Scripting Best Practices for Performance

Scripting allows users to extend maximo business logic using Python/JS or for that matter any other JSR 223 compliant scripting language. All the script code gets compiled to Java bytecode and are cached as part of Maximo runtime caches. So when the script is invoked – it's the cached bytecode that is executed by the JVM using the JSR 223 bridge. Since the scripting code executes in the same thread as other Maximo business logic (written in Java), a poorly written script code can impact the performance of the system negatively. We have listed below a few common mistakes that we have seen. In general we need to follow the Maximo Performance guidelines as scripting in the end is equivalent to Maximo custom code.

Choosing the right launch point and event

Launchpoints are script trigger points. Often choosing the right launch point can help avoid certain performance issues in scripting. For example, in Maximo 75 release of scripting, there was no support for attribute value initialization. This led many script developers to use the Object Launch point (OLP) Init event to initialize the Mbo attribute values. Though functionally there was not problem with that approach – it potentially can lead to performance issues when selecting a bunch of Mbo's (in list tab or APIs/MIF, Escalations). The OLP init event script gets executed for every mbo that is selected in the MboSet – even though the attribute whose value was getting initialized by the script was not even used/shown. This can be avoided by changing the Object launch point to attribute launch point – Initialize Value event. A sample script code for that is shown below (thisvalue is the current attribute init value)

```
If priority is not None:
    thisvalue=2*priority
```

The Mbo framework will invoke this script only when this attribute is referred to by the code or the UI.

Another example of Launch point choice comes up in the Integration skipping events use case. Often, we would use the user exit scripting to determine if we need to skip an outbound integration message. However, at this point the system has already entailed the cost of serializing the Mbo's. Instead we should use the Publish Channel Event Filter scripting which gets invoked right when the event is triggered and way before any serialization of Mbos happen. A sample script below shows the Event Filter scripting which works with the Mbo's.

```
If service.getMbo().getString("status")=="APPR":
    evalresult=False
evalresult=True
```

Avoid costly Object init events if invoked from list tab

Often you may want to do costly Object init scripts only when the object is initialized from the main tab (UI) and not from the list tab. This is because in such cases the sample code below helps.

```
from psdi.common.context import UIContext
if UIContext.getCurrentContext() is not None and
UIContext.isFromListTab()==False:
    …..costly initlization….
```

Watch out for conflicting launch point event scripts

Scripting framework would allow attaching multiple scripts to the same launch point event. This poses a problem if the script code expects to execute in certain order after or before certain other script in the same launch point event. Since the Maximo event topic is an un-ordered map, the events get fired without a fixed order. This can potentially cause issues if the order dependency is not managed properly. One should evaluate the reason to attach multiple scripts for the same launch point event – and evaluate if it makes more sense to combine them into one script. The other option is to make sure there is no dependency between the scripts.

Avoid calling save in middle of a transaction

This is a common coding pattern we see in scripts that can cause problems to Maximo transactions and event firing.

Ideally when a maximo transaction is in progress, the script should try to be part of that encompassing transaction. The mbos created/updated by a script are automatically part of the encompassing transaction as long as those were created from the script launch point mbo/related mbo. If we create a Mbo using the MXServer.getMXServer().getMboSet("…") api would be outside the encompassing transaction uness they are added explicitly to the encompassing transaction like below

```
mbo.getMXTransaction().add(<newly created a mboset>)
```

Calling MboSet.count() many times

We see a common programing mistake in the scripts where we are checking the count of a MboSet multiple times. Note that the count() call ends up firing a sql every time its called. So an optimal approach would be to invoke it once and store the value in a var and reusing that var for subsequent code flow. An example is shown below

Good code:

```
cnt = mboset.count()

if cnt<=1:

    service.log("skipping this as count is "+cnt)
```

Bad code:

```
If mboset.count()<=1:

    service.log("skipping this as count is "+mboset.count())
```

Closing the MboSet

Maximo Mbo framework would always release the MboSets created after a transaction is complete. That is true as long as all the MboSet's were created as a related set to launch point mbo or any related Mbo to the launch point mbo. If however the MboSet is created using the MXServer.getMXServer().getMboSet(..) api, the script code is responsible for closing and clearing that MboSet up. We suggest a try finally block to do that (a sample shown below)

```
try:

     ….

finally:

     mboset.cleanup()
```

If this is not done, it tends to start building up and may result in OOM errors.

Check if logging in enabled before logging

We often see logging done inside the script without checking the log level. A sample below shows how that can impact performance

```
service.log(“count of mbos “+mboset.count())
```

Now this unfortunately would result in mboset.count() getting called – even though the script logging is diabled.

```
from psdi.util.logging import MXLoggerFactory

logger = MXLoggerFactory.getLogger("maximo.script”);

debugEnabled = logger.isDebugEnabled()


if debugEnabled:
    service.log(“count of mbos “+mboset.count())
```

Starting 7612, we will add a function in the “service” variable that will allow one to check this easily like below

```
If service.isLoggingEnabled():
     service.log(“count of mbos “+mboset.count())
```