Custom Menu Items

The context menus of the Dashboard can be extended with custom items to invoke site-specified scripts. To extend a context menu, an entry must be made in a JSON configuration file named menus.config in the site's configuration directory.

Example: Triggering A Request For Help Email

The following example demonstrates how a request for help email could be made through a custom menu item.

Configuration Entry

```
{
  "job":
  [
    {
      "comment": "artists use this to request technical assistance for selected jobs",
      "title": "ask for help",
      "crews": ["lighting", "animation"],
      "suppress": true,
      "exec": ["menu-scripts/askForHelp"],
      "values": ["jid", "owner"],
      "enabled": true
    }
  ]
}
```

In this example, the menu item will appear in the job list's context menu because it is defined in the list keyed by job in the top-level dictionary. task and blade are the other possible menus under which custom menu items may appear.

The **comment** attribute documents the purpose of the menu item. It currently does not play a functional role in the Dashboard.

The **title** attribute defines the name of the menu item as it appears in the menu. In this example, the job list's context menu will show ask for help.

The **crews** attribute restricts the running of the script to users of the specified crews. If no crews are listed, any user can invoke the menu item. In this example, users in the lighting and animation crews are permitted to invoke the menu item.

The **suppress** attribute indicates whether the Dashboard will display the output of the script in a new window. In this case, there is no helpful output, so setting this to false helps to avoid window bloat. However, some scripts such as a report generator or image previewer could process the selected items and return them in a textual or graphical response. If there is any script output, the suppress setting is ignored; this is useful for catching exceptions and displaying the traceback as in the example at the end of this page.

The **exec** attribute specifies the vector of arguments that will be executed by the engine. If the first arguement starts with /, it is considered an absolute path; otherwise, it is considered path relative to the site's configuration directory. In this example, the engine will run a script called askForHelp in the menu-scripts subdirectory the site's configuration directory.

The **values** attribute indicates which attributes of the selected items will be sent to stdin of the executed script. In this example, the jidand owner of the selected jobs will be sent to stdin of askForHelp; the script could use those values to create a message that describes which user(s) is/are requesting help and which specific jobs they need help with. A reference of all attributes can be found here. Job attributes such as jid, priority, and owner may be listed for task menu items.

The **enabled** attributed indicates whether the menu item will be displayed. This is useful for temporarily hiding menu items, or leaving examples in the config file for future use. By default, all menu items are enabled, so it is only required when hiding is desired.

JSON Payload

The JSON payload sent to the stdin of the script will be a list of dictionaries, each dictionary representing one selected item. In the above example, values is set to ["jid", "owner"], and the JSON payload sent to askForHelp for three selected jobs would look something like this:

```
[
  {"jid": 1056, "owner": "harry"},
  {"jid": 1058, "owner": "sally"},
  {"jid": 1042, "owner": "sally"}
]
```

Script

Here is an example implementation of askForHelp:

```
#!/usr/bin/python

import sys, json, smtplib
from email.mime.text import MIMEText

def askForHelp(jobs):
    # email wranglers for help with selected jobs

    # the list of job ids is used informatively in the title and body
    jidsStr = " ".join([str(job["jid"]) for job in jobs])

    # Create a text/plain message
    text = "Please inspect the following jobs:\n%s" % jidsStr
    msg = MIMEText(text)

    msg['Subject'] = "Help Request for %s" % jidsStr

    # site would customize senders and recipients
    # sender could be set to login username of dashboard through env var
    sender = "tractor@example.com"
    # establish list of distinct job owners
    owners = set([job["owner"] + "@example.com" for job in jobs])
    recip = ["wranglers@example.com"] + list(owners)

    msg['From'] = sender
    msg['To'] = ",".join(recip)

    s = smtplib.SMTP('localhost')
    s.sendmail(sender, recip, msg.as_string())
    s.quit()

if __name__=="__main__":
    # determine selected jobs
    jsonData = sys.stdin.read()
    jobs = json.loads(jsonData)
    askForHelp(jobs)

    # in this case, no response is expected because the script was
    # configured with suppress set to true.
    # but if one was expected, we could emit "Message sent."
    # as follows:
    #
    # sys.stdout.write("Content-type:text/html\r\n\r\n")
    # sys.stdout.write("<html><body>")
    # sys.stdout.write("Message sent.")
    # sys.stdout.write("</body></html>")
```

Example: Image Output

The following example demonstrates how an image can be returned and displayed in a new window in the browser.

Configuration Entry

```
{
  ...
  "task":
  [
    ...
    {
      "comment": "display an image for the selected task",
      "title": "show image",
      "exec": ["menu-scripts/showImage"],
      "values": ["jid", "tid"]
    }
    ...
  ]
  ...
}
```

suppress is False by default, so it can be left out of the menu definition if desired.

JSON Payload

The JSON payload for a single selected task would look like this:

```
[
  {"jid": 1098, "tid": 5}
]
```

Script

By having the script set Content-type appropriately in the response header, the browser can then now how to handle and display the returned image data.

For this example to work, you would need to have an image file called eye.png in the engine's /tmp directory that is readable by the engine's process owner.

```
#!/usr/bin/python

import sys, json

if __name__=="__main__":
    # determine selected tasks
    jsonData = sys.stdin.read()
    if jsonData:
        tasks = json.loads(jsonData)

    # something could be done to determine which image to return
    # for now just return a test image in /tmp of the engine host
    filename = "/tmp/eye.png"

    # return the image; it will be displayed in a new browser window
    with open(filename, "rb") as content_file:
        content = content_file.read()
    sys.stdout.write("Content-type: image/png\n\n")
    sys.stdout.write(content)
```

Troubleshooting

When setting up or editing custom menu items, the dashboard will need to be reloaded to reflect the changes.

Missing Menu Item

If you are unable to see a custom menu item, make sure that you belong to the menu item's crews if any are listed. Custom menu items are displayed only to users who are authorized to invoke them.

Another cause of missing custom menu items is a JSON syntax error in the config file. This can be detected by inspecting the browser's console for a related error message, or by running the menus.config file through a JSON validator such as http://jsonlint.com.

Testing From the Command Line

If a menu item appears but fails to run as expected, try some of the following steps.

Check that the script is readable and executable by the engine owner.

Check that the menu-scripts directory is readable and executable by the engine owner.

Try running the script from the command line as the engine owner, piping a test JSON string if it helps cover code paths in question. The following example simulates the command run when two blades named hostA and hostB have been selected, and the menu item's values list indicates each blade's name should be sent.

```
% echo '[{"name": "hostA"}, {"name": "hostB"}]' | ./testScript
```

Editing in Windows

If the script is edited on Windows or in an editor in Windows mode, a trailing ^M character could be left after the interpreter name, causing the script to fail. The ^M is invisible in certain editors or editing modes. Neither the engine log nor the output window will display this error, making it difficult to detect.

Testing the script from the command line will reveal this error. Such an error in a python script would appear as:

```
/usr/bin/python^M: bad interpreter: No such file or directory
```

Catch Exceptions

One way to debug python scripts is to catch exceptions and have them sent as a response to the browser. This is helpful since stderr is not sent to the brower.

The following python example shows how any exception can be caught and provide useful feedback to a user or script writer.

```
#!/usr/bin/python

import sys
import json

def testScript(bladeinfo):
    # intentional error: bar is not defined
    foo = bar

if __name__ == "__main__":
    try:
        jsonData = sys.stdin.read()
        bladeinfo = json.loads(jsonData)
        testScript(bladeinfo)
    except:
        import traceback
        sys.stdout.write("Content-type:text/html\r\n\r\n")
        sys.stdout.write("<html><body><pre>")
        sys.stdout.write("menu script error:\n")
        sys.stdout.write(traceback.format_exc())
        sys.stdout.write("<pre></body></html>")
```

An edge case is that currently it is not possible to catch syntax errors, as the script will fail before getting to the main block. Testing the script from the command line will reveal such syntax errors.