User Interaction from bash Scripts

As MacAdmins our goal is to automate workflows when ever possible. The advantages of automation are great. You immediately reduce the workload, but also reduce the potential for someone to make a mistake, which would mean even more work later.

In nearly all cases, we want the automation to happen “magically” in the background, without any user interaction. User interaction slows down the process as the script is waiting for the user to confirm or enter data. User input also requires validating and checking user entered data.

In most cases, when you believe you need to prompt the user for, well, *anything*, you should take that moment to re-think your workflow. Maybe you can come up with a different workflow that can provide the data from a source that can be automated without user interaction.

If, however, you *really* are convinced that user interaction is necessary, then you need be aware there are many potential pitfalls in writing shell scripts with user interaction.

macOS Mojave will introduce even more pitfalls with its increased security features.

Do you really need UI?

A common example for user interaction is to get and set the Computer Name or Asset Tag/ID/Number from manual input.

This is a task that can be fully automated in many situations. As an admin you could provide a text file mapping serial numbers to a name and/or asset tag on a server, that a script can download (`curl`) and parse. If you have a management system or asset database, there is probably an XML or JSON API, you can use to retrieve this information from a script. You could read or scan the serial numbers from the labels of the Mac's boxes or even get the list with the purchase order from most vendors. With that data you can pre-fill your text files or databases.

However, in some cases, especially DEP workflows it may be difficult or impossible to predict which computer will end up on which desk,

especially with zero-touch deployment workflows, where a device can be sent to a user directly.

In this case, you have to question whether you need a unique, specific computer name or asset tag. Maybe the data which your management system already gathers, such as the user who enrolled the device and its serial number will (have to) be sufficient?

In many cases, however, the reasons to prompt for and set a computer name will not be technical but stem from other, external factors that the IT department may or may not have influence on.

AppleScript's Moment of Glory

`bash` was built to run in a text based terminal on many different operating systems. It has (and should have) no concept of a graphical user interface. `bash` alone is not useful to interact with the user, unless you want to open a terminal window.

However, AppleScript does have some (basic) commands to present dialogs and notifications to the user. We can call AppleScript command from the shell with the `osascript` command (OSA = open scripting architecture, the underlying framework that AppleScript uses)

> *There are other tools that allow to display user interface from a shell script. However, they all require an additional installation. AppleScript/`osascript` is simpler and built-in to the OS, so it is my first choice. That doesn't mean it is* always *the appropriate choice, though.*

You can go to Terminal and make a dialog appear with

```
$ osascript -e 'display dialog "Hello from bash!"'
```

The `display dialog` AppleScript command is documented in the "StandardAdditions" dictionary. You can see it when you choose 'Open Dictionary…' from the File Menu in Script Editor. Then select the "Scripting Additions.osax" dictionary and choose the "User Interaction" category.

This command has a lot of options that allow us to configure the dialog. You can experiment and test with these commands in the Script Editor application. For example, you can change the names of the buttons:

```
display dialog "Are you sure!?" buttons {"No", "Yes"}
```

You can also have just one button:

```
display dialog "Just accept it!" buttons {"Accept"} default button
1
```

Or three: (three is the maximum)

```
display dialog "The answer is C" buttons {"A", "B", "C"} default
button 3
button returned:C
```


The script will return which button the user clicked. However, instead of parsing the output with shell tools, you should let AppleScript do

the work:

```
button returned of (display dialog "Are you sure!?" buttons {"No", "Yes"})
```

The `choice` variable will be `No` or `Yes`.

Adding icons

You can also add an icon to the dialog:

```
display dialog "Hello" with icon note
```

Will show the current application's icon. You can also use `stop` or `caution` for different icons.

You can also add a path to an `icns` file:

```
display dialog "Hello!" with icon POSIX file "/Applications/Notes.app/Contents/Resources/AppIcon.icns"
```

Asking for Input

In some cases you want to get information back from the user. You can add a text field to the dialog with the `default answer` argument. The default answer can be an empty string:

```
display dialog "Who are you?" default answer "nobody"
display dialog "Who are you?" default answer ""
```

You can get the result of the dialog with the `text returned` property:

```
text returned of (display dialog "Who are you?" default answer
"nobody")
```

You can combine the `default answer` argument with all the above arguments as well.

Notifications

In some case you just want to notify the user that something happened, and not stall everything while the script waits for confirmation. You can use `display notification` for these situations:

```
display notification "Hello, again" with title "Hello"
```

Running from Shell

You can execute AppleScript from the shell with the `osascript` command.

```
osascript -e 'display dialog "Hello!" with icon note'
```


This works well enough. You also use shell variable substitution:

```
title='Hey, there!'
osascript -e "display dialog \"What's up?\" with title \"$title\""
```

When you use variable substitution, you have to use double quotes for the command strings. And then you have to escape any additional double quotes in the AppleScript command. With more complex commands and arguments this will get unwieldy very quickly.

In a bash script you can use a here document instead:

```
title='Hey, there!'
osascript <<-EndOfScript
display dialog "What's Up?" with title "$title"
EndOfScript
```

You start a here document with the <<- characters followed by a limit string of your choice (I chose `EndOfScript`). Then all the text until the limit string is repeated will be fed into the `osascript` command.

Bash variables will be substituted in the here document, so `$title` will be substituted with its contents.

Note: I prefer the <<- syntax over the << syntax with `osascript`. The <<- syntax will ignore leading white space in the following lines, allowing me to properly indent the AppleScript code, making the entire script more legible.

The Trouble with `root`

Management scripts will run with `root` privileges. They may also be run in situations where no user is logged in. AppleScript requires a user to be logged in (and a window server to be present) to display the alert.

In many situations all of this will 'just work' even when the script is executed from a `root` process, in others, the script will fail. It is generally safer to always check the current user and run the `osascript` command as the currently logged in user. You can learn about how and why to do this in this article on 'Root and Scripting'.

This also gives your script an option to continue silently or fail when no user is logged in.

```
user=$(python -c 'from SystemConfiguration import
SCDynamicStoreCopyConsoleUser; import sys; username =
(SCDynamicStoreCopyConsoleUser(None, None, None) or [None])[0];
username = [username,""][username in [u"loginwindow", None, u""]];
sys.stdout.write(username + "\n");')
if [[ $user != "" ]]; then
    uid=$(id -u "$user")
    launchctl asuser $uid /usr/bin/osascript -e "button returned
of (display dialog \"Hello\")"
fi
```

Putting it all Together

While all of these examples are simple enough, you can already see that, once you consider all the possible combinations, everything will get fairly complex.

Over time I have put together a few bash functions that I use in my scripts. Even they only cover quite simple workflows, but can be useful as a sample for more complex needs:

```
#!/bin/bash

export PATH=/usr/bin:/bin:/usr/sbin:/sbin

consoleUser() {
        echo "show State:/Users/ConsoleUser" | scutil | awk '/Name :/ && !
}

displaydialog() { # $1: message
        message=${1:-"Message"}
        user=$(consoleUser)
    if [[ $user != "" ]]; then
        uid=$(id -u "$user")
                launchctl asuser $uid /usr/bin/osascript <<-EndOfScript
                        button returned of ¬
                            (display dialog "$message" ¬
                                buttons {"OK"} ¬
                                default button "OK")
                EndOfScript
        fi
}

displaynotification() { # $1: message $2: title
        message=${1:-"Message"}
        title=${2:-"Script Notification"}
        user=$(consoleUser)
    if [[ $user != "" ]]; then
        uid=$(id -u "$user")
                launchctl asuser $uid /usr/bin/osascript <<-EndOfScript
                        display notification "$message" with title "$title
                EndOfScript
        fi
}
```

```
35
36  displayfortext() { # $1: message $2: default text
37          message=${1:-"Message"}
38          defaultvalue=${2:-"default value"}
39          user=$(consoleUser)
40      if [[ $user != "" ]]; then
41          uid=$(id -u "$user")
42
43              launchctl asuser $uid /usr/bin/osascript <<-EndOfScript
44                          text returned of ¬
45                                  (display dialog "$message" ¬
46                                          default answer "$defaultvalue" ¬
47                                          buttons {"OK"} ¬
48                                          default button "OK")
49                          EndOfScript
50          else
51              exit 1
52          fi
53  }
54
55  # run all functions
56
57  displaydialog "Hello"
58  name=$(displayfortext "Enter your name:" "$(consoleUser)")
59  displaynotification "Hello, $name"
```

bashdisplay.sh hosted with ❤ by GitHub view raw

System Events, Privacy and macOS Mojave

In many scripts the author wraps the `display dialog` command wrapped in a `tell` statement:

```
osascript -e 'tell app "System Events" to display dialog "Hello"'
```

or, like this:

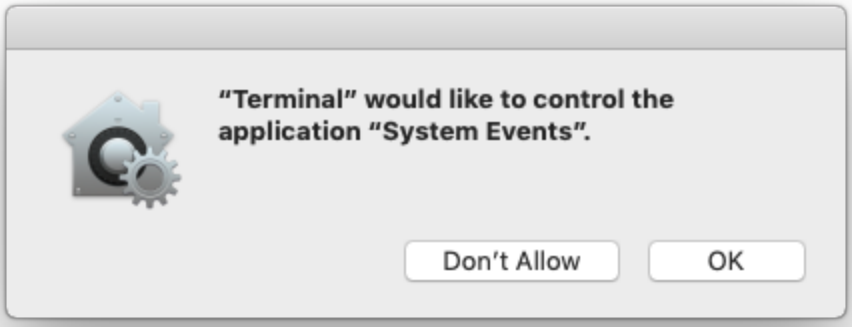
```
osascript <<-EndOfScript
tell application "Finder"
    activate
    display dialog "Hello"
end tell
EndOfScript
```

The reason for this is that it will ensure that the dialog is displayed on top of all other windows. The `activate` command will push the targeted process to the front. Both “System Events” and “Finder” are used frequently.

In most cases this is not necessary, as the dialog will properly display on top of all other windows, even as a standalone command. There may be some weird conditions when other applications are launched in the same time frame, though. In other cases it may be better to use `display notification`, anyway.

In macOS Mojave, Apple Events (AppleScript Commands) can *only* be sent to another application with user permission.

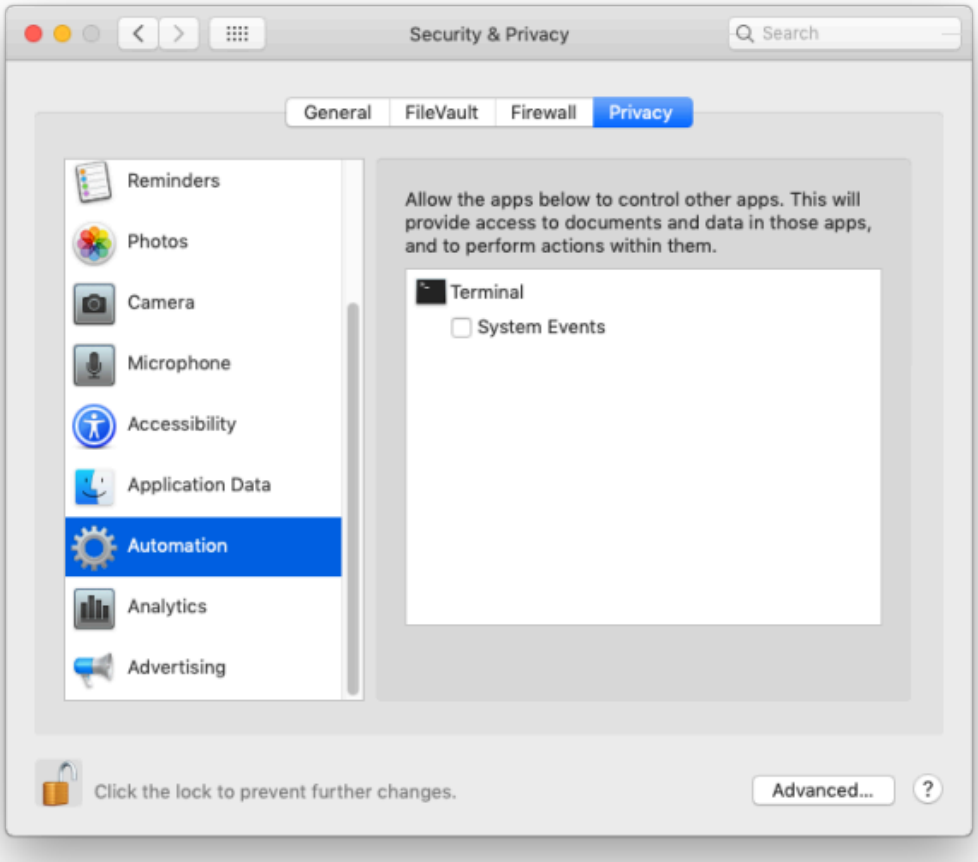
When you try this command in macOS Mojave, the user will be prompted to allow the Terminal application to control the System Events application



If the user denies this request, the `osascript` command will fail with an error:

```
$ osascript -e 'tell app "System Events" to display dialog
"Hello"'
28:50: execution error: Not authorized to send Apple events to
System Events. (-1743)
```

Once a user has denied access, they will not be prompted again. They will have to go to the 'Privacy' tab in the 'Security & Privacy' pane in System Preferences, search for 'Automation' and allow access for Terminal.



However, management scripts will usually *not* be executed from Terminal, but from within the context of an installation script or management agent. It may be possible to pre-approve configurations with a UAMDM configuration profile, but it will be impossible to anticipate all contexts in which your scripts may run.

For macOS Mojave it will be better to either modify your script to **not** wrap the command in a `tell` statement. Alternatively, you can use a different solution for the UI altogether. (e.g. `jamfHelper`, Yo, Pashua or CocoaDialog)

All of these wrapped commands in scripts will break in macOS Mojave!

MacAdmins need to go through all their management scripts and check for AppleScript UI commands wrapped in `tell` statements.

When you use the standalone `display dialog` it will be the AppleScript process itself that displays the dialog. This requires no specific permission.

Even if your dialogs may not appear on top of all the windows, this is a preferable solution.

This will also affect scripts using `osascript` to control or receive data from other applications.

Summary

- question and revisit every use of user interaction in your workflow
- when you *really* have to, you can use `osascript` with the `display` AppleScript commands
- to be safe, run the commands as the current user
- increased macOS Mojave security will require you to verify *all* your scripts using `osascript`

PUBLISHED BY

ab

Mac Admin, Consultant, and Author View all posts by ab →

2018-08-22 ab Scripting

5 thoughts on “User Interaction from bash Scripts”

g*
2018-10-06 at 13:41

Hi, thanks for this. Quite useful. My question: is it possible to use a custom icon for a notification? I can see how it is done for a dialog, but not a notification.

ab
2018-10-09 at 08:39

Totally custom icons are not possible with AppleScript. You can make it display another app’s icon by sending the ‘display notification’ command to that app, i.e. ‘tell app “Finder” to display notification …’ however, on Mojave, this will run afoul of the privacy protection and you will have to whitelist your process to be able to send events to Finder.

g*
2018-10-09 at 20:10

Aah, I see. Thanks for the response.

Gahis
2020-12-11 at 08:34

I was wondering how to create an automator shell script to rename files, while prompting with window (or terminal) to enter the new name. I have spend months trying to do this using bash script in automator, unsuccessfully.

for f in *.md
do
mv “$f” “test – $f”
done

This is basically the idea in script form. What I need is the “test” to be a prompted dialog where I enter what I want.

ab
2020-12-14 at 09:11

This is where you hit the limitations or Automator. Adding extra data into an action is nearly impossible without building custom actions. But, as it happens, Automator already has a custom action called “Rename Finder Items” which is quite flexible and powerful.

As an extra note: when you have multiple items selected in Finder and choose “Rename…” from the context menu you will get a simplified version of this rename interface.

Comments are closed.

PREVIOUS

Weekly News Summary for Admins — 2018-08-17

NEXT

Weekly News Summary for Admins — 2018-08-24