Scripting

Frame Guest Agent (FGA) gives customers the ability to tailor the behavior of their workload VMs (e.g., Sandbox, shadow, production, utility server, etc.) to execute scripts at different event-points of a VM and/or user session. These events are defined as “lifecycle hooks.”

Once a Frame account is created, the Frame administrator can place custom scripts (PowerShell for Windows, Bash scripts for Linux) in the Sandbox and then publish the Sandbox to make them available to the workload VMs.

Custom Script Location

> In the context of scripts, spaces are not valid characters. For example `script1.ps1` is a valid script name, while `script 1.ps1` is *not* valid.

FGA scripts must be placed into a specific directory. The FGA Users Scripts directory for **Windows** VMs is:

`C:\ProgramData\Nutanix\Frame\FGA\Scripts\User`

Example valid script paths for **Windows**:

- `C:\ProgramData\Nutanix\Frame\FGA\Scripts\User\powershell\my_custom_script.ps1`
- `C:\ProgramData\Nutanix\Frame\FGA\Scripts\User\my_custom_script.ps1`
- `C:\ProgramData\Nutanix\Frame\FGA\Scripts\User\a\b\c\d\my_custom_script.ps1`

For **Linux** VMs, the FGA Users Scripts directory is:

`/opt/nutanix/frame/fga/scripts/user`

Example valid script paths for **Linux**:

- `/opt/nutanix/frame/fga/scripts/user/powershell/my_custom_script.sh`
- `/opt/nutanix/frame/fga/scripts/user/my_custom_script.sh`
- `/opt/nutanix/frame/fga/scripts/user/a/b/c/d/my_custom_script.sh`

Script Invocation

Invoking or executing custom scripts requires a definition.yml file (case sensitive) to be present in the FGA User Scripts directory. For example, this path for Windows would be `C:\ProgramData\Nutanix\Frame\FGA\Scripts\User\definition.yml`. FGA reads this file at each lifecycle hook and will invoke scripts as defined in this file.

Definition.yml

The `definition.yml` file describes detailed script context and execution information:

- Script **groups** -- a convenient way to group multiple scripts based on their context.
- Lifecycle hooks or **run-policies** -- phases in Frame's lifecycle of VMs and Sessions.
- Script **paths** and filenames -- only use relative paths (from the FGA Users Script directory) or filenames. Look out for typos!
- **Pool-types** -- specifies which type of Frame VMs you'd like the scripts to execute on (Sandbox, Production, etc.)
- **Error policies** -- specifies the behavior FGA should follow if an error is encountered: continue or abort.

Below is an example of a definition.yml file.

```
groups:
-
  desc: Scripts on first boot
  name: first-boot
  timeout: 30
  run-policy: first-boot
  pool-type:
    - sandbox
    - production
  scripts:
    -
      desc: My script A
      path: A.ps1
      error-policy: continue
      timeout: 10
    -
      desc: My script B
      path: B.ps1
      error-policy: continue
      timeout: 10
-
  desc: Scripts on each boot
  name: every-boot
  timeout: 30
  run-policy: every-boot
  pool-type:
    - sandbox
    - production
  scripts:
    -
```

```
        desc: My script C
        path: C.ps1
        error-policy: continue
        timeout: 10
      -
        desc: My script D
        path: D.ps1
        error-policy: continue
        timeout: 10
  -
    desc: Scripts before session is started
    name: pre-session
    timeout: 30
    run-policy: pre-session
    pool-type:
      - sandbox
      - production
    scripts:
      -
        desc: My script before sessions is started
        path: before-session.ps1
        error-policy: continue
        timeout: 20
  -
    desc: Scripts after session is closed
    name: post-session
    timeout: 30
    run-policy: post-session
    pool-type:
      - sandbox
      - production
    scripts:
      -
        desc: My script after sessions is closed
        path: after-session.ps1
        error-policy: continue
        timeout: 20
```

Script Execution Order

Order in the definition.yml file is important. When executing the scripts, FGA iterates through the definition.yml file and executes scripts in order from top to bottom. This also applies to Script Groups. If you provide two script groups of the same type (pre-session, post-session, etc), the group that is higher in the YAML file will get executed before any groups that are lower in the file.

In the following definition.yml example, Group 3 will get executed before Group 1 (since Group 3 is higher in the file). Also, `before-session-3.ps1` will get executed prior to `before-session-2.ps1` for the same reason:

```
groups:
-
  desc: Group 3 - Scripts before session is started
  name: Group 3 Pre-session
  timeout: 30
  run-policy: pre-session
  pool-type:
    - production
  scripts:
    -
      desc: The 3 pre-session script
      path: before-session-3.ps1
      error-policy: continue
      timeout: 20
    -
      desc: The 2 pre-session script
      path: before-session-2.ps1
      error-policy: continue
      timeout: 20
-
  desc: Group 1 - Scripts after session is closed
  name: Group 1 pre-session
  timeout: 30
  run-policy: pre-session
  pool-type:
    - production
  scripts:
    -
      desc: My script after sessions is closed
      path: after-session.ps1
      error-policy: continue
      timeout: 20
```

> **Important!**
> Each script will need to be fully completed before the next script is executed. (Sequential execution vs. parallel execution)

Frame Lifecycle Hooks

The following table outlines various Frame Lifecycle Hooks, used as values for run-policy in a definition.yml script group. Each lifecycle hook is optional.

VM Lifecycle Hooks

Hook Name	Description	Applicable pool groups	User Context

`pre-generalization`	Executed before Domain-Join generalization process (which includes sysprep) is started.	shadow, persistent_desktop_shadow	**Frame** user
`post-generalization`	Executed after Domain-Join generalization process (which includes sysprep) is finished.	shadow, persistent_desktop_shadow	**Frame** user
`first-boot`	Executed only on the very first boot of the virtual machine, after it is created. This hook is available for all instance types, allowing scripts to make stateful changes right after instances are provisioned/created. For domain joined instances and enterprise profiles, the local user `frameuser` needs to have elevated privileges.	shadow, production, persistent_desktop_shadow, persistent_desktop_production	**Frame** user
`every-boot`	Executed upon every system/OS boot. This hook can be used to update the image before non-persistence is enabled (on shadow/production instances).	shadow, production, sandbox, utility	**Frame** user

Session Lifecycle Hooks

Hook Name	Description	Applicable pool groups	User Context

`pre-session`	Executed right before a user session is started. **User Context info:** For example, if the workload VM is joined to the domain and the user authenticates to the Windows domain, then the scripts would run in the domain user context. Pre-session scripts are executed after the Windows login screen and before the onboarded application or desktop is displayed. Pre-session scripts never run in SYSTEM context.	sandbox, production, persistent_desktop_production, utility	**Currently logged in user**
`on-idle`	Executed when session goes to idle state (workload stops streaming).	sandbox, production, persistent_desktop_production, utility	**Currently logged in user**
`on-active`	Executed when session goes from idle to active state (workload resumes streaming).	sandbox, production, persistent_desktop_production, utility	**Currently logged in user**
`post-session`	Executed immediately after a session is closed. This occurs after the session has stopped streaming and before both the Windows logoff process and the profile disk unmount process.	sandbox, production, persistent_desktop_production, utility	**Currently logged in user**

Script User Context (Windows)

It is important for Frame administrators to know which Windows user is running which script. Refer to the table above to see each lifecycle hook and their associated execution context. Frame's default `Frame` user is a member of the local Windows administrator group.

To determine the user context, you can execute `whoami` within your script and it will print out the current script context.

Script Group Properties

desc

> A detailed description of the group.

name

> A descriptive name of the group.

timeout

> Integer value in seconds. Defines the maximum amount of time for *all scripts executing within a group*. If the execution time of the scripts exceeds this timeout value, FGA will kill the currently running script and skip any remaining the scripts in the group.

run-policy

> Defines the Frame Lifecycle Hook that will execute the scripts.

pool-type

> Defines target pool types on which the scripts in the group are going to be applied. If none of the pool types is set, FGA will execute the script on all pool types.
>
> **Pool types and descriptions**
>
> - `sandbox` - Scripts are going to be applied only on sandbox VMs
> - `production` - Scripts are going to be applied only on production VMs
> - `shadow` - Scripts are going to be applied only on shadow VMs that are part of the publishing process
> - `utility` - Scripts are going to be applied only on utility VMs
> - `persistent_desktop_production` - Scripts are going to be applied on persistent desktops

> - `persistent_desktop_shadow` - Scripts are going to be applied on freshly provisioned persistent desktops that are not yet assigned

Newly published VMs that have been placed in the production pool will still be marked as `shadow` until they are rebooted.

scripts

> Specifies the list of scripts that needs to be executed.

Did you know?
A “pool” in Frame's context is a grouping of VMs/instances associated with a Frame account.

Script Item Properties

desc

> Description of the script.

path

> Script file location, relative to base user scripts directory.

error-policy

> Defines FGA strategies when script fails.

> Follow these three simple steps to get started creating scripts and customizing the VMs on first boot (or every boot) and your Frame sessions.
>
> **Error-policy values**
>
> - `continue` - FGA to proceed on even if the script fails.
> - `abort` - FGA to abort the task if script fails; **use with caution.**
>   - For Session Lifecycle hooks, this will immediately end the session for the end-user.
>   - For VM Lifecycle hooks, this can prevent publishes from completing, or spinning up new instances when the pool's capacity is expanded

timeout{#script-timeouts}

> Integer value in seconds. Defines the maximum amount of time for the script to run. If the execution time of the script exceeds the timeout value, FGA will kill the running script. **Be sure that your script's combined timeouts are less than or equal to the group's timeout duration.**

Getting Started

Follow these three simple steps to get started creating scripts and customizing the VMs on first boot (or every boot) and your Frame sessions.

1. Create a Simple Script

Create a new PowerShell file (hello-world.ps1) in the FGA Scripts User directory.

```
# Writing a simple text file with warm greeting.
$timestamp = Get-Date -Format "dddd MM/dd/yyyy HH:mm K"
"$timestamp - Hello, World!" | Out-File
"C:\ProgramData\Nutanix\Frame\Logs\sandbox_greeting.txt" -Append
```

2. Your *definition.yml*

Next, you need to communicate to FGA about this script via a `definition.yml` file to specify which Frame Session Lifecycle hook will trigger this script, timeout settings, the pool type(s),

etc.

```
groups:
-
  desc: Simple script example
  name: A very simple pre-session script.
  timeout: 10
  run-policy: pre-session
  pool-type:
    - sandbox
  scripts:
    -
      desc: Frame Hello World Example script!
      path: hello-world.ps1
      error-policy: continue
      timeout: 10
```

The `definition.yml` file specifies that we have one script group that only executes during the pre-session Lifecycle Hook. We also specify that this group should only execute on the sandbox instance *pool*. Next, we specify a single script called `hello-world.ps1`. This script can run for a maximum of 10 seconds before reaching either timeout value, causing FGA to quit waiting and continue on.

3. Testing and Debugging

Finally, double-check to make sure the files are saved, named correctly, and are located in the correct locations. Next, quit your sandbox session. Now we're ready to test!

Testing our script is simple. When starting a new Sandbox Session, you should see a file populated at `C:/sandbox_greeting.txt`, showing our execution timestamp and our greeting.

Logging and Troubleshooting

Logging can be incredibly useful for debugging. However, if you're using non-persistent desktops and trying to debug scripts in production, any logging you do on the VM is wiped out after the session.

Logging tips for various environments

1. **Sandbox and Persistent Desktops**: Create any text file somewhere on C: with the output of your logs.
2. **Non-persistent Shadow and Production VMs**: External logging service, either on the public internet (like Loggly, Splunk, etc.) or perhaps a service running on an accessible Utility Server.

Troubleshooting Tips

- Be mindful of about group and script timeouts, how they're different, and that your values don't overlap.
- Always take backups before scripting.
- If your scripts reference any file paths, be sure to use absolute filepaths, even for files residing in the FGA User Scripts folder.
- If possible, test by executing the Powershell code in the Sandbox using PowerShell ISE.
- If your scripts contain any sensitive information or credentials, be sure to write a cleanup pre-session script that can execute after your credentials are used and *delete the sensitive script/files* before users are connected to a Session.
- Use identifying elements from Frame environment variables to help understand who and when (if needed).
- You can set and get environment variables or registry entries to help communicate between scripts and lifecycle hooks.
- Avoid using `error-policy: abort` for sandbox scripts, as failing scripts can make the Sandbox inaccessible, requiring a restoration from a backup or termination (and recreation) of the Sandbox.
- If your scripts need to run with Elevated Privilege in Windows 10, make sure that the below registry key value is set to `0`. Otherwise, your scripts will fail due to Microsoft Windows User Account Controls (UAC) preventing the script from executing in an elevated context.
  - Key: `SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System`
  - Value: "EnableLUA"
  - Type: `REG_DWORD`

Environment Variables

FGA populates environment variables via registry entries to `HKCU/Environment`. These values are dynamically set for each Frame session and are useful when used with logging.

Env Variable	Description
`FRAME_VENDOR_ID`	A unique identifier tied to the Frame Account the VM is associated with.
`FRAME_VENDOR_NAME`	The name of the instance's Account.
`FRAME_USER_EMAIL`	The current session user's email address.
`FRAME_SESSION_ID`	The unique Session ID associated with the current session. Useful for debugging and support purposes.

Env Variable	Description
`FRAME_SESSION_INFO`	Provides additional session information represented within a JSON object: `colorspace` (`YUV420` or `YUV444`), `protocol` (`FRP7` or `FRP8`), and `connectionType` (`tcp` or `udp`)
`FRAME_SESSION_LABEL`	Custom value set in Session Settings through the Advanced Server Argument `-frame-session-label`
`FRAME_SESSION_TOKEN`	Unique token specific to the session. This token can be used to query user assertion/claims data, or base64 decoded to access various user and authentication-related data.
`FRAME_VENDOR_EMAIL`	Contrary to the name, this value is the unique GUID associated to the Frame account the VM belongs to.
`FRAME_MAX_SESSION_DURATION`	The maximum amount of time (in seconds) the user can be in the session. This value is set in the Account, Launchpad, Sandbox, and Utility Server session settings.
`FRAME_SESSION_START_TIME`	The date/time when the session started.
`FRAME_POOL_GROUP_TYPE`	Type of pool group the session is currently connected to (`sandbox`, `production`, `test`, `utility`).
`FRAME_POOL_NAME`	Name of instance pool the session is currently connected to as defined within *Dashboard > Capacity*
`FRAME_INSTANCE_TYPE`	The instance type name of the current instance.
`FRAME_IMAGE_FAMILY`	Image Family Name that the instance is based off of.
`FRAME_DATACENTER`	Name of the datacenter where the instance is located.
`FRAME_CLOUD_PROVIDER`	Name of the infrastructure provider (`ahv`, `azure`, `gcp`, `amazon`).

Frame Script Helper Tool

Frame Script Helper is a tool designed for the creation and maintenance of the ***definition.yml*** file. Frame Script Helper can be found in `C:\ProgamData\Nutanix\Frame\Tools`. Key tool features include the import, validation, editing, and export of definition.yml files.

> As this tool was written for Windows, it only supports `.ps1` scripts.

Import and Validation of the Definition.yml File

Importing the definition.yml file can be accomplished in two ways:

- **Automatically**: When launched the tool will search for a definition.yml file in `C:\ProgramData\Nutanix\Frame\FGA\Scripts\User`
- **Manually**: By using the import section from toolbar menu: Configuration>Import

> The file will not be imported until any errors are resolved.

If the tool cannot find a definition.yml file at startup, it will display a prompt; you can still continue using the tool. All changes will be saved afterwards into a new definition.yml file.

Edit the Definition.yml File

All available Frame Lifecycle hooks (see above for hook definitions) are displayed to the left of the tool window.

To begin, select one of the hooks. By default, "first-fga-boot" is selected. Next to the name of the hook are two numbers. The first number represents the number of pool-type groups, the second represents the total number of scripts assigned to that hook.

To attach your scripts to the selected hook, drag scripts files from File Explorer into the drop area at the top of the screen.

> Another way to import the scripts is to directly drag their file onto the existing pool-type group.

Next, a prompt will ask you to specify the pool-type groups for your scripts. Check the box next to each desired pool and click “Next” when all desired options are checked.

Frame Script Helper Tool - Pool Types

Changing Script Order

To change the order of the scripts, simply drag a script to desired position in the pool-type group.

>

> Scripts can only be moved within the current group, not to another.

Rearranging the Pool-Type Groups

Order of the pool-types can be also changed. Each of the pool-type group "box" can be dragged over the other group.

Export the Definition.yml File

After making your changes, you can save your file by clicking “File” and then “Save”, or even export it to a new file from **Configuration > Export**.

Advanced Scripting

Update Script (Windows only)

If found, FGA will execute an “update” script right after a VM boots, but *before FGA User scripts are executed*. This allows Frame administrators to create custom update scripts to automated or dynamic maintenance operations on their custom scripts and files before user scripts defined in definition.yml are executed. The update script must be in the following path:

`C:\ProgramData\Nutanix\Frame\FGA\Scripts\User\update.ps1`

Updating the user scripts might include completely new packages of scripts or just a new definition.yml file. The following are best practices when it comes to updating your custom scripts using the Update script feature.

1. Make sure your updated scripts and/or definition.yml are packaged in a bundle.
2. Make sure your workload VMs are able to access and download that bundle.
3. Create an update script that downloads the package and checks if there is a need for the update to be applied.
   - If not, the "update" script should just exit.
   - If yes, the "update" script should perform the update (it can delete existing definition.yml and all script files and then recreate definition.yml file and/or scripts).
4. Place the update script content inside the update.ps1 (for Windows) and update.sh (for Linux).

When the VM gets into boot phase, FGA is going to search for the "update" script. If it does not find the script, FGA will read the definition.yml file and run the scripts as they are defined in the definition.yml file. If the FGA finds the "update" script, FGA will execute the script first. Once the script is executed, FGA will proceed to execute the custom scripts following the definition.yml file.

If the update script on Windows OS takes more than 10 minutes to execute, FGA will timeout and continue with its startup workflow.

Example Scripts

Below are some example use cases where a custom script and associated definition.yml file can customize the end user experience.

Mount a Network Volume

Scenario: A Frame administrator would like to mount an existing read-only file share once a user starts a Frame session in either the Sandbox (Frame administrator) or a production workload VM (user).

The following simple PowerShell script "mount-read-only-volume.ps1" is placed in the scripts folder to be executed:

```
echo off
net use * /delete /Y
net use K: \\10.0.0.5\Fileshare /user:username password
```

The corresponding **definition.yml** file:

```
groups:
-
  desc: Scripts before session is started in either Sandbox or production workload VMs
  name: pre-session
  timeout: 30
  run-policy: pre-session
  pool-type:
    - sandbox
    - production
  scripts:
    -
      desc: Pre-session script to mount a read-only volume
      path: mount-read-only-volume.ps1
      error-policy: continue
      timeout: 20
```

Exclusion/Inclusion Rules for Enterprise Profiles

Scenario: A Frame administrator wishes to exclude specific folders from persisting and/or include specific folders to persist in users' enterprise profiles. The following PowerShell script “exclude-include-folders.ps1” is placed in the scripts folder to be executed:

```
# Exclude four folders in the %USERPROFILE%
Add-ProfileDiskExclusion -SourcePath $env:USERPROFILE\Downloads -TargetPath C:\_Profile
Add-ProfileDiskExclusion -SourcePath $env:USERPROFILE\Music -TargetPath C:\_Profile
Add-ProfileDiskExclusion -SourcePath $env:LOCALAPPDATA\Autodesk -TargetPath C:\_Profile
Add-ProfileDiskExclusion -SourcePath $env:LOCALAPPDATA\Spotify -TargetPath C:\_Profile

# Include two separate folders C:\MyData and C:\MoreData to the user's enterprise profile.
Add-ProfileDiskInclusion -SourcePath C:\MyData
Add-ProfileDiskInclusion -SourcePath C:\MoreData
```

The corresponding definition.yml file specifies that this PowerShell script should only be executed in the context of production workload VMs where the user's enterprise profile volume is used (in a non-persistent Frame account):

```
groups:
-
  desc: Scripts before session is started in production workload VMs
  name: pre-session
  timeout: 30
  run-policy: pre-session
  pool-type:
    - production
  scripts:
    -
      desc: Pre-session script to include/exclude folders from the user's enterprise profile
      path: exclude-include-folders.ps1
      error-policy: continue
      timeout: 20
```

Autohide the Windows Task Bar in App Mode 2.0

Scenario: A Frame administrator wants to have the Windows Task Bar automatically hidden when using an Application Launchpad with App Mode 2.0. The Windows registry key that controls this behavior is at `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\StuckRects3`. The following PowerShell script is placed in the scripts folder to be executed:

```
$p='HKCU:SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\StuckRects3';
$v=(Get-ItemProperty -Path $p).Settings;
$v[8]=1;
Set-ItemProperty -Path $p -Name Settings -Value $v;
```

```
Stop-Process -f -ProcessName explorer
```

The corresponding definition.yml file specifies that this PowerShell script should only be executed in the context of production workload VMs:

```
groups:
-
  desc: Scripts before session is started in production workload VMs
  name: pre-session
  timeout: 30
  run-policy: pre-session
  pool-type:
    - production
  scripts:
    -
      desc: Pre-session script to autohide the Windows Task Bar
      path: autohide-taskbar.ps1
      error-policy: continue
      timeout: 20
```

If you wish to show the Windows Task Bar, then the PowerShell script can be used:

```
$p='HKCU:SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\StuckRects3';
$v=(Get-ItemProperty -Path $p).Settings;
$v[8]=2;
Set-ItemProperty -Path $p -Name Settings -Value $v;
Stop-Process -f -ProcessName explorer
```

Add Applications dynamically to the Frame Taskbar in App Mode 2.0

Scenario: A Frame administrator needs to dynamically add an application shortcut/launch link to the Frame Taskbar in App Mode 2.0. App Mode 2.0 allows you to dynamically modify the Frame Taskbar to any configuration you need (in real time).

The following PowerShell script should be placed in the scripts folder to be executed. This is designed as a pre-session script that modifies the Frame Taskbar configuration before the session starts. The script will override any of the default Frame Taskbar configuration and show Notepad only (update-taskbar.ps1):

> Notice that any URI path requires two backslashes in the path (Within the JSON file only)

```
$jsonFilePath = 'C:\ProgramData\Nutanix\Frame\Config\server_launchpad_override.json'

$jsonOverrideContent = @"
{
    "name": "root",
    "order": 0,
    "applications": [
        {
            "name": "Frame",
            "path": "C:\\Windows\\System32\\notepad.exe",
            "icon_url": "",
            "order": 0,
     "arguments": ""
        }
    ],
    "folders": []
}
"@

Set-Content -Path $jsonFilePath -Value $jsonOverrideContent
```

The corresponding definition.yml file specifies that this PowerShell script should only be executed in the context of production workload VMs:

```
groups:
-
  desc: Scripts before session is started in production workload VMs
  name: pre-session
  timeout: 30
  run-policy: pre-session
  pool-type:
    - production
  scripts:
    -
      desc: Pre-session script to add Notepad into the Frame Taskbar icon/shortcut list
      path: update-taskbar.ps1
      error-policy: continue
      timeout: 20
```

Playing in the Sandbox

In this example, we'll create a few more scripts, each set to execute at common session lifecycle hooks.

> Before continuing with any changes to your sandbox, please make a backup first! Sandbox scripts, if improperly configured, can cause undesired behaviors and in some cases can cause the sandbox to become unresponsive, requiring restoration from a

> backup or a fresh sandbox.

To get started, grab your shovel and connect to your Sandbox. Then, open up your favorite text editor and create the following **five** files in the **FGA User Scripts Directory**.

Pre-session script: pre-session.ps1

To test this, the scripts will create or append to a single log file created at `.\logs\sandbox_bucket.txt` with a timestamp and message from the script.

```
$RootPath = "C:\ProgramData\Nutanix\Frame\FGA\Scripts\User"
If (!(test-path "$RootPath\logs")){ New-Item -Path "$RootPath\logs" -Type Directory }
$timestamp = Get-Date -Format "dddd MM/dd/yyyy HH:mm K"
"$timestamp - Hello from pre-session.ps1" | Out-File "$RootPath\logs\sandbox_bucket.txt" -
Append
```

Session Disconnect script: on-idle.ps1

```
$RootPath = "C:\ProgramData\Nutanix\Frame\FGA\Scripts\User"
If (!(test-path "$RootPath\logs")){ New-Item -Path "$RootPath\logs" -Type Directory }
$timestamp = Get-Date -Format "dddd MM/dd/yyyy HH:mm K"
"$timestamp - Hello from on-idle.ps1" | Out-File "$RootPath\logs\sandbox_bucket.txt" -Append
```

Session Resume script: on-active.ps1

```
$RootPath = "C:\ProgramData\Nutanix\Frame\FGA\Scripts\User"
If (!(test-path "$RootPath\logs")){ New-Item -Path "$RootPath\logs" -Type Directory }
$timestamp = Get-Date -Format "dddd MM/dd/yyyy HH:mm K"
"$timestamp - Hello from on-active.ps1" | Out-File "$RootPath\logs\sandbox_bucket.txt" -
Append
```

Post-session script: post-session.ps1

```
$RootPath = "C:\ProgramData\Nutanix\Frame\FGA\Scripts\User"
If (!(test-path "$RootPath\logs")){ New-Item -Path "$RootPath\logs" -Type Directory }
$timestamp = Get-Date -Format "dddd MM/dd/yyyy HH:mm K"
"$timestamp - Hello from post-session.ps1" | Out-File "$RootPath\logs\sandbox_bucket.txt" -
Append
```

Let's walk through each line in these files.

- **Line 1**: This line declares the root path as a variable variable. We could also use `Set-Location`.
- **Line 2**: Creates the log folder inside of the root path if it doesn't exist.

- **Line 3**: Gets a timestamp in a readable format (based on the Sandbox's local time).
- **Line 4**: Writes our timestamp and hello message to the specified "sandbox_bucket" txt file.

Defining our scripts: definition.yml

We'll create four different script *groups* (one for each lifecycle hook). These scripts will only execute on the Sandbox; each will have a total of 5 seconds to execute (timeout); and if these scripts fail for any reason, FGA will continue with the next lifecycle procedures.

```
groups:
-
  desc: Pre-session script
  name: Pre-session script group
  timeout: 5
  run-policy: pre-session
  pool-type:
    - sandbox
  scripts:
    -
      desc: Example sandbox pre-session script.
      path: pre-session.ps1
      error-policy: continue
      timeout: 5
-
  desc: On-idle disconnect script
  name: On-idle script group
  timeout: 5
  run-policy: on-idle
  pool-type:
    - sandbox
  scripts:
    -
      desc: Example sandbox on-idle script.
      path: on-idle.ps1
      error-policy: continue
      timeout: 5
-
  desc: On-active resume script
  name: On-active script group
  timeout: 5
  run-policy: on-active
  pool-type:
    - sandbox
  scripts:
    -
      desc: Example sandbox on-active script.
      path: on-active.ps1
      error-policy: continue
      timeout: 5
-
  desc: Post-session script
  name: Post-session script group
  timeout: 5
```

```
run-policy: post-session
pool-type:
  - sandbox
scripts:
  -
    desc: Example sandbox post-session script.
    path: post-session.ps1
    error-policy: continue
    timeout: 5
```

> Notice that we're only setting `- sandbox` for each **pool-type**. Publishing this Sandbox configured like this would ensure that these scripts will not execute on any other VM on this account (Shadow, non-persistent Production, Persistent VM instances).

Testing our Sandbox scripts

You made it this far, you have your scripts and definition.yml file in place on the Sandbox, great! Let's test. Before we do, **changes to the definition.yml are active in real-time**. The Frame Guest Agent (FGA) does a fresh read of the definition.yml at each Session lifecycle hook. This means that if you are in your Sandbox session right now, we can test the disconnect and resume scripts by disconnecting from your Sandbox using the Gear menu at the bottom left corner of the Frame Session. Next, resume the session -- we should see two new log entries in our `/logs/sandbox_bucket.txt` files.

Let's test all four lifecycle hooks.

1. This time, **quit the session** via the Gear menu. Wait for the session to fully close.
2. Once you can, **start a new session**.
3. **Disconnect** from the Gear Menu. This will fire our disconnect or "on-idle" script.
4. **Resume the Session**. This will trigger our resume or "on-active" script.
5. **Close the session**.
6. Finally, **start one more session** to verify our `sandbox_bucket.txt` file contains messages and timestamps in order.

That's it! If you run into any problems, here are a few basic troubleshooting steps before contacting Support:

1. Please ensure that your files are in the correct paths.
2. Please ensure that the file names match in the directory as well as in the definition.yml file.
3. Test your PowerShell scripts manually from the Sandbox.

Cleanup

Once you're done with testing these scripts and scenarios, all you need to do is remove the scripts from the definition.yml, delete the files, or customize them to your liking!

Onboarding Applications via CLI

Onboarding Windows applications to a Sandbox can be automated. If the applications are present on the Sandbox, we have a command-line utility that can be used to onboard a single application, multiple applications, or multiple apps defined in a file.

Introducing ShellHandler! Find and run `ShellHandler.exe` located in `C:\Program Files\Nutanix\Frame\Server\`. Using the command-line arguments listed below, you can onboard applications in a number of ways:

ShellHandler Arguments

<div style={ {width: "150px"} }>Command-line Argument	Description
`onboard`	Instructs the ShellHandler that we're onboarding an application or more. onboard must be the first argument and proceeded next by one of the three following onboard options. **Example:** `ShellHandler.exe onboard -i C:\MyApp.exe -s`
`--item` or `-i`	Onboard a single application item by filepath. **Example:** `ShellHandler.exe onboard --item C:\MyApp.exe -s`
`--list` or `-l`	Onboard multiple applications via a list of application filepaths. **Example:** `ShellHandler.exe onboard --list "C:\MyApp1.exe, C:\MyApp2.exe, C:\MyApp3.exe" -s`
`--file` or `-f`	Onboard multiple applications via a file containing a list of application filepaths. **C:\ExampleAppList.txt:** C:\MyApp1.exe C:\MyApp2.exe C:\Program Files\TestApp\HelloWorld.exe **Example:** `ShellHandler.exe onboard --file C:\ExampleAppList.txt -s`

<div style={ {width: "150px"} }>Command-line Argument	Description
`--silent` or `-s`	If present, application(s) filepaths will be onboarded without any user prompt. **Example:** `ShellHandler.exe onboard -i C:\MyApp.exe --silent`

Revision #6
Created 1 October 2025 04:54:34
Updated 15 January 2026 15:10:49 by Dominik Conrad