Adding Custom Plugins

Qube ArtistView can be customized through plugins. Plugins are python scripts that live in the "plugins" dir next to the qubeArtistView binary (or within the application bundle on OS X). Plugins define both right-click menu items and the right-hand side tabs, e.g. Job Properties and Output log.

- Plugin Paths
- Adding Custom Right Tabs
    - Plugin Layout
    - Types of Tab Plugins (tab_type attribute)
    - Accessing Qube Information from the Plugin
    - Example
- Adding Right-Click Menu Items
    - Plugin Layout
    - Working with Permissions
    - Accessing Qube Information from the Plugin
    - Challenging the User - "Are you sure?"
    - Performing a search from a Plugin
    - Example

Plugin Paths

- You can easily navigate to the stock plugins dir by going to File > Open Plugins Directory
- You can add additional plugins path(s) under the Plugins tab in the preferences
- You can add additional plugins path(s) by setting a comma-separated list of paths in an environment variable called `ARTISTVIEW_ADDITIONAL_PLUGINS_PATHS`

Adding Custom Right Tabs

All of the right-side tabs in ArtistView are driven by Python plugins. Depending on the type of tab displayed, the plugin's job is to retrieve data from the selected items (jobs, instances, frames, or hosts) and present some type of data to be displayed. While ArtistView is written in PyQt, no knowledge of PyQt is needed to create a plugin.

Plugin Layout

- All plugins are of class UserUIPlugin and extend QbUIPlugin.
- In the class's __init__ function, the following are defined:
    - ***Required:***
        - **name**: the display name of the plugin
        - **type**: the type of plugin. Options:
            - **tab**: this plugin creates a new tab
            - **menu**: this plugin creates a right-click menu item
        - **context**: Where the tab will be displayed. Options:
            - **job**: this tab plugin will be shown when the job list is visible.
            - **host**: this plugin will be shown with the host/workers list is visible.
        - **tab_type**: the type of tab plugin. See above "Types of Tab Plugins" for a list of options.
        - **functions**: a dictionary that maps UI focus to a plugin function. As an example, a tab plugin showing logs should show the job log when the job list has focus, an instance log when the instance list has focus, and a frame log when the frame list has focus. More information on the details of the user-defined function can be found in the section below entitled "Accessing Qube Information from the Plugin"
    - ***Optional:***
        - **hidden**: Setting this to true makes the function completely hidden from the UI - it will not even show up in the preferences.
        - **search_field**: Setting this to true displays a search field below the content. That search field will search through displayed content. This only has meaning for "html" plugins.
        - **frame_slider**: Setting this to true displays a frame slider below the content. This only has meaning for "preview" plugins.
        - **aspect_mode**: For the preview tab, this drives how the frame will fit into the window. Options:
            - **-1**: display the original image - do not resize or scale.
            - **0**: ignore aspect ratio - stretch the image to fully fill the display window.
            - **1**: resize to fit within the display window, but maintain aspect ratio.
            - **2**: resize to fit shortest dimension into the display window, and maintain aspect ratio
        - **sort_order**: order in which to display this tab in the list of tabs
- Beyond the __init__, only the functions defined in the **functions** member variable are required. They should return the data specified in the "Types of Plugins" section above.

Types of Tab Plugins (tab_type attribute)

- **html**:
  - Example: The stock "Job Properties" tab
  - Will display basic html but not much in the way of css or javascript.
  - The functional goal of this plugin is to return basic html as a string.
- **webkit**:
  - Example: The stock "Thumbnails" tab.
  - Will display advanced html, including css and javascript - much like a typical web browser.
  - The functional goal of this plugin is to return a complete [x]html page.
- **tree**:
  - Example: QubeTabTreeExample.py in the plugins directory
  - Will display a list of [nested] lists in a tree widget.
  - The functional goal of this plugin is to return a [nested list of] list[s].
- **preview**:
  - Example: The stock "Preview" tab.
  - Will create an image viewer with scrubber for flipping between output frames.
  - The functional goal of this plugin is to return a list of paths to images.
- **openGL**:
  - Deprecated in 6.5-0

Accessing Qube Information from the Plugin

In order for the plugin to do anything meaningful, it must know what entities are selected and have access to the Qube data that drives those entities. Any user function defines must take the kwarg "`**selected`". `Selected` will be a dictionary containing lists of items that are selected in the interface. The dictionary is keyed on the type of item. Regardless of the function, selected will always contain the following keys:

- jobs - a list of currently selected jobs, if any
- subjobs - a list of currently selected subjobs/instances, if any
- frames - a list of currently selected frames/work items/agenda items, if any
- hosts - a list of currently selected hosts, if any
- user_response - True or False answer from challenge given to user (the challenge would come from an "askUser" function).

In the body of your custom function, then, you will retrieve the selected items that matter, based on your function mapping, and do with them what you choose.

Example

This is the QubeTabHelloWorld.py example file you will find in your plugins directory. It unhidden, this plugin would display a tab called "Hello Wold (tab)" that you display basic HTML about the currently selected items. The comments in this example have been changed from those you will find on your system.

The plugin without docstrings and debug logging is quite simple:

```
import sys
import logging
from view.qubeArtistViewUIPlugin import QbUIPlugin
from htmlUtils import makeHtmlTable

class UserUIPlugin(QbUIPlugin):
    def __init__(self):
        super(UserUIPlugin,self).__init__()
        self.name    = "Hello World (tab)" # Display name of the plugin
        self.type    = "tab"               # type of plugin - menu or tab
        self.context = "job"               # context in which the plugin will show
        self.hidden  = True                # Flag to hide this plugin (default is
False)
        self.log_separator_width = 60      # Spacer width in log output
        self.tab_type = "html"             # type of plugin - text == html
        self.search_field = False          # being a text type plugin, setting this to
"True" will give us a search field on the tab
        self.sort_order = 11               # order this tab will show up in the list
of tabs
        self.functions = {'job':self.helloWorldJob,
                          'subjob':self.helloWorldSubjob,
                          'frame':self.helloWorldFrame}

    def helloWorldJob(self,**selected):
        jobs = selected.get('jobs',[])
        if not jobs:
            return ' '
        job_ids = [j['id'] for j in jobs]
        selected_job = jobs[0]
        selections = "Selected job ids are: %s" % ', '.join(map(str,job_ids))
        single_selection = "<p>If we do an operation on one job, it will be job %d" %
jobs[0].get('id')
        return selections + single_selection

    def helloWorldSubjob(self,**selected):
        subjobs = selected.get('subjobs',[])
        if not subjobs:
            return ' '
        subjob_ids = ['%s.%s' % (sj['pid'],sj['id']) for sj in subjobs]
        selected_subjob = subjobs[0]
        selections = "Selected subjob ids are: %s" % ', '.join(map(str,subjob_ids))
        single_selection = "<p>If we do an operation on one subjob, it will be subjob
%d.%d" % (selected_subjob.get('pid'),selected_subjob.get('id'))
        return selections + single_selection

    def helloWorldFrame(self,**selected):
        frames = selected.get('frames',[])
        if not frames:
            return ' '
        frame_ids = ['%s:%s' % (f['pid'],f['id']) for f in frames]
        selected_frame = frames[0]
        selections = "Selected frame ids are: %s" % ', '.join(map(str,frame_ids))
        single_selection = "<p>If we do an operation on one frame, it will be frame
%d:%d" % (selected_frame.get('pid'),selected_frame.get('id'))
        return selections + single_selection
```

With doc strings, you get a little more information:

QubeTabHelloWorld.py

```
import sys
import logging
from view.qubeArtistViewUIPlugin import QbUIPlugin

class UserUIPlugin(QbUIPlugin):
    """
    UserUIPlugin - user created plugins for Qube UI to add functinality
    to the interface.

    A plugin must supply a type, name, and context.


     'name' is the display name of the plugin


     'context' determines under which tabs this [tab] plugin
     will be displayed. Current available contexts are job, or host.


     'type' is the type of plugin - menu or tab


     'hidden' is a flag that, if set to True, will prevent the tab
     from displaying.


     'sort_order' determines the order in which this tab will be displayed.
     It relies on the fact that other [tab] plugins also define a
     number.  If the number isn't defined, it's assigned the default of 999.


     'tab_type' is a flag that determines the type of data displayed
     in the tab widget & hints at the type of data expected.
     Available tab types are:
       html: Will display basic html but not much in the way of css.
       tree: Will display a list of [nested] lists in a tree widget
       preview: Will create an image viewer with scrubber for flipping
        between output frames.
       openGL: Will display non-interactive openGL.


     'search_field' will display a search field at the bottom of the tab
      that will allow the user to search for text in the tab.  This only
      works with html tabs.


     'functions' is a mapping of functions to current context.  For example,
      with a tab that displays logs, one might want to display an entire job
      log when the job list has focus or just the frame log if the frame list
      has focus.  This mapping provides the mechanism for returning differnt
      data depending on context.  Possible function keys are 'job', 'subjob',
      'frame', 'host'.
    The plugin's 'functions' pass, as kwargs, the
     currently selected job(s), subjob(s), frame(s), and/or host(s).
     Each job, subjob, frame, or host, is a qb.Job, qb.Subjob, qb.Work,
     or qb.Host, respectively, and can be operated on as one would with
     the qb API.
    """
    def __init__(self):
        super(UserUIPlugin,self).__init__()# Required to initialize the parent
        self.name    = "Hello World (tab)" # Display name of the plugin
        self.type    = "tab"               # type of plugin - menu or tab
        self.context = "job"               # context in which the plugin will show -
job or host
        self.hidden  = True                # Flag to hide this plugin (default is
False)
        self.tab_type = "html"             # type of plugin - text == html
        self.search_field = False          # being a text type plugin, setting this to
"True" will give us a search field on the tab
        self.sort_order = 11               # order this tab will show up in the list
of tabs


  # This is the function mapping - an essential part of a Tab plugin.  This defines
which functions
  # will be run based on the list which has focus in the interface.
        self.functions = {'job':self.helloWorldJob,
                          'subjob':self.helloWorldSubjob,
                          'frame':self.helloWorldFrame}

    def helloWorldJob(self,**selected):
        """
        This is the function that will be run by the UI (as is defined in the
'functions' member
        variable).  The return value should be what is expected by the type of tab
we're running.
        Return values for types of tabs should be:

         'html':    A single string of html.  Newlines are accepted.  Some css is
accepted.
                    ex: <h1>Top</h1>Pre-formatted text:<pre>  1. thing1\n  2.
thing2</pre>
         'tree':    A list/tuple of list/tuples.  You can nest the lists if you want
to
                    create children.  All items in the list must be strings (no ints)
                    ex: (('a','b','c',('1','2','3',('aa','bb'))),('d','e','f'))
         'preview': A list of image locations.
                    ex: ['/path/to/image.0001.png', '/path/to/image.0002.png', ...]

        The paramter 'selected' will have most, if not all, of the following keys:
        jobs:    A list of the selected jobs in the UI
                 Each list item is a subclass of a qb.Job & therefore has all the
attributes
                 that a qb.Job would have.

        subjobs: A list of the selected subjobs
                 Each list item is a subclass of a qb.Subjob

        frames:  A list of the selected frames
                 Each list item is a subclass of qb.Work
        hosts:   A list of the selected hosts
                 Each list item is a subclass of qb.Host

  This particular function will return basic HTML about the current job selection(s).
        """
        logging.debug("%s running %s." % (self.name,sys._getframe().f_code.co_name)) #
this line is optional - it is for debugging only

        jobs = selected.get('jobs',[])     # This gets you a list of Qube Jobs.   Each
job is the same as would be returned by qb.jobinfo()
        if not jobs:
            return ' '
        job_ids = [j['id'] for j in jobs]
        selected_job = jobs[0]
        selections = "Selected job ids are: %s" % ', '.join(map(str,job_ids))
        single_selection = "<p>If we do an operation on one job, it will be job %d" %
jobs[0].get('id')
        return selections + single_selection

    def helloWorldSubjob(self,**selected):
        """
        This function will be used when the subjob/instance list has focus.  It
returns
  basic HTML about the current selection(s)
        """
        logging.debug("%s running %s." % (self.name,sys._getframe().f_code.co_name))

        subjobs = selected.get('subjobs',[])
        if not subjobs:
            return ' '
        subjob_ids = ['%s.%s' % (sj['pid'],sj['id']) for sj in subjobs]
        selected_subjob = subjobs[0]
        selections = "Selected subjob ids are: %s" % ', '.join(map(str,subjob_ids))
        single_selection = "<p>If we do an operation on one subjob, it will be subjob
%d:%d" % (selected_subjob.get('pid'),selected_subjob.get('id'))
        return selections + single_selection

    def helloWorldFrame(self,**selected):
        """
        This function will be used when the subjob/instance list has focus.  It
returns
  basic HTML about the current selection(s)
        """
        logging.debug("%s running %s." % (self.name,sys._getframe().f_code.co_name))

        frames = selected.get('frames',[])
        if not frames:
            return ' '
        frame_ids = ['%s:%s' % (f['pid'],f['id']) for f in frames]
        selected_frame = frames[0]
        selections = "Selected frame ids are: %s" % ', '.join(map(str,frame_ids))
        single_selection = "<p>If we do an operation on one frame, it will be frame
%d:%d" % (selected_frame.get('pid'),selected_frame.get('id'))
        return selections + single_selection
```

Adding Right-Click Menu Items

All of the right-click menu items in ArtistView are driven by Python plugins. These plugins have access to the data behind the selected entities and can do with that data what they choose. When the plugin operation is complete, the interface will update. While ArtistView is written in PyQt, no knowledge of PyQt is needed to create a plugin.

Plugin Layout

- All plugins are of class UserUIPlugin and extend QbUIPlugin.
- All menu plugins must define `__init__` and `run` methods.
- In the class's __init__ function, the following are defined:
    - ***Required:***
        - **name**: the display name of the plugin
        - **type**: the type of plugin. Options:
            - **tab**: this plugin creates a new tab
            - **menu**: this plugin creates a right-click menu item
        - **context**: Where the menu will be displayed - if you need the same menu in multiple places, you will need to create multiple plugins. Options:
            - **job**: this menu plugin will be a part of the right-click menu list when clicking on a job.
            - **subjob**: this menu plugin will be a part of the right-click menu list when clicking on a subjob/instance.
            - **frame**: this menu plugin will be a part of the right-click menu list when clicking on a frame/agenda item/work item.
            - **host**: this menu plugin will be a part of the right-click menu list when clicking on a job.
    - ***Optional:***
        - **hidden**: Setting this to true makes the function completely hidden from the UI - it will not even show up in the preferences.
        - **permission**: Permission required to use this plugin (see "Working with Permissions" section above")
        - **sort_order**: order in which to display this tab in the list of tabs
- The class's run method does all the work.
    - There is no return value.
    - Run is passed a **selected argument that contains data from the interface. See "Access Qube Information from the Plugin"
    - There is no limit to what the run method can do - it doesn't have to do anything Qube related. It could run a qb API call as well as it could make a call to subprocess.Popen, interact with your production tracking system, etc.

Working with Permissions

Qube has a set of internal permissions viewable under the WV User Permissions in WranglerView. Those permissions determine which users can perform which actions and are ultimately controlled by the supervisor. ArtistView can reflect those permissions in its interface by assigning a `permission` member variable to the permission required to perform the actions performed by the plugin. For example, in order to block a job, one must have the "block" permission. In order the modify a job, one must have the "modify" permission.

These permissions can also be used as an obfuscation layer. For example, requiring an "admin" permission would block the plugin from being performed by anyone who is not a Qube admin. This plugin could perform non-Qube-specific actions, but would only be available to Qube administrators.

If a user does not have required permission to run the plugin, then the plugin will not show at all. If the user has required permission, but is not an admin, then the plugin will be visible but disabled when the user attempts to perform the action on any entity they do not own (for example, one cannot modify someone else's job unless they are an admin).

Accessing Qube Information from the Plugin

In order for the plugin to do anything meaningful, it must know what entities are selected and have access to the Qube data that drives those entities. Any user function defines must take the kwarg "`**selected`". `Selected` will be a dictionary containing lists of items that are selected in the interface. The dictionary is keyed on the type of item. Regardless of the function, selected will always contain the following keys:

- jobs - a list of currently selected jobs, if any
- subjobs - a list of currently selected subjobs/instances, if any
- frames - a list of currently selected frames/work items/agenda items, if any
- hosts - a list of currently selected hosts, if any

In the body of your run method, then, you will retrieve the selected items that matter, and do with them what you choose.

When the plugin completes, it signals the interface to update the related jobs or hosts.

Challenging the User - "Are you sure?"

If you would like to display a dialog that the user must agree to before the plugin is to be run, you do so by adding an `askUser` method to the plugin class. This function should return a dictionary that drives the interface. That dictionary must contain the following key/value pairs:

- "title": The window title of the challenge dialog, i.e. "Performing an action"
- "text": The main text to be displayed, i.e. "Are you sure?"
- "info_text": The small-print text displayed below the main text, i.e. "You're about to do something important. Be sure you know what you're doing."
- "detailed_text": If this exists, a "More info" button will be displayed that, when pressed, will expand the dialog, display a scroll bar & allow much more text to be displayed, i.e. "This is what you're about to do to these jobs: ...."
- "icon": The type of icon to display. Options are (the strings):
    - 'information' - a white speech box
    - 'question' - a yellow question mark
    - 'warning' - a yellow exclamation point
    - 'critical' - a red stop sign
    - 'noicon' - nothing
- "button": A list of button(s) text to be displayed. All positive answers eventually return True, all negatives return False. Options are (the strings):
    - 'ok'
    - 'cancel'
    - 'save'
    - 'discard'
    - 'yes'
    - 'no'
    - 'apply'
    - 'abort'
    - 'close'

Performing a search from a Plugin

If you would like to set a search field and perform a search, programmatically, from a plugin, you do so by adding an `updateUI` method to the plugin class. This function should return a dictionary of UI fields that need updating. For 6.7, the only UI fields that can be updated are the search fields which are denoted by the keys `search_jobs`, `search_frames`, `search_instances`, or `search_workers` for the search field in the job list, frame, list, instance list, or worker list, respectively.

For example, to perform a search for all jobs with a matching pgrp to the selected job(s), the updateUI function would look like:

```
def updateUI(self,**selected):
        jobs = selected.get('jobs',[])
        pgrp_id_searches = set(("pgrp:%d"%j['pgrp'] for j in jobs))
        return {"search_jobs":" OR ".join(map(str,list(pgrp_id_searches)))}
```

..which would return a dictionary that looks like `{"search_jobs":"pgrp:1234 OR pgrp:1235"}`. This would then fill the job list search field with `pgrp:1234 OR pgrp:1235` and perform the search.

Searches are performed after the plugin's run method is scheduled to run, however, because the run method runs in its own thread, there is no guarantee that the run method will have completed before the search is performed.

Example

This is the QubeMenuJobHelloWorld.py plugin from the plugins directory that is shipped with ArtistView. If unhidden, this plugin will challenge the user if they really wish to run the plugin. Their choice will logged to stderr in the run method.

The plugin without doc strings or the user challenge info is quite simple:

```
from view.qubeArtistViewUIPlugin import QbUIPlugin
import qb
import logging
class UserUIPlugin(QbUIPlugin):
    def __init__(self):
        super(UserUIPlugin,self).__init__()
        self.name    = "Hello World (job)"  # Display name of the plugin
        self.type    = "menu"               # type of plugin - menu or tab
        self.context = "job"                # context in which the plugin will show
        self.permission = None              # Qube permission required to perform this
task (optional)
        self.hidden  = True                 # Flag to hide this plugin (default is
False)
        self.sort_order = 15                # order this will show up in the menu



    def run(self,**selected):
        job_ids = [j.get('id') for j in selected.get('jobs',[])]
        # Confirm with user:
        if not selected.get('user_response',True):
            logging.info("Skipping job hello world because of user response")
            return
        logging.info("Hello world plugin has completed complete for jobs %s" % ',
'.join(map(str,job_ids)))
```

The same plugin with doc strings is a little more descriptive:

QubeMenuJobHelloWorld.py

```
from view.qubeArtistViewUIPlugin import QbUIPlugin
import qb
import logging
class UserUIPlugin(QbUIPlugin):
    """
    UserUIPlugin - user created plugins for Qube UI to add functinality
    to the interface.
    A plugin must supply a type, name, context.  A menu plugin must supply
    a 'run' function.
     'name' is the display name of the plugin
     'context' determines in which menus/areas this [menu] plugin
     will be displayed. Current available contexts are job, subjob,
     frame, host.
     'type' is the type of plugin - menu or tab
     'permission' is the *Qube* permission required to run this plugin
     'hidden' is a flag that, if set to True, will prevent the menu
     item from displaying.
     'sort_order' determines the order in which this item will show
     in the menu. It relies on the fact that other [menu] plugins also define a
     number.  If the number isn't defined, it's assigned the default of 999.
    The plugin's run function passes, as kwargs, the
     currently selected job(s), subjob(s), frame(s), host(s), and/or user
     response if they were challenged by an askUser function definition.
     Each job, subjob, frame, or host, is a qb.Job, qb.Subjob, qb.Work,
     or qb.Host, respectively, and can be operated on as one would with
     the qb API.
    """
    def __init__(self):
        super(UserUIPlugin,self).__init__()
        self.name    = "Hello World (job)"  # Display name of the plugin
        self.type    = "menu"               # type of plugin - menu or tab
        self.context = "job"                # context in which the plugin will show
        self.permission = None              # Qube permission required to perform this
task (optional)
        self.hidden  = True                 # Flag to hide this plugin (default is
False)
        self.sort_order = 15                # order this will show up in the menu

    def askUser(self, **selected):
        """
        This function is designed to provide a method to allow a dialog to be
presented
        to the user prior to execution.  The result will be available to the run
function
        as a True or False value attached to user_response in the 'selected' kwargs
param
        of the run function, e.g. selected['user_response']
        Return dict should have at least one of the following keys:
         text (reqired): Main text of the dialog.
                         ex: 'You are about to do something...'
         title:          Text displayed in the title bar of the dialog.
                         ex: 'Doing Something.'

         info_text:      Smaller text under the main.
                         ex: 'Do you really want to do this thing to these things?'

         detailed_text:  [Long form] Details about what is going to happen - hidden by
default.
                         ex: 'This thing you are about to do is going to do this, and
this,
                         and that; and is irreversible. Potentially effected jobs
are:....'

         icon:           Type of icon to show on the dialog.  Options are the strings:
                         'information', 'question', 'warning', 'critical', 'noicon'
                         ex: 'information'
         button:         List/tuple of button(s) to display.  Options are the strings:
                         'ok', 'cancel'; 'save', 'discard'; 'yes', 'no'; 'abort',
'close'.
                         Note: All positive options make the dialog return True,
                         all negative options return False.
                         True: ok, yes, save, apply.
                         ex: ('yes','no')
        """
        job_ids = [j.get('id') for j in selected.get('jobs',[])]
        jobs_str = ', '.join(map(str,job_ids))
        return {"text":"Hello world 'text' for job(s) %s" % jobs_str,
                "info_text":"Hello world 'info_text' for job(s)",
                "detailed_text":"Hello world 'detailed_text' for jobs: %s" % jobs_str,
                "title":"Hello world 'title'",
                "icon":"information",
                "buttons": ("Ok","Cancel")}
    def run(self,**selected):
        """
        This is the function that will be run by the UI.  There is no return value.
        When the run function completes, a signal is emitted to the UI to update
        the job(s) that were effected.  This signal will be sent regardless of whether
        or not the user responded positively to the challenge.
        The parameter 'selected' will have most, if not all, of the following keys:
        jobs:    A list of the selected jobs in the UI
                 Each list item is a subclass of a qb.Job & therefore has all the
attributes
                 that a qb.Job would have.

        subjobs: A list of the selected subjobs
                 Each list item is a subclass of a qb.Subjob

        frames:  A list of the selected frames
                 Each list item is a subclass of qb.Work

        hosts:   A list of the selected hosts
                 Each list item is a subclass of qb.Host
        user_response: A True/False response from the user, if the user was challenged
                       (which happens when this plugin defines an
'askUser(**selected)' method.
                         Note: All positive options make the dialog return True,
                         all negative options return False.
                         True: ok, yes, save, apply.
                         ex: ('yes','no')
        """
        job_ids = [j.get('id') for j in selected.get('jobs',[])]
        # Confirm with user:
        if not selected.get('user_response',True):
            logging.info("Skipping job hello world because of user response")
            return
        logging.info("Hello world plugin has completed complete for jobs %s" % ',
'.join(map(str,job_ids)))
```