Creating a custom Environment Handler

To follow along, the complete code listing for the Project X Handler is found here.

Let's walk through each of the sections of code to see what this handler does.

- First, let's look at the handler initialization:

```
from TrEnvHandler import TrEnvHandler
import logging

class projxhandler(TrEnvHandler):
    def __init__(self, name, envkeydict, envkeys):
        self.logger = logging.getLogger("tractor-blade")
        self.logger.info("initializing projxhandler: %s" % (name))
        TrEnvHandler.__init__(self, name, envkeydict, envkeys)
```

  The iniitalization is very straight forward. The TrEnvHandler (base class) is imported first, along with the logging module.

  The __init__ routine sets up the logger, then calls the superclass initialization.
- The main processing in an enviroment handler comes in two methods.

  The updateEnvironment method():

```
def updateEnvironment(self, cmd, env, envkeys):
    self.logger.debug("projxhandler.updateEnvironment: %s" % repr(envkeys))
    self.initLocalVars()
    if envkeys and type(envkeys) == type([]):
        for envkey in envkeys:
            key,val = envkey.split("=")
            self.logger.debug("projxhandler.envkey: %s" % envkey)
            if key == "SCENE":
                self.scene = envkey
                self.environmentdict["SCENE"] = self.scene
            elif key == "SHOT":
                self.shot = envkey
                self.environmentdict["SHOT"] = self.shot

    return TrEnvHandler.updateEnvironment(self, cmd, env, envkeys)

def initLocalVars(self):
    self.scene = None
    self.shot = None
```

  The updateEnvironment method is called one time per handler, with a list of envkeys from the pending command, which match the envkeys processed by the handler. In our example, as seen in the dictionary above, this handler is called when the envkey key contains either "ProjectX", "SHOT=somevalue" or "SCENE="somevalue".

  It's assumed that the command envkey definition would generally look something like:

```
RemoteCommand{ ..... } -envkey { ProjectX SHOT=27234-A SCENE=5125 }
```

  The primary responsibility of the updateEnvironment() method is to define values which are placed in the environment of the handler itself. It's final call is generally to the base class method to updateEnvironment(), which merges the handlers environment with that of the launching command as provided by the blade.

  The code in this handler loops through all of the envkeys assocaited with the launching command. It then sets some internal variables, which can be used later, and it sets up several environment variables in the handlers local environment dictionary.

  In the environment dictionary above for the Project X handler, you'll find the following:

```
"environment": {
  "CURRENT_SHOW": "Project X, Venture of Mystery",
  "CURRENT_SHOT": "$SHOT",
  "CURRENT_SCENE": "$SCENE",
  "RMANTREE": "/opt/pixar/RenderManProServer-15.2",
  "PATH": "$RMANTRE:/bin;$PATH"
},
```

  The CURRENT_SHOT and CURRENT_SCENE environment variables are based upon variables that are expected to be in the dictionary when it is processed. For that reason, in the handler code you see that self.environmentdict["SCENE"] and self.environmentdict["SHOT"] are defined in the handlers updateEnvironment() method. When the base class updateEnvironment() method is called, this dictionary is merged into the environment of the launching command, such that CURRENT_SHOT and CURRENT_SCENE are correctly defined.

  We also see an example of internal flattening of the environment where RMANTREE is defined to point to a specific RPS version for this show, and $RMANTREE/bin is loaded into the start of the PATH environment variable.
- After the updateEnvironment() method has been called, the blade then calls the handlers remapCmdArgs() method. The purpose of this method is to allow the handler to rewrite the original command, before it is launched.

  The rewrite method looks like this:

```
    def remapCmdArgs(self, cmdinfo, launchenv, thisHost):
        self.logger.debug("projxhandler.remapCmdArgs: %s" % self.name)
        argv = TrEnvHandler.remapCmdArgs(self, cmdinfo, launchenv, thisHost)
        self.logger.info("scene: %s, shot:%s" %
            (self.scene, self.shot))

        # indicate command was launched by traactor
        launchenv["TRACTOR"] = "1"

        if argv[0] == "render" and "RMANTREE" in launchenv:
            argv[0] = os.path.join(launchenv["RMANTREE"],"bin","prman"))
            argv.[1:1] = ["-statsfile", "%s-%s" % (self.scene, self.shot))

        # on windows for add the Visual Studio default libs and includes
        p = platform.platform()
        if p.find("Windows") != -1:
            if launchenv.has_key("INCLUDE"):
                launchenv["INCLUDE"] += ";" + launchenv["VCINCLUDE"]
            else:
                launchenv["INCLUDE"] = launchenv["VCINCLUDE"]

            if launchenv.has_key("LIB"):
                launchenv["LIB"] += ";" + launchenv["VCLIB"]
            else:
                launchenv["LIB"] = launchenv["VCLIB"]

        return argv
```

The first thing that handlers will generally do is call the base class remapCmdArgs() method. The base class handles general remapping that is used by all of our software. It handles the standard remapping that the alfred scripting language allows, like "%h, %H. It also looks for any command line with a $val in it. The $val is then expected to be supplied from the launch environment. For example *$render* might be remapped to *prman*.

After calling the base class, the handler can provide any custom remapping it requires. In our example handler, it does the following:

- It sets the environment variable TRACTOR to 1 to let downstream scripts know this is a tractor launched job.
- It looks to see if the application to launch is *render* and whether or not RMANTREE is in the environment. If so, it changes the launch command to (effectively) *$RMANTREE/bin/prman*, and it augments the command by adding an output stats file, based upon the current scene and shot values.
- Then this handler checks the platform of the current blade. If it is determined to be a windows platform, it adds the Visual Studio INCLUDE and LIBS values into the launch environment.

Depending upon the number of envkeys which are supplied for a command, more than one environment handler will process the command. The *d efault* handler is always called first, and then any other handlers will be cascaded. When the last handler has processed the environment and launch command, the blade then launches the modified command.