XenMobile RestAPI Usage With Powershell - Invoke-RestMethod
Matt Bodholdt
9/11/2017

Automation is a key part of managing systems effectively. The XenMobile Rest API allows engineers to automate tasks that would otherwise require human intervention and manual processes.

Recently I've seen a few publications on how to consume the XenMobile Rest API with Powershell, the most recent being a pair of blogs by Ole Kristian Lona (Blog Part 1 and Blog Part 2). Also the XenMobileShell by Ward van Besien. Like Ole, I prefer to skip the custom Powershell modules and demonstrate consuming the API directly with Invoke-RestMethod. Doing so, I feel, allows the greatest amount of flexibility since there's no dependencies and really isn't as difficult as it might seem!

The overall purpose of this post is to provide further example of how to consume those resources. Along with the post, I've provided a handful of scripts that I have developed and used personally for various items like reporting, removing inactive devices, setting custom device properties, generating enrollments, and exporting specific device properties. It's important to remember that these scripts have worked in scenarios that I have faced and have not been tested across many types of environments. As such, I cannot provide any form of guarantee and if your environment breaks because of these scripts I cannot be held responsible.

Now that the disclaimer is out of the way, let's take a look at some examples…

The first hurdle is the initial authentication:

Using an encrypted password stored to a file. This method is great for running these scripts as a scheduled task. (See Encrpyted_Credential_To_File_CUGC.ps1 for how to store that encrypted string.) Note that the user must have the proper permissions to utilize the Rest API.

```
$username = "XMS_API_USER"
$cpath = Get-Content "\\server\share\encrpytedpw.txt"
$securePwd = $cpath | ConvertTo-SecureString
$creds = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
```

Alternatively, one could use Get-Credential and authenticate in a pop up box using Powershell ISE. You would use one method or the other for authentication, not both.

```
$creds = Get-Credential
```

After the credentials are provided, the body of the log on and log off bodies are formed. Once this is complete the initial credential object is discarded.

```
$authbody = @"
{
"login":"$($creds.GetNetworkCredential().UserName)",
"password":"$($creds.GetNetworkCredential().password)"
}
"@
```

```
$logoutbody = @"
{“login”:”$($creds.GetNetworkCredential().UserName)”}
"@
Clear-Variable creds
```

Next the actual authentication request is made, if it fails an error is thrown and processing stops. If it succeeds the authentication header is formed. The $authheader variable contains the properly formatted auth_token header that will be sent with all subsequent web requests. The other variables are no longer needed and discarded.

```
$authtoken = Invoke-RestMethod -Method Post –URI "https://$($xmsbaseurl)/xenmobile/api/v1/
authentication/login" -ContentType "application/json" -Body $authbody
Clear-Variable authbody
if ($authtoken.auth_token -eq $null) { throw "Authentication Failure" }
$authheader = New-Object "System.Collections.Generic.Dictionary[[String],[String]]"
$authheader.Add("auth_token", "$($authtoken.auth_token)")
Clear-Variable authtoken
```

Log off is handled like this at the end of the script:

```
Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/authentication/logout"
-ContentType "application/json" -Headers $authheader -Body $logoutbody
```

Now that we’re past authentication, let’s look at how to query for devices:

First example returns devices inactive for 30+ days:

```
$devicefilterbody = @"
{
"start": "0",
"limit": "5000",
"sortOrder": "ASC",
"sortColumn": "ID",
"enableCount": "false",
"filterIds": "['device.inactive.time.more.than.30.days']"
}
"@
$devicefilter = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/
device/filter" -ContentType "application/json" -Headers $authheader -Body $devicefilterbody
```

Next, returning devices based on a search parameter such as username or serial number:

```
$search = “username"
$devicefilterbody = @"
{
"start": "0",
"limit": "5000",
"sortOrder": "ASC",
"sortColumn": "ID",
"enableCount": "false",
"search": "$($search)"
```

```
}
"@
$devicefilter = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/
device/filter" -ContentType "application/json" -Headers $authheader -Body $devicefilterbody
```

Next, query for enterprise mode devices:

```
$devicefilterbody = @"
{
"start": "0",
"limit": "10000",
"sortOrder": "ASC",
"sortColumn": "ID",
"enableCount": "false",
"filterIds": "['device.mode.enterprise.managed']"
}
"@
$devicefilter = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/
device/filter" -ContentType "application/json" -Headers $authheader -Body $devicefilterbody
```

Next, unfiltered query returns all devices up to the limit value:

```
$devicefilterbody = @"
{
"start": "0",
"limit": "10000",
"sortOrder": "ASC",
"sortColumn": "ID",
"enableCount": "false"
}
"@
$devicefilter = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/
device/filter" -ContentType "application/json" -Headers $authheader -Body $devicefilterbody
```

The data from a device filter query is retuned under $devicefilter.filteredDevicesDataList. There are a number of interesting items returned just by this query; however, once you have the filtered device list and see what's available you might realize that you need a little more detail like a software inventory, first/last access times, etc. For that, you'll need a different query per device. **$devicedetail (below) is the request by device id that returns more detailed data by device id**. This is the beginning of a foreach device loop when the original list was a device filter query:

```
$devicefilter.filteredDevicesDataList | ForEach {
$devicedetail = Invoke-RestMethod -Method Get –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/
device/$($_.id)" -ContentType "application/json" -Headers $authheader
```

The data from this query is retuned under $devicedetail.device

Examples of interesting data under $devicedetail.device:

To get the version of secure mail,etc installed on that device:

```
($devicedetail.device.softwareInventory | where {$_.packageInfo -like "com.citrix.mail*"}).version
($devicedetail.device.softwareInventory | where {$_.packageInfo -like "com.citrix.browser*"}).version
```

Various timing items, first set origin:

```
$origin = New-Object -Type DateTime -ArgumentList 1970, 1, 1, 0, 0, 0, 0
```

First connection date:

```
($origin.AddMilliseconds($devicedetail.device.firstConnectionDate)).ToLocalTime()
```

Last activity:

```
($origin.AddMilliseconds($devicedetail.device.lastActivity)).ToLocalTime()
```

LastSoftwareInventory:

```
($origin.AddMilliseconds($devicedetail.device.lastSoftwareInventoryTime)).ToLocalTime()
```

Another schwifty thing to do with the XenMobile Rest API is to set custom device properties or modify existing ones. **Here's an example of how to set a custom device property by device id:**

```
$propertyname = "TEST_WITH_THIS"
$propertyvalue = "Yes"

$addpropertybody = @"
{
"name": "$($propertyname)",
"value": "$($propertyvalue)"
}
"@

$add = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/device/
property/$($_.id)" -ContentType "application/json" -Headers $authheader -Body $addpropertybody
```

Remove a device by device id:

```
$removal = Invoke-RestMethod -Method Delete -Uri "https://$($xmsbaseurl)/xenmobile/api/v1/device/$
($_.id)" -ContentType "application/json" -Headers $authheader
```

Switching gears from device properties, lets look at some enrollment functions:

To list all pending enrollments:

```
$enrollmentstatusbody = @"
{
"limit": "2000",
"sortOrder": "ASC",
"sortColumn": "ID",
"enableCount": "false",
"filterIds": "['enrollment.invitationStatus.pending']"
}
"@
```

```
$pendingenrollments = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/enrollment/filter" -ContentType "application/json" -Headers $authheader -Body $enrollmentstatusbody
```

Sending an enrollment invitation (two_factor):

First, define the enrollment notification templates to use:

```
$iosbyodtemplate = "Enrollment PIN BYOD iOS"
$ioscorptemplate = "Enrollment PIN Corporate iOS"
$androidbyodtemplate = "Enrollment PIN BYOD Android"
$androidcorptemplate = "Enrollment PIN Corporate Android"
```

In a foreach loop through the enrollment objects, determine and set ownership, platform, and notification template:

```
if ($_.Platform -like "iOS") { $platform = "iOS"
if ($_.Ownership -like "Corporate") {
$ownership = "CORPORATE"
$pintemplate = "$($ioscorptemplate)" }
else {
$ownership = "BYOD"
$pintemplate = "$($iosbyodtemplate)"
}
}
```

Then form the body of the enrollment and make the request:

```
$enrollmentbody = @"
{
"platform": "$($platform)",
"deviceOwnership": "$($ownership)",
"mode": {
"name": "two_factor"
},
"userName": "$($_.UserName)",
"notificationTemplateCategories":[{
"category": "ENROLLMENT_AGENT",
"notificationTemplate": {
"name": "NONE"
}
},
{
"category": "ENROLLMENT_URL",
"notificationTemplate": {
"name": "Enrollment Invitation"
}
},
{
"category": "ENROLLMENT_PIN",
"notificationTemplate": {
"name": "$($pintemplate)"
```

```
}
},
{
"category": "ENROLLMENT_CONFIRMATION",
"notificationTemplate": {
"name": "Enrollment Confirmation"
}
}],

"notifyNow": true
}
"@

$enrollmentrequest = Invoke-RestMethod -Method Post –URI  "https://$($xmsbaseurl)/xenmobile/api/v1/enrollment" -ContentType "application/json" -Headers $authheader -Body $enrollmentbody
```

Now on to the example scripts!

Each of the following scripts will need to be modified before they work in any environment. Variables for XMS base URL, output directory, mail server, credential path, etc may need to be changed. Any variables that need to be modified will be towards the top of the scripts. Note,

1. **XM_Add_Custom_Property_To_Device_CUGC.ps1**
   - Use this script to add a custom device property to a device or set of devices based on a filter set in the script. The use case is testing certain applications and/or polices on a subset of devices within in a larger delivery group using advanced deployment rules querying a custom property. The script can be used to change an existing device property as well. This script is meant to be run interactively in Powershell ISE.
2. **XM_Remove_Inactive_Devices_CUGC.ps1**
   - Use this script as a scheduled task to remove devices from XMS after they've met a threshold of inactivity (60 days as is). Can be run in Powershell ISE as well.
3. **XM_Wifi_MAC_Export_CUGC.ps1**
   - Need a list of managed devices wifi mac addresses exported that can stay current? Prime example of a way to export data programmatically for ingestion by other systems. The same principals can be applied to any property returned by the filter query with minor modification.
4. **XM_Reporting_CUGC.ps1**
   - This script reports a number of device and user properties into a few CSV's. One shows all active devices and is fairly detailed (including Secure Hub/Mail/Web versions), another shows unique user count and info, a third will report on users who are disabled in AD. When run as a scheduled task can be helpful for tracking system usage, app version compliance, licenses consumption, etc over time. Data example:

SamAccountName	UserName	DepartmeiPlatform	DeviceModel	OSVersion	SecureHubVersion	SecureMailVersion	SecureWebVersion	ShareFileVersion	SecureTasksVersion	Supervised	serialNumber	OwnedBy	managed	mdmKnoxmamKnoxlastAccess	inactivityDays	lastSoftwareInventoryTime	firstConnectionDate
userx	McGee, Snakes	111456 iOS	iPhone	10.3.3	10.6.20	10.6.20.14	10.6.20.9	NA	NA	TRUE	DNQTVALBXXX	Corporate	TRUE	TRUE TRUE	9/5/2017 11:52	0	9/5/2017 11:52	7/31/2017 21:06

5. **XM_Enrollment_Invitation_via_API_CUGC.ps1**
   - This script generates enrollments based on device type and ownership specified in a CSV file. This script is geared for the two_factor enrollment mode but can be customized easily.

6. **XM_Deploy_To_Devices_w_Pending_App_Install_CUGC.ps1**
    - Deploy to devices who have an app in a failed or pending state (select the app from a list of apps enumerated from XMS). This script is mind of messy as it was one of my early dives into the XM Rest API but it does demonstrate how to get apps and app info as well as deploy to a list of devices. This script solved a problem for me during a XenMobile apps upgrade sometime last year.
7. **XM_Anonymous_Devices_CUGC.ps1**
    - Use this script as a scheduled task to remove BYOD devices with "anonymous" (AD disabled) users and log/report corporate owned devices. Can be run in Powershell ISE as well.
8. **Encrypted_Credential_To_File_CUGC.ps1**
    - Use this to save and read encrypted credentials to a file. Necessary if you want to run any of these as a scheduled task.

These scripts are available for download here!

Official documentation around the Rest API is available here: https://docs.citrix.com/en-us/xenmobile/server/rest-apis.html

The best, most detailed document (for 10.7) is here: https://docs.citrix.com/content/dam/docs/en-us/xenmobile/server/downloads/xenmobile-10-7-public-rest-api.pdf