. Using a virtual environment means that when you install a library, it doesn't pollute your system-wide python installation. This makes it possible to install different versions of a library in different environments and they will never conflict. It's a good habit to get into when doing python development. So, first we're going to install virtualenv.

If you already have pip installed, you can simply:

```
$ sudo pip install virtualenv
```

or if you only have easy_install:

```
$ sudo easy_install virtualenv
```

I also like to use some helpful virtualenv scripts, so install virtualenv-wrapper as well:

```
$ sudo pip install virtualenv-wrapper
```

1.1.2 Creating a Virtual Environment

Now we can create our virtualenv:

```
$ mkvirtualenv xbmcswift2
```

When this completes, your prompt should now be prefixed by *(xbmcswift2)*. The new prompt signals that we are now working within our virtualenv. Any libraries that we install via pip will only be available in this environment. Now we'll install xbmcswift2:

```
$ pip install xbmcswift2
```

Everything should be good to go. When you would like to work on your project in the future, issue the following command to start your virtual env:

```
$ workon xbmcswift2
```

and to deactive the virtualenv:

```
$ deactivate
```

You should check out the *Running xbmcswift2 on the Command Line* page next.

1.2 Quickstart

If you haven't already installed xbmcswift2, head over to the *installation* page.

The purpose of xbmcswift2 is to empower plugin writers to develop and debug their plugins faster. This is faciliated by:

- A bootstrap script to create an empty addon complete with folder structure and required files.
- Seamless testing of addons by enabling an addon to be run on the command line or in XBMC. xbmcswift2 handles mocking the xbmc python modules to ensure your addon will run (in a limited fashion) outside of XBMC *without* any code changes.
- Basic URL routing code, so you can focus on writing the web parsing code specific to your plugin, and not deal with repeated boilerplate and url parsing.
- A library of helpful functions and code patterns to enhance your addon's functionality.

1.2.1 Introduction to XBMC Addons

Before going any further, you should already be familiar with the general file structure and necessary files for an XBMC addon. If not, please spend a few minutes reading about addons in the XBMC wiki.

1.2.2 Creating the Plugin Skeleton

xbmcswift2 comes with a helpful console script that will create a plugin skeleton for you, including all the necessary folders and files to get started. Simply run *xbmcswift2 create* and answer a few questions to personalize your addon.

Below is an example session:

```
$ xbmcswift2 create

    xbmcswift2 - A micro-framework for creating XBMC plugins.
    xbmc@jonathanbeluch.com
    --

I'm going to ask you a few questions to get this project started.
```

```
What is your plugin name? : Hello XBMC
Enter your plugin id. [plugin.video.helloxbmc]:
Enter parent folder (where to create project) [/private/tmp]:
Enter provider name : Jonathan Beluch (jbel)
Projects successfully created in /private/tmp/plugin.video.helloxbmc.
Done.
```

1.2.3 Hello XBMC

If you navigate to the newly created folder `plugin.video.helloxbmc`, you'll find an `addon.py` exactly like the one below.

```
from xbmcswift2 import Plugin


plugin = Plugin()


@plugin.route('/')
def index():
    item = {
        'label': 'Hello XBMC!',
        'path': 'http://s3.amazonaws.com/KA-youtube-converted/JwO_25S_eWE.mp4/JwO_25S_eWE.mp4',
        'is_playable': True
    }
    return [item]


if __name__ == '__main__':
    plugin.run()
```

The above code is a fully functioning XBMC addon (not that it does much!). So what does the code do?

1. After importing the Plugin class, we create our plugin instance. xbmcswift will parse the proper addon name and id from the addon.xml file.
2. We are using the `plugin.route` decorator on the `index` function. This binds a url path of '/' to the index function. ('/' is the default URL path).

   Note: The url rule of '/' must always exist in a plugin. This is the default route when a plugin is first run.
3. The index function creates a single dict with some key/vals. This is how you create a listitem using xbmcswift2. At a minimum, most items have a `path` and `label`. The `is_playable` flag tells XBMC that this is a media item, and not a URL which points back to an addon.
4. We return a list from the index function, that contains a single item. For a typical xbmcswift2 view, this is the proper way to add list items.
5. We call `plugin.run()` to run our plugin. It is imperative that this line is inside the `__name__` guard. If it is not, your addon won't run correctly on the command line.

1.2.4 Running Addons from the Command Line

One of the shining points of xbmcswift2 is the ability to run plugins from the command line. To do so, ensure your working directory is the root of your addon folder (where you addon.xml file is located) and execute `xbmcswift2 run`.:

```
$ xbmcswift2 run
2012-05-02 19:02:37,785 - DEBUG - [xbmcswift2] Adding url rule "/" named "index" pointing to functio
2012-05-02 19:02:37,798 - DEBUG - [xbmcswift2] Dispatching / to once
2012-05-02 19:02:37,798 - INFO - [xbmcswift2] Request for "/" matches rule for function "index"
----------------------
 #  Label       Path
----------------------
[0] Hello XBMC! (None)
----------------------
```

Right away we can see the output of our plugin. When running in the CLI, xbmcswift2 prints log messages to STDERR, so you can hide them by appending `2>/dev/null` to the previous command.. Below the logs we can see a simple display of our listitems, in this case a single item.

See *Running xbmcswift2 on the Command Line* for a more detailed explanation of running on the command line.

1.2.5 URL Routing

Another advantage of using xbmcswift2, is its clean URL routing code. This means you don't have to write your own code to parse the URL provided by XBMC and route it to a specific function. xbmcswift2 uses a a path passed to the `route()` decorator to bind a URL to a function. For example, a route of `/videos/` will result in a URL of `plugin://plugin.video.helloxbmc/videos/` calling the decorated function.

It's even possible to pass variables to functions from the URLs. You might have a function like this to list videos for a given category:

```
@plugin.route('/categories/<category>/')
def show_videos(category):
    '''Display videos for the provided category'''
    # An incoming URL of /categories/science/ would call this function and
    # category would have a value of 'science'.
    items = get_video_items(category)
    return plugin.finish(items)
```

Currently, there is no type coercion, so all variables plucked from URLs will be strings.

Now we have a way of directing incoming URLs to specific views. But how do we link list items to other views in our code? We'll modify our Hello XBMC addon:

```
@plugin.route('/')
def index():
    items = [
        {'label': 'Hola XBMC!', 'path': plugin.url_for('show_label', label='spanish')},
        {'label': 'Bonjour XBMC!', 'path': plugin.url_for('show_label', label='french')},
    ]
    return items


@plugin.route('/labels/<label>/')
def show_label(label):
    # Normally we would use label to parse a specific web page, in this case we are just
    # using it for a new list item label to show how URL parsing works.
    items = [
        {'label': label},
    ]
    return items
```

Let's run our plugin interactively now to explore:

```
$ xbmcswift2 run interactive
2012-05-02 19:14:53,792 - DEBUG - [xbmcswift2] Adding url rule "/" named "index" pointing to functio
2012-05-02 19:14:53,792 - DEBUG - [xbmcswift2] Adding url rule "/labels/<label>/" named "show_label"
2012-05-02 19:14:53,793 - DEBUG - [xbmcswift2] Dispatching / to interactive
2012-05-02 19:14:53,794 - INFO - [xbmcswift2] Request for "/" matches rule for function "index"
--------------------------------------------------------------------
 #  Label         Path
--------------------------------------------------------------------
[0] Hola XBMC!    (plugin://plugin.video.helloxbmc/labels/spanish/)
[1] Bonjour XBMC! (plugin://plugin.video.helloxbmc/labels/french/)
--------------------------------------------------------------------
Choose an item or "q" to quit: 0

2012-05-02 19:14:59,854 - INFO - [xbmcswift2] Request for "/labels/spanish/" matches rule for functi
---------------------------------------------
#  Label   Path
---------------------------------------------
[0] ..      (plugin://plugin.video.helloxbmc/)
[1] spanish (None)
---------------------------------------------
Choose an item or "q" to quit: q

$ python addon.py interactive
--
[0] Hola XBMC! (plugin://plugin.video.helloxbmc/labels/spanish/)
[1] Bonjour XBMC! (plugin://plugin.video.helloxbmc/labels/french/)
Choose an item or "q" to quit: 0
--
[0] spanish (None)
```

We've introduced a few new topics here.

- We passed `interactive` as a positional argument to the `xbmcswift2 run` command. This enables us to interact with the list items rather than just print them once and exit.
- We've used `url_for()` to create a url pointing to a different view function. This is how view functions create list items that link to other functions.
- Our function `show_label` requires an argument 'label', so we pass a keyword argument with the same name to url_for.
- To set the url for a list item, we set the 'path' keyword in the item dictionary.
- xbmcswift2 display a list item of '..', which is simliar to XBMC's '..' list item. This enables you to go back to the parent directory.

To learn more about URL routing and other available options, check out the <API> or the <patterns page>.

1.2.6 Playing Media

The last thing we haven't covered is how to play an actual video. By default, all items returned are directory items. This means that they act as a directory for more list items, and its URL points back into the plugin. To differentiate playable media from directory items, we set `is_playable` to `True` in our item dictionary.

First, let's add a new view to play some media:

```
@plugin.route('/videos/')
def show_videos():
    items = [
```

```
        {'label': 'Calculus: Derivatives 1',
         'path': 'http://s3.amazonaws.com/KA-youtube-converted/ANyVpMS3HL4.mp4/ANyVpMS3HL4.mp4',
         'is_playable': True,
         }
    ]
    return plugin.finish(items)
```

As you can see, the URL value for *path* is a direct link to a video asset, we are not calling `url_for`. If you need to use XBMC's `setResolveUrl` functionality, see the patterns section for `plugin.set_resolved_url`.

Now let's update out item dictionary in show_label to add a path:

```
{'label': label, 'path': plugin.url_for('show_videos')},
```

Now, you have a fully functioning XBMC addon, complete with nested menus and playable media.

One more section before going off on your own!

1.2.7 Using xbmc, xbmcgui, xbmcaddon

You can always import and call any of the xbmc modules directly if you need advanced functionality that xbmcswift2 doesn't support. However, if you still want the ability to run plugins from the command line you should import the xbmc modules from xbmcswift2.

```
from xbmcswift2 import xbmc, xbmcgui
```

Since these modules are written in C, they are only available when running XBMC. To enable plugins to run on the command line, xbmcswift2 has mock versions of these modules.

1.2.8 Going further

This should be enough to get started with your first simple XBMC addon. If you'd like more information, please check out the detailed *Tutorial* and also review common *Patterns*.

1.3 Tutorial

At the end of this tutorial we're going to have a basic version of the Academic Earth Plugin. This plugin plays videos from http://www.academicearth.org/.

Since this tutorial is meant to cover the usage of xbmcswift2, we will not be covering HTML scraping. It makes sense to partition your scraping code into a separate module from your addon's core functionality. In this example, we're going to use a scraping library for academic earth that I already have written.

1.3.1 Creating the Plugin Structure

The first step is to create your working directory for your addon. Since this can be repetitive, xbmcswift2 provides a script which will create the necessary files and folders for you. So we'll do just that:

```
(xbmcswift2)jon@lenovo tmp  $ xbmcswift2 create

    xbmcswift2 - A micro-framework for creating XBMC plugins.
    xbmc@jonathanbeluch.com
    --
```

```
I'm going to ask you a few questions to get this project started.
What is your plugin name? : Academic Earth Tutorial
Enter your plugin id. [plugin.video.academicearthtutorial]:
Enter parent folder (where to create project) [/tmp]:
Enter provider name : Jonathan Beluch (jbel)
Projects successfully created in /tmp/plugin.video.academicearthtutorial.
Done.
```

If you `cd` into the created directory, you should see the familiar addon structure, including `addon.py`, `addon.xml`, `resourcres` directory, etc.

1.3.2 Setup for this Tutorial

To make this tutorial go a bit smoother, we're going to use some existing code which handles the scraping of the Academic Earth website. Download this file and extract it to `resources/lib/`.

```
$ cd resources/lib
$ wget https://github.com/downloads/jbeluch/xbmc-academic-earth/academicearth.tgz
$ tar -xvzf academicearth.tgz
$ rm academicearth.tgz
```

We should now have an `academicearth` directory in our lib directory.

Since our api library requires the use of BeautifulSoup, we'll need to add this as a depenency to our addon.xml file.

If you open the addon.xml file, you'll notice that xbmcswift2 is already in your dependencies:

```
<import addon="xbmc.python" version="2.0" />
<import addon="script.module.xbmcswift2" version="1.1.1" />
```

We'll add BeautifulSoup right after those lines:

```
<import addon="script.module.beautifulsoup" version="3.0.8" />
```

The last step is to install BeautifulSoup locally, so we can run our addon on the command line.:

```
$ pip install BeautifulSoup
```

1.3.3 Creating our Addon's Main Menu

Let's modify the the index function, to look like this:

```
@plugin.route('/')
def main_menu():
 items = [
     {'label': 'Show Subjects', 'path': plugin.url_for('show_subjects')}
 ]
 return items
```

The `main_menu` function is going to be our default view. Take note that is has the route of `/`. The first time the addon is launched, there will be no state information, so the requested URL will match '/'.

If you were to run the plugin now, you'd see an exception about a view not being found. This is because we are specifying a view name of 'show_subjects' but we don't have a view with that name! So let's create a stub for that view.

```
@plugin.route('/subjects/')
def show_subjects():
    pass
```

So now we have a basic plugin with two views. Keep in mind as we go along, that we can always run the plugin from the command line.:

```
$ xbmcswift2 run 2>/dev/null
------------------------------------------------------------
 #  Label    Path
 ------------------------------------------------------------
 [0] Subjects (plugin://plugin.video.academicearth/subjects/)
 ------------------------------------------------------------
```

1.3.4 Creating the Subjects View

Now let's add some logic to our `show_subjects` function.

```
@plugin.route('/subjects/')
def show_subjects():
    api = AcademicEarth()
    subjects = api.get_subjects()

    items = [{
        'label': subject.name,
        'path': plugin.url_for('show_subject_info', url=subject.url),
    } for subject in subjects]

    sorted_items = sorted(items, key=lambda item: item['label'])
    return sorted_items
```

You can see that we are going to be using our Academic Earth api module here. So we need to import the class before we instantiate it: `from resources.lib.academicearth.api import AcademicEarth`.

The call to `get_subjects` returns a list of Subject objects with various attributes that we can access.

So our code simply loops over the subjects and creates a dictionary for each subject. These simple dictionaries will be converted by xbmcswift2 into proper list items and then displayed by XBMC. The two mandatory keys are `label`, which is the text to display for the item, and `path`, which is the URL to follow when the item is selected.

Here, if the user selects a subject list item, we want to send them to the `show_subject_info` function. Notice we are also passing a keyword argument to the `url_for` method. This is the main way that we can pass information between successive invocations of the addon. By default, XBMC addons are stateless, each time a user clicks on an item the addon is executed, it does some work and then exits. To keep track of what the user was doing, we need to encode the information in the url. xbmcswift2 handles the url encoding as long as you pass the arguments to url_for.

The last lines of code in our view simply sort the list of dictionaries based on the label and then return the list.

The last step we need to take before running our addon is to stub out the `show_subject_info` view.

```
@plugin.route('/subjects/<url>/')
def show_subject_info(url):
    pass
```

Note that since we are passing a url argument to `url_for1`, we need to ensure our view can handle the argument. This involves creating a placeholder in the url, `<url>` and then ensuring our view takes a single argument, `url`. xbmcswift2 will attempt to match incoming URLs against the list of routes. If it finds a match, it will convert any instances of `<var_name>` to variables and then call the view with those variables. See *URL Routing* for more detailed information about routing.

Now let's run our plugin in interactive mode (for the sake of brevity I've replaces a lot of entries in the example output with `...`):

```
$ xbmcswift2 run interactive 2>/dev/null
------------------------------------------------------------
 #  Label    Path
------------------------------------------------------------
[0] Subjects (plugin://plugin.video.academicearth/subjects/)
------------------------------------------------------------
Choose an item or "q" to quit: 0

------------------------------------------------------------------------------------------------------------------
 #   Label                    Path
------------------------------------------------------------------------------------------------------------------
[ 0] ..                       (plugin://plugin.video.academicearth/)
[ 1] ACT                      (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 2] Accounting               (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 3] Algebra                  (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 4] Anthropology             (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 5] Applied CompSci          (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 6] Architecture             (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
...
[67] Visualization & Graphics (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
------------------------------------------------------------------------------------------------------------------
Choose an item or "q" to quit:
```

The first output we see is our main menu. Then we are prompted for an item to select (only 1 available in this case). When we select Subjects, we are then routed to our show_subjects view.

1.3.5 Adding Code to `show_subject_info`

Let's add some logic to our `show_subject_info` view:

```
@plugin.route('/subjects/<url>/')
def show_subject_info(url):
    subject = Subject.from_url(url)

    courses = [{
        'label': course.name,
        'path': plugin.url_for('show_course_info', url=course.url),
    } for course in subject.courses]

    lectures = [{
        'label': 'Lecture: %s' % lecture.name,
        'path': plugin.url_for('play_lecture', url=lecture.url),
        'is_playable': True,
    } for lecture in subject.lectures]

    by_label = itemgetter('label')
    items = sorted(courses, key=by_label) + sorted(lectures, key=by_label)
    return items
```

Most of this should look very similar to our code for show subjects. This time however, we have two different types of Academic Earth content to handle, courses and lectures. We want courses to route to `show_course_info`, which will list all of the lectures for the course. Lectures, however, are simply videos, so we want these list items to play a video when the user selects one. We are going to route lectures to `play_lecture`.

A new concept in this view is the `is_playable` item. By default, list items in xbmcswift2 are not playable. This

means that XBMC expects the list item to point back to an addon and will not attempt to play a video (or audio) for the given URL. When you are finally ready for XBMC to play a video, a special flag must be set. xbmcswift2 handles this for you, all you need to do is remember to set the `is_playable` flag to True.

There is another new concept in this view as well. Typically, if you tell XBMC that a URL is playable, you will pass a direct URL to a resource such as an mp4 file. In this case, we have to do more scraping in order to figure out the URL for the particular video the user selects. So our playable URL actually calls back into our addon, which will then make use of plugin.set_resolved_url().

1.3.6 Adding the `show_course_info` and `play_lecture` views

Let's add the following code to complete our addon:

```
@plugin.route('/courses/<url>/')
def show_course_info(url):
    course = Course.from_url(url)
    lectures = [{
        'label': 'Lecture: %s' % lecture.name,
        'path': plugin.url_for('play_lecture', url=lecture.url),
        'is_playable': True,
    } for lecture in course.lectures]

    return sorted(lectures, key=itemgetter('label'))


@plugin.route('/lectures/<url>/')
def play_lecture(url):
    lecture = Lecture.from_url(url)
    url = 'plugin://plugin.video.youtube/?action=play_video&videoid=%s' % lecture.youtube_id
    plugin.log.info('Playing url: %s' % url)
    plugin.set_resolved_url(url)
```

The `show_course_info` view should look pretty familiar at this point. We are just listing the lectures for the given course url.

The `play_lecture` view introduces some new concepts however. Remember that we told XBMC that our lecture items were *playable*. Since we gave a URL which pointed to our addon, we now have to use `plugin.set_resolved_url(url)`. This communicates to XBMC, that this is the *real* url that we want to play.

We are introducing one more layer of indirection here however. Since all of the content on Academic Earth is hosted on youtube, our addon would normally require lots of extra code just to parse URLs out of youtube. However, the youtube addon conveniently does all of that! So, we will actually set the playable URL to point to the youtube plugin, which will then provide XBMC with the actual playable URL. Sounds a bit complicated, but it makes addons much simpler in the end. Our addon simply deals with parsing the Academic Earth website, and leaves anything youtube specific to the youtube addon.

The last step is now to add youtube as a dependency for our addon. Let's edit the addon.xml again and add youtube:

```
<import addon="plugin.video.youtube" version="3.1.0" />
```

1.3.7 Conclusion

We're finished! You should be able to navigate your addon using the command line. You should also be able to test your addon directly in XBMC. I personally like to use symlinks to test my addons. On linux, you could do something like this:

```
$ cd ~/.xbmc/addons
$ ln -s ~/Code/plugin.video.academicearthtutorial
```

Note that you'll also have to install the xbmcswift2 XBMC distribution. The easiest way is to install one of the addons listed on the *Addons Powered by xbmcswift2* page. Since they all require xbmcswift2 as a dependency, it will automatically be installed. The other option is to download the newest released version from this page and unzip it in your addons directory.

1.4 The ListItem

xbmcswift2 prefers to represent XBMC list items as plain python dictionaries as much as possible. Views return lists of dictionaries, where each dict represents an XBMC listitem. The list of valid keys in an item dict can always be validated by reviewing the available arguments to `xbmcswift2.ListItem.from_dict()`. However, we'll go into more detail here.

Valid keys in an item dict are:

- label
- label2
- icon
- thumbnail
- path
- selected
- info
- properties
- context_menu
- replace_context_menu
- is_playable
- info_type
- stream_info

1.4.1 label

A required string. Used as the main display label for the list item.

1.4.2 label2

A string. Used as the alternate display label for the list item.

1.4.3 icon

A path to an icon image.

1.4.4 thumbnail

A path to a thumbnail image.

1.4.5 path

A required string.

For non-playable items, this is typically a URL for a different path in the same addon. To derive URLs for other views within your addon, use `xbmcswift2.Plugin.url_for()`.

For playable items, this is typically a URL to a remote media file. (One exception, is if you are using the set_resolved_url pattern, the URL will be playable but will also call back into your addon.)

1.4.6 selected

A boolean which will set the item as selected. False is default.

1.4.7 info

A dictionary of key/values of metadata information about the item. See the XBMC docs for a list of valid info items. Keys are always strings but values should be the correct type required by XBMC.

Also, see the related info_type key.

1.4.8 properties

A dict of properties, similar to info-labels. See http://mirrors.xbmc.org/docs/python-docs/xbmcgui.html#ListItem-setProperty for more information.

1.4.9 context_menu

A list of tuples, where each tuple is of length 2. The tuple should be (label, action) where action is a string representing a built-in XBMC function. See the XBMC documentation for more details and *Using the Context Menu* for some example code.

1.4.10 replace_context_menu

Used in conjunction with *context_menu*. A boolean indicating whether to replace the existing context menu with the passed context menu items. Defaults to False.

1.4.11 is_playable

A boolean indicating whether the item dict is a playable item. False indicates that the item is a directory item. Use True when the path is a direct media URL, or a URL that calls back to your addon where set_resolved_url will be used.

1.4.12 info_type

Used in conjunction with *info*. The default value is usually configured automatically from your addon.xml. See http://mirrors.xbmc.org/docs/python-docs/xbmcgui.html#ListItem-setInfo for valid values.

1.4.13 stream_info

A dict where each key is a stream type and each value is another dict of stream values. See http://mirrors.xbmc.org/docs/python-docs/xbmcgui.html#ListItem-addStreamInfo for more information.

1.5 Running xbmcswift2 on the Command Line

1.5.1 Commands

When running xbmcswift2 from the command line, there are two commands available, *create* and *run*. *create* is a script that will create the basic scaffolding and necessary files for an XBMC addon and personalize it by asking you a few questions. *run* enables you to debug your addon on the command line.

To see the command line help, simply execute `xbmcswift2 -h`. Both of the commands are explained further below.

create

To create a new addon, change your current working directory to a location where you want your addon folder to be created. Then execute `xbmcswift2 create`. After answering a few questions, you should have the basic addon structure in place.

run

When running an addon on the command line, there are three different run modes available, once, interactive, and crawl.

There is also a second positional argument, `url`, which is optional. By default, xbmcswift2 will run the root URL of your addon (a path of '/'), e.g. `plugin://plugin.video.academicearth/`. This is the same default URL that XBMC uses when you first enter an addon. You can gather URLs from the output of xbmcswift2.

The options `-q` and `-v` decrease and increase the logging level.

Note: To enable running on the command line, xbmcswift2 attempts to mock a portion of the XBMC python bindings. Certain functions behave properly like looking up strings. However, if a function has not been implemented, xbmcswift2 lets the function call pass silently to avoid exceptions and allow the plugin to run in a limited fashion. This is why you'll often see WARNING log messages when running on the command line.

If you plan on using the command line to develop your addons, you should always import the xbmc modules from xbmcswift2:

```
from xbcmswift2 import xbmcgui
```

xbmcswift2 will correctly import the proper module based on the environment. When running in XBMC, it will import the actual modules, and when running on the command line it will import mocked modules without error.

---

once

Executes the addon once then quits. Useful for testing when used with the optional `url` argument.:

```
$ xbmcswift2 run once # you can omit the once argument as it is the default

------------------------------------------------------------
 #  Label    Path
------------------------------------------------------------
[0] Subjects (plugin://plugin.video.academicearth/subjects/)
------------------------------------------------------------


$ xbmcswift2 run once plugin://plugin.video.academicearth/subjects/
------------------------------------------------------------------------------------------------------------------
 #   Label                    Path
------------------------------------------------------------------------------------------------------------------
 [ 0] ACT                      (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academi
 [ 1] Accounting               (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academi
 [ 2] Algebra                  (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academi
 [ 3] Anthropology             (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academi
 [ 4] Applied CompSci          (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academi
 ...
```

interactive

Allows the user to step through their addon using an interactive session. This is meant to mimic the basic XBMC interface of clicking on a listitem, which then brings up a new directory listing. After each listing is displayed the user will be prompted for a listitem to select. There will always be a `..` option to return to the previous directory (except for the initial URL).:

```
$ xbmcswift2 run interactive
------------------------------------------------------------
 #  Label    Path
------------------------------------------------------------
[0] Subjects (plugin://plugin.video.academicearth/subjects/)
------------------------------------------------------------
Choose an item or "q" to quit: 0

------------------------------------------------------------------------------------------------------------------
 #   Label                    Path
------------------------------------------------------------------------------------------------------------------
[ 0] ..                       (plugin://plugin.video.academicearth/)
[ 1] ACT                      (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 2] Accounting               (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 3] Algebra                  (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 4] Anthropology             (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
------------------------------------------------------------------------------------------------------------------
Choose an item or "q" to quit: 1

------------------------------------------------------------------------------------------------------------------
 #  Label                   Path
------------------------------------------------------------------------------------------------------------------
[0] ..                      (plugin://plugin.video.academicearth/subjects/)
[1] ACT - Science Test Prep (plugin://plugin.video.academicearth/courses/http%3A%2F%2Fwww.academicear
------------------------------------------------------------------------------------------------------------------
```

crawl

Used to crawl every available path in your addon. In between each request the user will be prompted to hit Enter to continue.:

```
$ xbmcswift2 run crawl 2>/dev/null
------------------------------------------------------------
 #  Label    Path
------------------------------------------------------------
[0] Subjects (plugin://plugin.video.academicearth/subjects/)
------------------------------------------------------------
Enter to continue or "q" to quit
------------------------------------------------------------------------------------------------------------------
 #   Label                    Path
------------------------------------------------------------------------------------------------------------------
[ 0] ACT                      (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 1] Accounting               (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 2] Algebra                  (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 3] Anthropology             (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
[ 4] Applied CompSci          (plugin://plugin.video.academicearth/subjects/http%3A%2F%2Fwww.academic
------------------------------------------------------------------------------------------------------------------
Enter to continue or "q" to quit
------------------------------------------------------------------------------------------------------------------
 #   Label
------------------------------------------------------------------------------------------------------------------
[ 0] A Cultural and Scientific Survey of the Eye and Vision
[ 1] Autism and Related Disorders
[ 2] Biology
[ 3] Core Science - Biochemistry I
[ 4] Darwin's Legacy
------------------------------------------------------------------------------------------------------------------
Enter to continue or "q" to quit
```

CHAPTER 2

Advanced User Guide

2.1 URL Routing

If you just need a basic introduction to URL routing, you can check out *Quickstart* or *Tutorial*. This page explains all of the options and more advanced usage of URL routing.

2.1.1 Encoding Parameters in URLs

You can pass parameters between view functions but putting instances of `<var_name>` in your url patterns that are passed to the route decorator.

For instance, if I had a view that should take a category_id, my call to route would look like so:

```
@plugin.route('/categories/<category_id>')
def show_category(category_id):
    pass
```

xbmcswift2 will attempt to match any incoming URLs against that pattern, so all of the following URLs would match:

- `/categories/123`
- `/categories/apples`
- `/categories/apples%3Dpears`

xbmcswift2 will then extract the part of the URL that machtes a pattern withing the angle brackets and will call your view with those variables (variables will always be strings). So if you have one pattern in your URL, your view function should take at least one argument.

2.1.2 Multiple Parameters

It's possible to pass more than one parameter.

```
@plugin.route('/categories/<category_id>/<subcategory>')
def show_category(category_id, subcategory):
    pass
```

The order of the arguments will always match the order specified in the URL pattern.

2.1.3 Multiple URL Patterns and Defaults

Sometimes it becomes useful to resuse a view for a different URL pattern. It's possible to bind more than one URL pattern to a view. Keep in mind however, that to use `url_for` unambiguously, you'll need to provide the *name* argument to `route` to differentiate the two.

```
@plugin.route('/categories/<category_id>', name='show_category_firstpage')
@plugin.route('/categories/<category_id>/<page>')
def show_category(category_id, page='0'):
    pass
```

So now two different URL patterns will match the show_category view. However, since our first pattern doesn't include <page> in the pattern, we'll need to provide a default to our function. We can either provide a default in the method signature, like above, or we can pass a dict for the `options` keyword argument to `route`.:

```
@plugin.route('/categories/<category_id>', name='show_category_firstpage', options={'page': '0'})
@plugin.route('/categories/<category_id>/<page>')
def show_category(category_id, page):
    pass
```

In these two examples, we would build urls for the different routes like so:

```
# For the show_category_firstpage view
plugin.url_for('show_category_firstpage', category_id='science')

# For the show_category view
plugin.url_for('show_category', category_id='science', page='3')
```

2.1.4 Extra Parameters

Ocassionaly you might need to pass an argument to a view, but you don't want to necessarily want to clutter up the URL pattern. Any extra keyword arguments passed to url_for, that don't match a variable name in the URL pattern, will be appended as query string arguments. They can then be accessed using `plugin.request.args`.

2.1.5 URL Encoding and Pickling

Currently all keyword arguments to `url_for` that match variable names in the URL pattern must be instances of basestring. This means ints must be converted first using `str()`. Arguments will then be urlencoded/urlunencoded by xbmcswift2.

Any extra arguments that will end up in the query string, will be pickled and urlencoded automatically. This can be advantageous, if you want to store a simple list or something. However, pickling and urlencoding a python object can result in a very large URL and XBMC will only handle a finite length, so use this feature judiciously.

2.2 Caching and Storage

xbmcswift2 offers a few options for caching and storage to help improve the user experience of your addon. swift offers a simple storage mechanism that allows you to store arbitraty python objects to use between requests.

> **Warning:** The current implementation of xbmcswift2's storage is very basic and is not thread safe. If your addon does background calls via the context menu and manipulates storages in these backgound threads, you might run into some issues.

2.2.1 Storing Arbitraty Python Objects

All caches/storage objects in xbmcswift2 act like python dictionaries. So to get a cache, simply call the `get_storage` method.

```
people = plugin.get_storage('people')

# now we can use people like a regular dict
people['jon'] = 'developer'
people.update({'dave': 'accountant'})

people.items()
# [('jon', 'deveoper'), ('dave', 'accountant')]
```

Caches are automatically persisted to disk each time an addon finishes execution. If you would like to sync the cache to disk manually, you can call `cache.sync()` directly. However, this is not normally necessary.

2.2.2 File Formats

By default, caches are saved to disk in the pickle format. This is convenient since it can store Python objects. However, you can also pass 'csv' or 'json' for the `file_format` keyword arg to the get_storage call.

2.2.3 Expirations

Caches also offer an optional argument, `TTL`, which is the max lifetime for objects specified in minutes.

```
people = plugin.get_storage('people', TTL=24)
```

2.2.4 Caching Decorator

xbmcswift2 provides a convenient caching decorator to automatically cache the output of a function. For example, suppose we have a function `get_api_data`, that goes out to a remote API and fetches lots of data. If the website only updates the API once a day, it doesn't make sense to make this request every time the addon is run. So we can use the caching decorator with a TTL argument.

```
@plugin.cached(TTL=60*24)
def get_api_data();
    # make remote request
    data = get_remote_data()
    return data
```

The default TTL is 1 day if not provided.

2.2.5 Caching Views

It's also possible to cache views (functions decorated with `plugin.route()`). To simplify addon code, there is a special decorator called `cached_route`. All of the arguments to cached_route are the same as the regular `route` decorator. Currently, it is not possible to specify a TTL for this decorator; it defaults to 24 hours.

```
@plugin.cached_route('/')
def main_menu();
    # do stuff
```

Warning: This is only currently possible for views that return lists of dictionaries. If you call plugin.finish() you *cannot* currently cache the view. See the below section 'Caveats' for more information.

Warning: It is currently only possible to attach a single cached_route to a view. If you have multiple routes on a given view, try refactoring some logic out to a new function that can be cached, instead of using the cached_route decorator.

2.2.6 Caveats

The caching features of xbmcswift2 are still young and thus have some potential problems to be aware of.

- First, if you are calling `plugin.finish` from a view, it is not currently possible to cache the view. This is because there are a few side effects which happen in `finish` which would not be cached. If this is the case, perhaps you can move some functionality in your view into a new function, and cache that result instead.
- Ensure variables are part of your method signature. If you cache a given function, ensure that all possible inputs are in your method signature. xbmcswift2 uses the arguments passed to your function as the unique key for the cache. Therefore it's possible to cache different return values for different inputs for a function. But if you check some global state from inside your function, the caching logic will have no knowlege of this and will return the *wrong* result.
- Currently, caches can grow very large since they do not automatically purge themselves based on filesize. Depending on what you are caching, you might need to introduce some logic to clear the cache.

```
cache = plugin.get_cache('people')
cache.clear()
cache.sync()
```

- It's advisable to include caching as the final step in your development process. If you are still developing your addon, occasionally incorrect return values can be cached which will cause you headaches.

2.3 Patterns

2.3.1 Caching

View Caching

Use the `cached_route()` decorator instead of the normal *route* decorator. This will cache the results of your view for 24 hours.

NOTE: You must be returning a list of plain dictionaries from your view and cannot return plugin.finish(). This is due to a current limitation in the cache which doesn't keep track of side effects such as a call to plugin.finish. If you need to call plugin.finish() because you are passing non-default arguments, then see the next example which uses plugin.cached().

```
@plugin.cached_route('/subjects/', options={'url': full_url('subjects')})
def show_subjects(url):
    '''Lists available subjects found on the website'''
    html = htmlify(url)
    subjects = html.findAll('a', {'class': 'subj-links'})

    items = [{
        'label': subject.div.string.strip(),
```

```
        'path': plugin.url_for('show_topics', url=full_url(subject['href'])),
    } for subject in subjects]
    return items
```

General Function Caching

To cache the results of any function call, simply use the `cached()` decorator. Keep in mind that the function name along with the args and kwargs used to call the function are used as the cache key. If your function depends on any variables in outer scope which could affect the return value, you should pass in those variables explictly as args to ensure a different cache entry is created.

```
@plugin.cached()
def get_api_data():
    return download_data()
```

Storing Arbitrary Objects

You can always create your own persistent storage using `get_storage()`. The returned storage acts like a dictionary, however it is automatically persisted to disk.

```
storage = plugin.get_storage('people')
storage['jon'] = {'vehicle': 'bike'}
storage['dave']       # Throws KeyError
storage.get('dave')   # Returns None
storage.clear()       # Clears all items from the storage
```

2.3.2 Adding pagination

If you are scraping a website that uses pagination, it's possible to present the same interface in XBMC without having to scrape all of the pages up front. To accomplish this, we are going to create our own *Next* and *Previous* list items which go the next and previous page of results respectively. We're also going to take advantage of a parameter option that gets passed to XBMC, *updateListing*. If we pass True for this parameter, then every time the use clicks the Next item, the URL won't be added to history. This enables the ".." list item to go to the correct parent directory, instead of the previous page.

Some example code:

```
@plugin.route('/videos/<page>')
def show_videos(page='1'):
    page = int(page)  # all url params are strings by default
    videos, next_page = get_videos(page)
    items = [make_item(video) for video in videos]

    if next_page:
        items.insert(0, {
            'label': 'Next >>',
            'path': plugin.url_for('show_videos', page=str(page + 1))
        })

    if page > 1:
        items.insert(0, {
            'label': '<< Previous',
            'path': plugin.url_for('show_videos', page=str(page - 1))
        })
```

```
    return plugin.finish(items, update_listing=True)
```

The first thing to notice about our view, is that it takes a page number as a URL parameter. We then pass the page number to the API call, get_videos(), to return the correct data based on the current page. Then we create our own previous/next list items depending on the current page. Lastly, we are returning the result of the call to plugin.finish(). By default, when you normally return a list of dicts, plugin.finish() is called for you. However, in this case we need to pass the update_listing=True parameter so we must call it explictly.

Setting update_listing to True, notifies XBMC that we are paginating, and that every new page should *not* be a new entry in the history.

2.3.3 Reusing views with multiple routes

It is possible to decorate views with more than one route. This becomes useful if you are parsing different URLs that share the same parsing code. In order to unambiguously use `url_for()`, you need to pass a value for the name keyword argument. When calling `url_for`, you pass this specified name instead of the name of the actual function.

If the decorated method requires arguments, it is possible to pass these as default keyword arguments to the `route` decorator. Also, the function itself can use python's default argument syntax.

```
@plugin.route('/movies/', name='show_movie_genres')
@plugin.route('/silents/', name='show_silent_genres', options={'path': 'index.php/silent-films-menu'}
@plugin.route('/serials/', name='show_serials', options={'path': 'index.php/serials'})
def show_genres(path='movies'):
    pass
```

2.3.4 Adding sort methods

Sort methods enable the user to sort a directory listing in different ways. You can see the available sort methods here, or by doing `dir(xbmcswift2.SortMethod)`. The simplest way to add sort methods to your views is to call plugin.finish() with a sort_methods argument and return the result from your view (this is what xbmcswift2 does behind the scenes normally).

```
@plugin.route('/movies')
def show_movies():
    movies = api.get_movies()
    items = [create_item(movie) for movie in movies]
    return plugin.finish(items, sort_methods=['playlist_order', 'title', 'date'])
```

See `xbmcswift2.Plugin.finish()` for more information.

2.3.5 Playing RTMP urls

If we need to play an RTMP url, we can use `xbmcswift.Plugin.play_video()`.

```
@plugin.route('/live/')
def watch_live():
    item = {
        'label': 'AlJazeera Live',
        'path': 'rtmp://aljazeeraflashlivefs.fplive.net:1935/aljazeeraflashlive-live/aljazeera_englis
    }
    return plugin.play_video(item)
```

2.3.6 Using settings

how to use settings

2.3.7 Using the Context Menu

XBMC allows plugin to authors to update the context menu on a per list item basis. This allows you to add more functionality to your addons, as you can allow users other actions for a given item. One popular use for this feature is to create allow playable items to be added to custom playlists within the addon. (See the itunes or reddit-music addons for implementations).

In xbmcswift2, adding context menu items is accomplished by passing a value for the *context_menu* key in an item dict. The value should be a list of 2-tuples. Each tuple corresponds to a context menu item, and should be of the format (display_string, action) where action is a string corresponding to one of XBMC's built-in functions. See XBMC's documentation for more information.

The most common actions are *XBMC.RunPlugin()* and *XBMC.Container.Update()*. RunPlugin takes a single argument, a URL for a plugin (you can create a URL with `xbmcswift2.Plugin.url_for()`). XBMC will then run your plugin in a background thread, *it will not affect the current UI*. So, RunPlugin is good for any sort of background task. Update(), however will change the current UI directory, so is useful when data is updated and you need to refresh the screen.

If you are using one of the two above built-ins, there are convenience functions in xbmcswift2 in the actions module.

Here is a quick example of updating the context menu.

```
from xbmcswift2 import actions

@plugin.url('/favorites/add/<url>')
def add_to_favs(url):
    # this is a background view
    ...

def make_favorite_ctx(url)
    label = 'Add to favorites'
    new_url = plugin.url_for('add_to_favorites', url=url)
    return (label, actions.background(new_url))


@plugin.route('/movies')
def show_movies()
    items = [{
        ...
        'context_menu': [
            make_favorite_ctx(movie['url']),
        ],
        'replace_context_menu': True,
    } for movie in movies]
    return items
```

Sometimes the context_menu value can become very nested, so we've pulled out the logic into the `make_favorite_ctx` function. Notice also the use of the *replace_context_menu* key and the True value. This instructs XBMC to clear the context menu prior to adding your context menu items. By default, your context menu items are mixed in with the built in options.

2.3.8 Using extra parameters in the query string

When calling `xbmcswift.Plugin.url_for()`, any keyword arguments passed that are not required for the specified view function will be added as query string arguments.

A dict of query string parameters can be accessed from `plugin.request.args`.

Any arguments that are not instances of basestring will attempt to be preserved by pickling them before being encoded into the query string. This functionality isn't fully tested however, and XBMC does limit the length of URLs. If you need to preserve python objects between function calls, see the Caching patterns.

2.3.9 Using Modules

Modules are meant to be mini-addons. They have some basic functionality that is separate from the main plugin. In order to be used, they must be registered with a plugin.

Creating an add to favorites plugin:

```
from xbmcswift import Module

playlist = Module(__name__)

@playlist.route('/add/')
def add_to_playlist():
    items = [playlist.qs_args]
        return playlist._plugin.add_to_playlist(items)
```

Examples of plugins

- add to favorites
- report to google form

2.3.10 Testing with Nose

How to test with nose

CHAPTER 3

API Reference

3.1 API

Covers every method of xbmcswift2

3.1.1 Plugin Object

class xbmcswift2.**Plugin**(*name=None*, *addon_id=None*, *filepath=None*, *info_type=None*)

The Plugin objects encapsulates all the properties and methods necessary for running an XBMC plugin. The plugin instance is a central place for registering view functions and keeping track of plugin state.

Usually the plugin instance is created in the main addon.py file for the plugin. Typical creation looks like this:

```
from xbmcswift2 import Plugin
plugin = Plugin('Hello XBMC')
```

Changed in version 0.2: The *addon_id* and *filepath* parameters are now optional. They will now default to the correct values.

Parameters

- **name** – The name of the plugin, e.g. 'Academic Earth'.
- **addon_id** – The XBMC addon ID for the plugin, e.g. 'plugin.video.academicearth'. This parameter is now optional and is really only useful for testing purposes. If it is not provided, the correct value will be parsed from the addon.xml file.
- **filepath** – Optional parameter. If provided, it should be the path to the addon.py file in the root of the addon directoy. This only has an effect when xbmcswift2 is running on the command line. Will default to the current working directory since xbmcswift2 requires execution in the root addon directoy anyway. The parameter still exists to ease testing.

add_items(*items*)

Adds ListItems to the XBMC interface. Each item in the provided list should either be instances of xbmcswift2.ListItem, or regular dictionaries that will be passed to xbmcswift2.ListItem.from_dict. Returns the list of ListItems.

Parameters **items** – An iterable of items where each item is either a dictionary with keys/values suitable for passing to `xbmcswift2.ListItem.from_dict()` or an instance of `xbmcswift2.ListItem`.

add_sort_method(*sort_method*, *label2_mask=None*)
A wrapper for xbmcplugin.addSortMethod(). You can use `dir(xbmcswift2.SortMethod)` to list all available sort methods.

Parameters

- **sort_method** – A valid sort method. You can provided the constant from xbmcplugin, an attribute of SortMethod, or a string name. For instance, the following method calls are all equivalent:
  - `plugin.add_sort_method(xbmcplugin.SORT_METHOD_TITLE)`
  - `plugin.add_sort_metohd(SortMethod.TITLE)`
  - `plugin.add_sort_method('title')`
- **label2_mask** – A mask pattern for label2. See the XBMC documentation for more information.

add_to_playlist(*items*, *playlist='video'*)
Adds the provided list of items to the specified playlist. Available playlists include *video* and *music*.

add_url_rule(*url_rule*, *view_func*, *name*, *options=None*)
This method adds a URL rule for routing purposes. The provided name can be different from the view function name if desired. The provided name is what is used in url_for to build a URL.

The route decorator provides the same functionality.

added_items
The list of currently added items.

Even after repeated calls to `add_items()`, this property will contain the complete list of added items.

addon
This plugin's wrapped instance of xbmcaddon.Addon.

cached(*TTL=1440*)
A decorator that will cache the output of the wrapped function. The key used for the cache is the function name as well as the **args* and ***kwargs* passed to the function.

Parameters **TTL** – time to live in minutes

Note: For route caching, you should use `xbmcswift2.Plugin.cached_route()`.

---

cached_route(*url_rule*, *name=None*, *options=None*, *TTL=None*)
A decorator to add a route to a view and also apply caching. The url_rule, name and options arguments are the same arguments for the route function. The TTL argument if given will passed along to the caching decorator.

clear_function_cache()
Clears the storage that caches results when using `xbmcswift2.Plugin.cached_route()` or `xbmcswift2.Plugin.cached()`.

end_of_directory(*succeeded=True*, *update_listing=False*, *cache_to_disc=True*)
Wrapper for xbmcplugin.endOfDirectory. Records state in self._end_of_directory.

Typically it is not necessary to call this method directly, as calling `finish()` will call this method.

finish(*items=None*, *sort_methods=None*, *succeeded=True*, *update_listing=False*, *cache_to_disc=True*, *view_mode=None*)
Adds the provided items to the XBMC interface.

Parameters

- **items** – an iterable of items where each item is either a dictionary with keys/values suitable for passing to `xbmcswift2.ListItem.from_dict()` or an instance of `xbmcswift2.ListItem`.
- **sort_methods** – a list of valid XBMC sort_methods. Each item in the list can either be a sort method or a tuple of `sort_method, label2_mask`. See `add_sort_method()` for more detail concerning valid sort_methods.

  Example call with sort_methods:

  ```
  sort_methods = ['label', 'title', ('date', '%D')]
  plugin.finish(items, sort_methods=sort_methods)
  ```

- **view_mode** – can either be an integer (or parseable integer string) corresponding to a view_mode or the name of a type of view. Currrently the only view type supported is 'thumbnail'.

Returns a list of all ListItems added to the XBMC interface.

get_setting(*key*, *converter=None*, *choices=None*)

Returns the settings value for the provided key. If converter is str, unicode, bool or int the settings value will be returned converted to the provided type. If choices is an instance of list or tuple its item at position of the settings value be returned. .. note:: It is suggested to always use unicode for text-settings

because else xbmc returns utf-8 encoded strings.

Parameters

- **key** – The id of the setting defined in settings.xml.
- **converter** – (Optional) Choices are str, unicode, bool and int.
- **converter** – (Optional) Choices are instances of list or tuple.

Examples:

- `plugin.get_setting('per_page', int)`
- `plugin.get_setting('password', unicode)`
- `plugin.get_setting('force_viewmode', bool)`
- `plugin.get_setting('content', choices=('videos', 'movies'))`

get_storage(*name='main'*, *file_format='pickle'*, *TTL=None*)

Returns a storage for the given name. The returned storage is a fully functioning python dictionary and is designed to be used that way. It is usually not necessary for the caller to load or save the storage manually. If the storage does not already exist, it will be created.

See also:

`xbmcswift2.TimedStorage` for more details.

Parameters

- **name** – The name of the storage to retrieve.
- **file_format** – Choices are 'pickle', 'csv', and 'json'. Pickle is recommended as it supports python objects.

Note: If a storage already exists for the given name, the file_format parameter is ignored. The format will be determined by the existing storage file.

- **TTL** – The time to live for storage items specified in minutes or None for no expiration. Since storage items aren't expired until a storage is loaded form disk, it is possible to call get_storage() with a different TTL than when the storage was created. The currently specified TTL is always honored.

get_string(*stringid*)
Returns the localized string from strings.xml for the given stringid.

get_view_mode_id(*view_mode*)
Attempts to return a view_mode_id for a given view_mode taking into account the current skin. If not view_mode_id can be found, None is returned. 'thumbnail' is currently the only suppported view_mode.

handle
The current plugin's handle. Equal to `plugin.request.handle`.

id
The id for the addon instance.

keyboard(*default=None*, *heading=None*, *hidden=False*)
Displays the keyboard input window to the user. If the user does not cancel the modal, the value entered by the user will be returned.

Parameters

- **default** – The placeholder text used to prepopulate the input field.
- **heading** – The heading for the window. Defaults to the current addon's name. If you require a blank heading, pass an empty string.
- **hidden** – Whether or not the input field should be masked with stars, e.g. a password field.

list_storages()
Returns a list of existing stores. The returned names can then be used to call get_storage().

log
The log instance for the plugin. Returns an instance of the stdlib's `logging.Logger`. This log will print to STDOUT when running in CLI mode and will forward messages to XBMC's log when running in XBMC. Some examples:

```
plugin.log.debug('Debug message')
plugin.log.warning('Warning message')
plugin.log.error('Error message')
```

name
The addon's name

notify(*msg=''*, *title=None*, *delay=5000*, *image=''*)
Displays a temporary notification message to the user. If title is not provided, the plugin name will be used. To have a blank title, pass '' for the title argument. The delay argument is in milliseconds.

open_settings()
Opens the settings dialog within XBMC

redirect(*url*)
Used when you need to redirect to another view, and you only have the final plugin:// url.

register_module(*module*, *url_prefix*)
Registers a module with a plugin. Requires a url_prefix that will then enable calls to url_for.

Parameters

- **module** – Should be an instance *xbmcswift2.Module*.
- **url_prefix** – A url prefix to use for all module urls, e.g. '/mymodule'

request
The current `Request`.

Raises an Exception if the request hasn't been initialized yet via `run()`.

route(*url_rule*, *name=None*, *options=None*)
A decorator to add a route to a view. name is used to differentiate when there are multiple routes for a given view.

run(*test=False*)
The main entry point for a plugin.

set_content(*content*)
Sets the content type for the plugin.

set_resolved_url(*item=None*, *subtitles=None*)
Takes a url or a listitem to be played. Used in conjunction with a playable list item with a path that calls back into your addon.

Parameters

- **item** – A playable list item or url. Pass None to alert XBMC of a failure to resolve the item.

  Warning: When using set_resolved_url you should ensure the initial playable item (which calls back into your addon) doesn't have a trailing slash in the URL. Otherwise it won't work reliably with XBMC's PlayMedia().

- **subtitles** – A URL to a remote subtitles file or a local filename for a subtitles file to be played along with the item.

set_view_mode(*view_mode_id*)
Calls XBMC's Container.SetViewMode. Requires an integer view_mode_id

storage_path
A full path to the storage folder for this plugin's addon data.

url_for(*endpoint*, ***items*)
Returns a valid XBMC plugin URL for the given endpoint name. endpoint can be the literal name of a function, or it can correspond to the name keyword arguments passed to the route decorator.

Raises AmbiguousUrlException if there is more than one possible view for the given endpoint name.

3.1.2 ListItem

class xbmcswift2.**ListItem**(*label=None*, *label2=None*, *icon=None*, *thumbnail=None*, *path=None*)
A wrapper for the xbmcgui.ListItem class. The class keeps track of any set properties that xbmcgui doesn't expose getters for.

add_context_menu_items(*items*, *replace_items=False*)
Adds context menu items. If replace_items is True all previous context menu items will be removed.

add_stream_info(*stream_type*, *stream_values*)
Adds stream details

as_tuple()
Returns a tuple of list item properties: (path, the wrapped xbmcgui.ListItem, is_folder)

as_xbmc_listitem()
Returns the wrapped xbmcgui.ListItem

classmethod from_dict(*label=None*, *label2=None*, *icon=None*, *thumbnail=None*, *path=None*, *selected=None*, *info=None*, *properties=None*, *context_menu=None*, *replace_context_menu=False*, *is_playable=None*, *info_type='video'*, *stream_info=None*)
A ListItem constructor for setting a lot of properties not available in the regular __init__ method. Useful to collect all the properties in a dict and then use the **dct to call this method.

get_context_menu_items()
Returns the list of currently set context_menu items.

get_icon()
Returns the listitem's icon image

get_is_playable()
Returns True if the listitem is playable, False if it is a directory

get_label()
Sets the listitem's label

get_label2()
Returns the listitem's label2

get_path()
Returns the listitem's path

get_played()
Returns True if the video was played.

get_property(*key*)
Returns the property associated with the given key

get_thumbnail()
Returns the listitem's thumbnail image

icon
Returns the listitem's icon image

is_selected()
Returns True if the listitem is selected.

label
Sets the listitem's label

label2
Returns the listitem's label2

path
Returns the listitem's path

playable
Returns True if the listitem is playable, False if it is a directory

select(*selected_status=True*)
Sets the listitems selected status to the provided value. Defaults to True.

selected
Returns True if the listitem is selected.

set_icon(*icon*)
Sets the listitem's icon image

set_info(*type*, *info_labels*)
Sets the listitems info

set_is_playable(*is_playable*)
Sets the listitem's playable flag

set_label(*label*)
Returns the listitem's label

set_label2(*label*)
Sets the listitem's label2

set_path(*path*)
Sets the listitem's path

set_played(*was_played*)
Sets the played status of the listitem. Used to differentiate between a resolved video versus a playable item. Has no effect on XBMC, it is strictly used for xbmcswift2.

set_property(*key*, *value*)
Sets a property for the given key and value

set_thumbnail(*thumbnail*)
Sets the listitem's thumbnail image

thumbnail
Returns the listitem's thumbnail image

3.1.3 Request

class xbmcswift2.**Request**(*url*, *handle*)
The request objects contains all the arguments passed to the plugin via the command line.

Parameters

- **url** – The complete plugin URL being requested. Since XBMC typically passes the URL query string in a separate argument from the base URL, they must be joined into a single string before being provided.
- **handle** – The handle associated with the current request.

handle = None
The current request's handle, an integer.

url = None
The entire request url.

3.1.4 Actions

xbmcswift2.actions

This module contains wrapper functions for XBMC built-in functions.

copyright

3. 2012 by Jonathan Beluch

license GPLv3, see LICENSE for more details.

xbmcswift2.actions.**background**(*url*)
This action will run an addon in the background for the provided URL.

See 'XBMC.RunPlugin()' at http://wiki.xbmc.org/index.php?title=List_of_built-in_functions.

xbmcswift2.actions.**update_view**(*url*)
This action will update the current container view with provided url.

See 'XBMC.Container.Update()' at http://wiki.xbmc.org/index.php?title=List_of_built-in_functions.

3.1.5 Extended API

Module

class xbmcswift2.**Module**(*namespace*)
Modules are basically mini plugins except they don't have any functionality until they are registered with a Plugin.

add_items(*items*)
Adds ListItems to the XBMC interface. Each item in the provided list should either be instances of xbmcswift2.ListItem, or regular dictionaries that will be passed to xbmcswift2.ListItem.from_dict. Returns the list of ListItems.

Parameters **items** – An iterable of items where each item is either a dictionary with keys/values suitable for passing to `xbmcswift2.ListItem.from_dict()` or an instance of `xbmcswift2.ListItem`.

add_sort_method(*sort_method*, *label2_mask=None*)
A wrapper for xbmcplugin.addSortMethod(). You can use `dir(xbmcswift2.SortMethod)` to list all available sort methods.

Parameters

- **sort_method** – A valid sort method. You can provided the constant from xbmcplugin, an attribute of SortMethod, or a string name. For instance, the following method calls are all equivalent:
  - `plugin.add_sort_method(xbmcplugin.SORT_METHOD_TITLE)`
  - `plugin.add_sort_metohd(SortMethod.TITLE)`
  - `plugin.add_sort_method('title')`
- **label2_mask** – A mask pattern for label2. See the XBMC documentation for more information.

add_to_playlist(*items*, *playlist='video'*)
Adds the provided list of items to the specified playlist. Available playlists include *video* and *music*.

add_url_rule(*url_rule*, *view_func*, *name*, *options=None*)
This method adds a URL rule for routing purposes. The provided name can be different from the view function name if desired. The provided name is what is used in url_for to build a URL.

The route decorator provides the same functionality.

added_items
Returns this module's added_items

addon
Returns the module's addon

cache_path
Returns the module's cache_path.

cached(*TTL=1440*)
A decorator that will cache the output of the wrapped function. The key used for the cache is the function name as well as the **args* and ***kwargs* passed to the function.

Parameters **TTL** – time to live in minutes

Note: For route caching, you should use `xbmcswift2.Plugin.cached_route()`.

---

clear_function_cache()
Clears the storage that caches results when using `xbmcswift2.Plugin.cached_route()` or `xbmcswift2.Plugin.cached()`.

end_of_directory(*succeeded=True*, *update_listing=False*, *cache_to_disc=True*)
Wrapper for xbmcplugin.endOfDirectory. Records state in self._end_of_directory.

Typically it is not necessary to call this method directly, as calling `finish()` will call this method.

finish(*items=None*, *sort_methods=None*, *succeeded=True*, *update_listing=False*, *cache_to_disc=True*, *view_mode=None*)
Adds the provided items to the XBMC interface.

Parameters

- **items** – an iterable of items where each item is either a dictionary with keys/values suitable for passing to `xbmcswift2.ListItem.from_dict()` or an instance of `xbmcswift2.ListItem`.
- **sort_methods** – a list of valid XBMC sort_methods. Each item in the list can either be a sort method or a tuple of `sort_method, label2_mask`. See `add_sort_method()` for more detail concerning valid sort_methods.

  Example call with sort_methods:

  ```
  sort_methods = ['label', 'title', ('date', '%D')]
  plugin.finish(items, sort_methods=sort_methods)
  ```

- **view_mode** – can either be an integer (or parseable integer string) corresponding to a view_mode or the name of a type of view. Currrently the only view type supported is 'thumbnail'.

Returns a list of all ListItems added to the XBMC interface.

get_setting(*key*, *converter=None*, *choices=None*)
Returns the settings value for the provided key. If converter is str, unicode, bool or int the settings value will be returned converted to the provided type. If choices is an instance of list or tuple its item at position of the settings value be returned. .. note:: It is suggested to always use unicode for text-settings

because else xbmc returns utf-8 encoded strings.

Parameters

- **key** – The id of the setting defined in settings.xml.
- **converter** – (Optional) Choices are str, unicode, bool and int.
- **converter** – (Optional) Choices are instances of list or tuple.

Examples:

- `plugin.get_setting('per_page', int)`
- `plugin.get_setting('password', unicode)`
- `plugin.get_setting('force_viewmode', bool)`
- `plugin.get_setting('content', choices=('videos', 'movies'))`

get_storage(*name='main'*, *file_format='pickle'*, *TTL=None*)
Returns a storage for the given name. The returned storage is a fully functioning python dictionary and is designed to be used that way. It is usually not necessary for the caller to load or save the storage manually. If the storage does not already exist, it will be created.

See also:

`xbmcswift2.TimedStorage` for more details.

Parameters

- **name** – The name of the storage to retrieve.
- **file_format** – Choices are 'pickle', 'csv', and 'json'. Pickle is recommended as it supports python objects.

  Note: If a storage already exists for the given name, the file_format parameter is ignored. The format will be determined by the existing storage file.

- **TTL** – The time to live for storage items specified in minutes or None for no expiration. Since storage items aren't expired until a storage is loaded form disk, it is possible to call get_storage() with a different TTL than when the storage was created. The currently specified TTL is always honored.

get_string(*stringid*)
Returns the localized string from strings.xml for the given stringid.

get_view_mode_id(*view_mode*)
Attempts to return a view_mode_id for a given view_mode taking into account the current skin. If not view_mode_id can be found, None is returned. 'thumbnail' is currently the only suppported view_mode.

handle
Returns this module's handle

keyboard(*default=None*, *heading=None*, *hidden=False*)
Displays the keyboard input window to the user. If the user does not cancel the modal, the value entered by the user will be returned.

Parameters

- **default** – The placeholder text used to prepopulate the input field.
- **heading** – The heading for the window. Defaults to the current addon's name. If you require a blank heading, pass an empty string.
- **hidden** – Whether or not the input field should be masked with stars, e.g. a password field.

list_storages()
Returns a list of existing stores. The returned names can then be used to call get_storage().

log
Returns the registered plugin's log.

notify(*msg=''*, *title=None*, *delay=5000*, *image=''*)
Displays a temporary notification message to the user. If title is not provided, the plugin name will be used. To have a blank title, pass '' for the title argument. The delay argument is in milliseconds.

open_settings()
Opens the settings dialog within XBMC

plugin
Returns the plugin this module is registered to, or raises a RuntimeError if not registered.

redirect(*url*)
Used when you need to redirect to another view, and you only have the final plugin:// url.

request
Returns the current request

route(*url_rule*, *name=None*, *options=None*)
A decorator to add a route to a view. name is used to differentiate when there are multiple routes for a given view.

set_content(*content*)
Sets the content type for the plugin.

set_resolved_url(*item=None*, *subtitles=None*)
Takes a url or a listitem to be played. Used in conjunction with a playable list item with a path that calls back into your addon.

Parameters

- **item** – A playable list item or url. Pass None to alert XBMC of a failure to resolve the item.

  Warning: When using set_resolved_url you should ensure the initial playable item (which calls back into your addon) doesn't have a trailing slash in the URL. Otherwise it won't work reliably with XBMC's PlayMedia().

- **subtitles** – A URL to a remote subtitles file or a local filename for a subtitles file to be played along with the item.

set_view_mode(*view_mode_id*)
Calls XBMC's Container.SetViewMode. Requires an integer view_mode_id

url_for(*endpoint*, *explicit=False*, ***items*)
Returns a valid XBMC plugin URL for the given endpoint name. endpoint can be the literal name of a function, or it can correspond to the name keyword arguments passed to the route decorator.

Currently, view names must be unique across all plugins and modules. There are not namespace prefixes for modules.

url_prefix
Sets or gets the url prefix of the module.

Raises an Exception if this module is not registered with a Plugin.

TimedStorage

class xbmcswift2.**TimedStorage**(*filename*, *file_format='pickle'*, *TTL=None*)
A dict with the ability to persist to disk and TTL for items.

close()
Calls sync

dump (*fileobj*)
Handles the writing of the dict to the file object

get (*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

initial_update (*mapping*)
Initially fills the underlying dictionary with keys, values and timestamps.

items () → list of D's (key, value) pairs, as 2-tuples

iteritems () → an iterator over the (key, value) items of D

iterkeys () → an iterator over the keys of D

itervalues () → an iterator over the values of D

keys () → list of D's keys

load (*fileobj*)
Load the dict from the file object

pop (*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem () → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

raw_dict ()
Returns the wrapped dict

setdefault (*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

sync ()
Write the dict to disk

update ([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values () → list of D's values

CHAPTER 4

Other Notes

4.1 Upgrading from xbmcswift

While the API for xbmcswift2 is very similar to xbmcswift, there are a few backwards incompatible changes. The following list highlights the biggest changes:

- Update all imports to use `xbmcswift2` instead of `xbmcswift`. This includes the dependency in your addon.xml file.
- In list item dictionaries, the `url` keyword has been changed to `path`.
- In xbmcswift views, the proper way to return from a view was `return plugin.add_items(items)`. In xbmcswift2 you can either `return plugin.finish(items)` or more simply `return items`.

```
# xbmcswift
return plugin.add_items(items)

# xbmcswift2
return plugin.finish(items)
# (or)
return items
```

- In the past, the `plugin.route()` decorator accepted arbitrary keyword arguments in the call to be used as defaults. These args must now be a single dictionary for the keyword arg `options`.

```
# xbmcswift
plugin.route('/', page='1')

# xbmcswift2
plugin.route('/', options={'page': '1'})
```

- In list item dictionaries, the `is_folder` keyword is no longer necessary. Directory list items are the default and require no special keyword. If you wish to create a playable list item, set the `is_playable` keyword to True.

4.2 Addons Powered by xbmcswift2

Want your addon included here? Send me an email at web@jonathanbeluch.com with your addon name and a link to a repository (XBMC's git repo is fine).

Addon	Source
Academic Earth	https://github.com/jbeluch/xbmc-academic-earth
Documentary.net	https://github.com/jbeluch/plugin.video.documentary.net
VimCasts	https://github.com/jbeluch/xbmc-vimcasts
Radio	https://github.com/dersphere/plugin.audio.radio_de
Shoutcast 2	https://github.com/dersphere/plugin.audio.shoutcast
Cheezburger Network	https://github.com/dersphere/plugin.image.cheezburger_network
Apple Itunes Podcasts	https://github.com/dersphere/plugin.video.itunes_podcasts
MyZen.tv	https://github.com/dersphere/plugin.video.myzen_tv
Rofl.to	https://github.com/dersphere/plugin.video.rofl_to
Wimp	https://github.com/dersphere/plugin.video.wimp

X