HP iMC 7 customization

Tomas Kubica

Document is provided with no guarantees and features might not be supported by HP. Please consult HP documentation or contact HP representative to understand what customizations are fully supported.
HP iMC customization introduction

This guide focuses on iMC customization and extensibility and is provided as is with no guarantees.

What customization options are covered in this guide?

- Customizing syslog-based alarms – ability to create custom syslog message parser and specific rules to upgrade such event to iMC Alarms defining conditions, reoccurrence etc.
- Importing custom SNMP traps from private MIBs – processing proprietary SNMP traps and providing rules to upgrade those into iMC Alarms
- Adding custom scripts – iMC configuration center has ability to define custom configuration scripts that can be reused when needed
- Extending GUI with custom device actions – right click on device in topology or action menu in devices list is extensible with custom actions on client side such as opening web management application or executing client application and passing arguments to it
- Adding custom vendor and device – when device does have standard SNMP MIB, but is not recognized by iMC you can create custom device vendor and type
- Creating custom device adapters for Configuration Center – iMC configuration center has built in support for configuration and OS backup of HP, Cisco and some other devices, but you can create your own adapters to enable such functionality with any device
- Creating custom batch device and interface operations – iMC ships with set of batch operations (or you might call it wizards) and you can create your own batch operations with extending GUI and providing scripting implementation so you can have your own wizards for batch configuration of devices or interfaces
- Cloud-based SMS gateway – iMC as well as UAM support sending SMS messages (forwarding Alarms, providing guest passwords etc.). There is support for hardware modems, but you can extend iMC with cloud-based SMS service implementation so you can use provider like Textmagic without need to use hardware gateway.

Tomas Kubica

kubica@hp.com
Contents

HP iMC customization introduction ................................................................. 2
Customizing syslog-based alarms ...................................................................... 4
Importing custom SNMP traps from private MIBs .............................................. 10
Adding custom scripts ..................................................................................... 15
Extending GUI with custom device actions ...................................................... 20
Adding custom vendor and device ................................................................. 27
Creating custom device adapters for Configuration Center ............................ 31
Creating custom batch device and interface operations ................................. 42
Creating and using custom cloud-based SMS gateway ................................... 56
Customizing syslog-based alarms

iMC comes with syslog server capability and set of predefined templates and rules to upgrade syslog messages into alarms. You can send syslog messages from other systems, collect them in iMC and create custom parsing and rules to upgrade into iMC Alarms. Please note that iMC is not designed for centralized aggregation and store of all logs from IT environment, products like HP ArcSight Logger are positioned to capture tens of gigabytes of log data every day.

You can start by simply point syslog capable device to iMC IP address and looking into Alarm, Syslog Management, Browse Syslog.

By default iMC will accept syslog messages from devices not added into iMC as well as messages that do not match any of existing syslog templates. You can change this behavior in Filtering Rule section.

Please note that it is not necessary to add device into iMC management in order to receive syslog message from it, but when you upgrade this message to Alarms it's indicated source IP will be 127.0.0.1 (NMS). To have iMC fill in proper source IP of the device when upgrading to Alarm make sure device has been added to iMC.

For now let’s keep it on defaults. For next example we have configured Ubuntu-based Linux server to send messages about authentication (ssh logins). To do so simply install sysklogd package and edit /etc/syslog.conf to include IP address of iMC server:

```
auth,authpriv.* @10.10.10.175
```

Generate some login failures and check iMC Browse Syslog page.
Also note that no Alarm is generated. You can configure iMC to parse messages, extract information and define conditions under which Alarm message is generated. We will create template for Linux login failure. First let’s examine raw syslog message.

We will create syslog template to match this message and extract information about user and source client IP. To do so click on Syslog Template and click Add.

We will use * signs and $(variable_name) to parse specific words:
*sshd*Failed password*user $(user) from $(from) port $(facility)

Create new template and click OK.
Your template is listed:

Now we will create new rule to upgrade this syslog message into Alarms. Click on Syslog to Alarm and click Add.

In first section enter name of this rule and make sure status is Enabled. You can add Description and also select Custom View that will limit this rule to affect only specific subset of your devices.
In next section you can filter on syslog type and level. Also you might want to generate Alarm only if the same syslog message has occurred multiple times during certain time window. Statistics setting define how Alarm engine aggregates (iMC might stop processing too high rate of alarms and instead display message that it has occurred 100 times or so) – in our case we want to keep this per device. Then you specify severity level of generated alarm and also its description. If you like you may simply use %Syslog% string, which will copy exact content of syslog message into Alarm description. In our case we will want to have our own wording and we will use some of variables we have extracted. Then click Select to choose our Syslog Template. Please note that content of this template will be copied into your syslog to Alarm rule – so changing template itself afterward will not be reflected in your rule unless you edit this rule and reselect again.
If you are creating rule that needs to have multiple matches within time window you might need to specify what changes in syslog text are taken as new message or as the same one. For example if you want to get alarm when user “tomas” has failed login into single server three times within a minute your will keep parameter user checked. If on contrary you want to get alarm anytime there are 3 login failures within a minute regardless of username used you uncheck this parameter.

Last section is used if you want iMC to have capability to automatically recover your event. In our case we are not going to use it, so administrator needs to click Recover button himself. Typical example of this feature is when reporting on something (like interface) going down and up. You will generate alarm when receiving message with down event and when you receive different message with up event as configured in this screen iMC will automatically recover that down alarm.
We are ready to go so click OK.

Syslog to Alarm rules are cached in memory to speed up processing. Therefore if you are not creating, but modifying your rules you need to restart imcfaultdm process in your Deployment monitoring agent.

You can now generate login failure event and check your Alarms on Alarm Browse, All Alarms.
Importing custom SNMP traps from private MIBs

iMC ships with a lot of SNMP trap definitions including a lot of standards based ones, but also has capability to import private MIBs to decode vendor specific messages. In this example we will use Fortinet device. Get private MIBs from vendor website (or use attached files for Fortinet) and copy files and INET-ADDRESS mib file to C:\Program Files\iMC\client\TrapMIB (/opt/iMC/client/TrapMIB/ in Linux) on iMC server.

- FORTINET-CORE-MIB.mib
- INET-ADDRESS-MIB.mib

Select Alarms, Trap Management, Trap Definition
Select Import trap definition from MIB file

Click FORTINET-CORE-MIB

Select all traps and click Import
Generate some event such as IP address change and Browse Traps

Note that iMC has not upgraded SNMP trap to Alarm since there is no predefined upgrade policy for imported traps. Click on Trap to Alarm
Click Add

Select Trap Type

Click Select
Select all Fortinet traps and click OK

Add some rule name and click OK.
Adding custom scripts

iMC Configuration Center is capable of creating, storing and executing custom CLI scripts on as needed or scheduled basis. You can use variables in your script which will be filled during scheduling deployment task – so for example you can create script to setup NTP service and instead of hardcoding IP address use variable so script be reused even when IP changes. Also metadata are stored with script, especially supported devices. In multi-vendor environment you might have syntax differences between platforms so you will associate only certain categories of devices with your script.

Open Service, Configuration Center, Configuration Templates.

Click Add CLI script
Enter name of your script and click Select to list device categories supported – in our case we will use HP VSR platforms.

We will use Basic configuration mode which is enough for our purpose. Script will simply enter system-view context, enable NTP service and configure server IP address. Instead of hardcoding IP in script we will use custom variable in syntax `${variable_name}`, which will be filled later during deployment process.

Click OK to save.

There are more ways to initiate setup of deployment task. We will click on … sign and select Deploy.
On next screen click on Select device and choose devices you want.

Choose your deployment strategy – you might want to backup running configuration before and after operation or decide to save running to startup after execution. When you are ready click Next.

On next screen iMC ask for value of variables in the script. Enter IP address of your NTP server. In our case we will use the same IP for all devices so check Use the same parameters for following devices and click Next.
On next screen you will decide when you want script to be executed. Options include once on repeat in cycle and execute immediately or schedule for later time. When you are ready click next.

Last step is overview of task. If you are OK with it click Finish.
Your script is being executed.

After some time you will get results.
We have successfully created and executed custom script. You can use this functionality to do even more complex changes to network configurations or execute scripts at certain times. If you need more interactive jobs such as extracting information from configuration files or even running state (display/show commands), check compliance with your best practices or security regulations you might use custom Compliance Center policies.

Extending GUI with custom device actions

Administrators might use other management related tools. Those are usually either web-based applications or local executables. iMC support defining custom device actions that are accessible in GUI environment such as on device list or topology and enable envoking web or executable application while passing arguments to it such as device IP, login username/password, SNMP string etc.

We will now explain how this can be achieved and then show two examples. Everything you need to do is create XLM file with proper structure and optionally upload icon files and application installer.

First you need to create file in folder /iMC/client/web/apps/imc/WEB-INF/. Name of the file must be something-devToolsConfig.xml where “something” is whatever you want.
You will start with this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tools-config xmlns="http://www.h3c.com/device-tools-config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.h3c.com/device-tools-config dev-tools-config_1_0.xsd">
  <description>Zenmap network scan tool configuration</description>
  <resource-bundle>com.h3c.imc.plat.LocalStrings</resource-bundle>
</tools-config>
```

You don’t need to change any of that, perhaps you will add your own description. Next we will open config and get id to it – you can use whatever you want as long it is unique.

```
<config id="zenmap">
```

Next section describe what devices will have your new action displayed. Here is syntax:

```xml
<conditions>
  <sys-oid use-when-no-sysoid="false">
    <value>1.2.3.4.5</value>
    <wildcard-matcher>*</wildcard-matcher>
  </sys-oid>
</conditions>
```

Attribute use-when-no-sysoid defines whether action will be displayed for devices that have no sysoid, for example desktops. You can enter <value> attributes with exact match on sysoid or use <wildcard-matcher> attribute that supports wildcarding, so you can for example define particular vendor products (1.2.3.*).

Next is action section that can contain only one action.

```
<action>
```

There are two types of actions – one runs executable on client device and other one opens web application. In both cases you can pass arguments such as device IP. Let’s first examine action to execute application on client.

```xml
<client-tool>
  <app.name>Zenmap scan</app.name>
  <img.url>res/zenmap.png</img.url>
  <pref.defpath>C:\Program Files (x86)\Nmap\zenmap.exe</pref.defpath>
  <pref.pathkey>com.h3c.imc.plat.exec.zenmap</pref.pathkey>
  <download.url>nmap-6.46-setup.exe</download.url>
  <app.arg>-t ${DEVICE_IP}</app.arg>
</client-tool>
```

Attribute <app.name> define string that is going to be visible in actions menu. <img.url> defines path to icon used. Note that position is relative and you need to upload your icon file to iMC server for all skins:

iMC/client/web/apps/imc/resources/images/blue/res/zenmap.png
iMC/client/web/apps/imc/resources/images/lightgray/res/zenmap.png
iMC/client/web/apps/imc/resources/images/classic/res/zenmap.png
iMC/client/web/apps/imc/resources/images/white/res/zenmap.png

Attribute <pref.defpath> defines location on client (PC where you open iMC GUI) of application you want to execute. If client will not find file in location specified user will be prompted to choose his own path. <pref.pathkey> is just programmatic reference. Only thing you need to make sure that whatever comes after exec. Needs to be unique.
In case client does not have that application installed he can download it right away from iMC server. In attribute `<download.url>` specify name if installer and place that file in `iMC/client/web/apps/imc/plat/exec/

Last attribute is `<app.arg>` where you can set what parameters will be used when executing client application. Note that you can use iMC variables to fill in information such as IP address of the device and many more. Here is list of options:

- `${DEVICE_ID}`: ID of the device.
- `${DEVICE_IP}`: IP address of the device.
- `${SYMBOL_ID}`: Symbol ID of the device.
- `${SYS_OID}`: Sys OID of the device.
- `${SNMP_VERSION}`: SNMP version of the device.
- `${READ_COMMUNITY}`: Read-only community name.
- `${SET_COMMUNITY}`: Write community name.
- `${TIMEOUT}`: Timeout.
- `${RETRY_COUNT}`: Retries.
- `${SNMP_USERNAME}`: SNMP username.
- `${SECURITY_LEVEL}`: Security level.
- `${AUTH_PASSWORD}`: Authentication password.
- `${AUTH_SCHEMA}`: Authentication scheme.
- `${PRIV_PASSWORD}`: Encryption password.
- `${PRIV_SCHEMA}`: Encryption scheme.
- `${TELNET_AUTH_TYPE}`: Telnet authentication type.
- `${TELNET_USER_NAME}`: Telnet username.
- `${TELNET_PASSWORD}`: Telnet password.
- `${TELNET_SUPER_PASSWORD}`: Telnet super password.
- `${TELNET_PORT}`: Telnet port number.
- `${OPERATOR_NAME}`: Operator username.
- `${OPERATOR_ROLE}`: Operator role.
- `${OPERATOR_LOGIN_ADDR}`: Operator login address.
- `${LOCALE}`: Current language information.

Let’s now examine URL type of action.

```xml
<url>
    <url.name>Download manuals</url.name>
    <param name="ProductNumber" value="JG811AAE"/>
</url>
```

Attribute `<url.name>` is string that will be visible as name of your action. `<url.value>` specifies icon in the same structure and path as client-tool action. `<url.value>` is URL you want to launch and `<param>` encodes parameters into URL string. You can use built in variables described before to fill in for example IP address of the device.

When troubleshooting your custom device actions check `imcforeground.log` in you iMC/client/log folder. This is example of typo in your XML:

```
org.xml.sax.SAXParseException; systemId: file:/opt/iMC/client/web/apps/imc/WEB-INF/zenmap-devToolsConfig.xml; lineNumber: 21; columnNumber: 13; The end-tag for element type "config" must end with a '>' delimiter.
```
We will now try both types of actions in practice. First we will enhance iMC GUI to add action to initiate Zenmap scan. Zenmap is GUI-based version of nmap port scanner allowing to examine open ports on target device for security purposes. We will want this option to be available for all devices including those that do not have sys-oid such as desktops. This is content of zenmap-devToolsConfig.xml file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <description>Zenmap network scan tool configuration</description>
  <resource-bundle>com.h3c.imc.plat.LocalStrings</resource-bundle>
  <config id="zenmap">
    <conditions>
      <sys-oid>
        <wildcard-matcher>*</wildcard-matcher>
      </sys-oid>
    </conditions>
    <action>
      <client-tool>
        <app.name>Zenmap scan</app.name>
        <img.url>res/zenmap.png</img.url>
        <pref.defpath>C:\Program Files (x86)\Nmap\zenmap.exe</pref.defpath>
        <pref.pathkey>com.h3c.imc.plat.exec.zenmap</pref.pathkey>
        <download.url>nmap-6.46-setup.exe</download.url>
        <app.arg>-t ${DEVICE_IP}</app.arg>
      </client-tool>
    </action>
  </config>
</tools-config>
```

Copy this file to iMC/client/web/apps/imc/WEB-INF. Also copy icon file to following locations on your server:

```
/opt/iMC/client/web/apps/imc/resources/images/blue/res/icon_dev_ssh_16x16.png
/opt/iMC/client/web/apps/imc/resources/images/lightgray/res/icon_dev_ssh_16x16.png
/opt/iMC/client/web/apps/imc/resources/images/classic/res/icon_dev_ssh_16x16.png
/opt/iMC/client/web/apps/imc/resources/images/white/res/icon_dev_ssh_16x16.png
```

Lastly download installer from [www.nmap.org](http://www.nmap.org) and place it to

```
iMC/client/web/apps/imc/plat/exec/
```

For URL action we will as example create Download manuals action for VSR1001. This is content of vsrmanual-devToolsConfig.xml you need to copy into WEB-INF folder:

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <description>VSR manuals</description>
</tools-config>
```
In order for new action to get activate you need to restart jserver process via Deployment agent (or restart full iMC).

We are ready to examine results now. Check your devices list and click on *** icon. You can see new actions are visible now:
Also you can open your device page in iMC and check actions on right side.

Lastly open network topology, right click device and see Tools menu:
Click on Zenmap scan and note that execution of client application is done via Java applet so you might be presented with some security warnings if you have not configured your client accordingly. Suppose client does not have zenmap installed on his system:

Note that you might select different location in case zenmap is installed in different directory or directly download installer from iMC server. If you have application installed OK:

Your zenmap is opened and note that Target IP is prepopulate based on iMC device you have initiated action on.
What if you click on Download manual action? iMC will open web page.

As a summary iMC allows you to define custom actions that are integrated with iMC GUI so you can enhance your experience and integrate iMC with other client or web-based applications you are using in your environment.

Adding custom vendor and device
iMC comes with built-in support for a lot of vendors and devices, but even if you want to manage something that is not covered you can enhance iMC to support it. We will now do initial step, which is creating custom device vendor and type. Suppose you have added device into iMC (Mikrotik in our example), SNMP communication is established, but device is reported as Unknown SNMP Product.

Go to System, Resource Management, Device Vendor.
Let’s first upload Mikrotik icon – click on Upload Vendor Icon.

Choose gif file 16x16 pixels, specify name string and clock upload.

Success Icon “mikrotik” successfully uploaded.

Go back to vendor list and click , enter vendor information and select your new icon.
Now we will create new series – click Device Series.

Click Add, define new series and click OK.

Lastly we are going to define model. Click Device Model.
Click Add and fill in required information. If you do not know sysoid directly you may click Query device and select from list (if device is already managed by IMC showing it as Unknown SNMP Product). You can choose any of existing categories, but if you click Device Category you can define new one and upload specific icon.

Go back to your device and you should see new name is applied.
Creating custom device adapters for Configuration Center

iMC ships with built-in support to save and restore configurations and device firmware for HP devices (both Comware and Provision) as well as majority of Cisco switches and routers. It also includes limited set of driver for Dell, Juniper, Aruba, F5, Extreme, Nortel and Huawei. You can either extend existing ones (to support more models) or even create new drivers for other devices.

When creating your own adapters you will use exactly the same system as existing ones. Best place to start learning is to examine existing adapters developed by HP. You can find them in folder:

/IMC/server/conf/adapters/ICC

In this guide we will first walk throw adapter definitions using existing adapter and later will create our own to enhance iMC capabilities to be able to backup configuration from Mikrotik device.

Main adapter folder has subfolders named by device vendor. If you are creating your drivers make sure folder exactly match Device Vendor in iMC (refer to chapter Adding custom vendor and device in this guide).

Root folder of your adapter contain file adapter-index.xml and subfolders with actual adapters. This file has simple structure:

```xml
<?xml version="1.0"?>
<!--sysoid adapt adapter-->
<adapters>
  <type name="SNMP">
    <adapter name="H3CV7Snmp">
      <description>for h3c comware v7 device</description>
      <sysoid>1.3.6.1.4.1.25506.*</sysoid>
    </adapter>
  </type>

  <type name="CLI">
    <adapter name="H3CCommon">
      <description>H3C series switch and router...</description>
      <sysoid>1.3.6.1.4.1.25506.*</sysoid>
    </adapter>
  </type>

  <type name="CLI">
    <adapter name="H3CCommon">
      <description>H3C series switch and router...</description>
      <sysoid>1.3.6.1.4.1.25506.1.65</sysoid>
      <sysoid>1.3.6.1.4.1.25506.1.31</sysoid>
    </adapter>
  </type>

  ...
</adapters>
```

This is XML structure that in our case define two adapters. One of type SNMP (SNMP protocol is used for actions) and one o type CLI (leverages TCL scripting). Adapter name needs to be equal to...
name of subfolder in your adapter root. Then you specify sysoid and you can use wildcards. When iMC is looking for a driver it first check vendor name and then whether OID matches something in adapter index file. If yes proper adapter is used. Let’s now examine actual adapters. SNMP type of driver is not documented and not supported for custom drivers – nevertheless you may look into H3CV7Snmp, learn its structure and use it. We will focus on CLI type adapters in this guide which is also covered in iMC documentation. Open H3CCommon folder.

First important file is adapter.xml:

```xml
<adapter name="H3CCommon">
  <description>H3C series switch and router..."</description>
  <version>1.0.0</version>
  <services>
    <service name="CLICommon">
      <item type="common">H3C_Common_CLI.xml</item>
    </service>
    <service name="CleanupParser">
      <item type="parser_definition">H3C_Cleanup_Parser.xml</item>
      <item type="perl_script">H3C_Cleanup_Parser_Script.pl</item>
    </service>
    <service name="ConfigBackup">
      <item type="builder_definition">H3C_Config_Backup_Builder.xml</item>
      <item type="tcl_script">H3C_Config_Backup_Builder_Script.xml</item>
    </service>
    <service name="ConfigDeploy">
      <item type="builder_definition">H3C_Config_Deploy_Builder.xml</item>
      <item type="tcl_script">H3C_Config_Deploy_Builder_Script.xml</item>
    </service>
    <service name="ImageDeploy">
      <item type="builder_definition">H3C_Image_Deploy_Builder.xml</item>
      <item type="tcl_script">H3C_Image_Deploy_Builder_Script.xml</item>
    </service>
  </services>
</adapter>
```

This XML structure define various services and their definitions. CLICommon is important one that sets foundation for other services. As we will see later this defines how to authenticate to CLI, what are various levels (enable, exec, ...) and so on. Then we have services that deal with configurations and firmware (in this guide we will focus on ConfigBackup and ConfigDeploy). This services have two items – builder definition (where actions are defined) and tcl script xml (defines in what TCL files actions are implemented). Another service is CleanupParser which defines set of Perl scripts used to do some cleanup modifications on data gather by other services (will explain later). Note that with your drivers you don’t have to implement everything (especially OS management can get very complicated do to need to check free space on flash, check for name conflicts etc.) – you may start with common definitions and ConfigBackup.

Let’s start with H3C_Common_CLI.xml
This file defines basic modes and functions. Initialize is mode that is executed after iMC connects to device. If you open contents of initialize.tcl you will see a lot of variable definitions such as strings used by expect to check on prompts. Another very important mode is exec and XLM specifies TCL to enter this mode and exit from it. This is basically what happens after telnet or CLI is started and device ask for password. You need to deal with various situations here (press Enter to continue, no login, telnet disabled, super password, SSH key authentication, ...). Open enter_exec.tcl and see how it is built. Do not panic! This is professional file that needs to deal with tremendous set of hardware types and all situations therefore it looks very complicated. But basically it creates a loop and wait until $exec_prompt is achieved (such as Switch>). For example when expect gets password prompt script will send it (by using iMC variable) send "$password". Everything else is a lot of other situations, scenarios and connection types.

Check enter_enable.tcl which is way simpler. This is to enter configuration context in Comware called system-view. Script do send "system-view\r" and just deal with some error situations and if $enable_prompt (as defined in initialize.tcl) is achieved it is all done.

Have a look now at ConfigBackup service and start with H3C_Config_Backup_Builder.xml – here is one part of this file:

```xml
<action name="backup_running_config">
  <step>
    <command name="backup_running_config_tftp" description="..."/>
  </command>
  <command name="backup_running_config_ftp" description="..."/>
  <command name="backup_running_config_sftp" description="..."/>
  <step>
    <command name="save_running_config" description="..."/>
  </command>
  <step>
    <command name="backup_running_config_sftp" description="..."/>
  </command>
</action>
```
As this adapter is truly universal it features many ways to get your running configuration. Commands are defined to get backup via TFTP, FTP, SFTP or even by capturing CLI output actually in two versions (screen scraping – which is least efficient and therefore is more like “if everything else fails” method). Note that some of commands also refer to parser function as defined in CleanupParser structures (that leads to function in Perl script). Typically with screen scraping of configuration device can get log message in a time configuration is being displayed resulting in this log message to appear in captured result. If this happen we need to clean this up. On some vendor devices even if you get configuration via file transfer it includes some comments with time stamp (usually starting with #). That would cause iMC to treat every backup as changed configuration even it is the same (just timestamp has changed), which would lead to too many versions to manage if you autobackup every day. Parser function might get rid of those comments.

Now we can look into H3C_Config_Backup_Builder_Script.xml, where there are references to actual implementations of actions. Here is subset of this file:

```xml
<command name="backup_running_config_tftp" method="TFTP">
  <error>...</error>
  <require-mode>exec</require-mode>
  <script>
    backup_running_config_tftp.tcl
  </script>
</command>
```

As you can see this file define how commands within actions are implemented by giving name of TCL script that will be executed. Also note require-mode attribute that refers back to CommonCLI modes. Let’s now walk throw some of those TCL files and get a little bit of understanding of what is going on with various methods.

We can start with `backup_running_config_tftp.tcl`. Most interesting part is this:

```tcl
send "save iccrunning.cfg\r"
set pos_ [string first "::" $TFTPServer]
if {$pos_ != -1} {
  set my_cmd "tftp ipv6 $TFTPServer put iccrunning.cfg $TFTPFile"
} else {
  set my_cmd "tftp $TFTPServer put iccrunning.cfg $TFTPFile"
}
if { $VpnName != "" } {
  append my_cmd " vpn-instance $VpnName"
}
send "$my_cmd\r"
```

As you can see script first store running configuration to file iccrunning.cfg and little later prepare command to push this file to TFTP server which is running on iMC and it’s ip address is available via $TFTPServer variable. Script parse this IP to find out whether to use ipv4 or ipv6 version of command and then create basic string (remote file name must be $TFTPFile which generated file name that iMC expects to arrive to TFTP server) and eventually add VRF information and send it.
What about backup_running_config_cli.tcl? Here is major part of script:

```tcl
send "display current-configuration\r"
set loop true
while {$loop == "true") {
    expect {
        -re "$more_prompt" {
            send " 
        }
        -re $error_pattern {
            set error_message $expect_out(1,string)
            expect -re $exec_prompt
            set ERROR_RESULT true
            set ERROR_MESSAGE "Device error: $error_message"
            set loop false
        }
        -re $exec_prompt {
            # Done
            set loop false
        }
    }
}

Command definition contained result attribute so iMC knows we need to capture output of this script. Script initiate display current-configuration command, sets a loop and wait till we get $exec_prompt again (such as Switch>). If paging is enabled we might get $more_prompt so we press a key if that happens. Next phase is to clean this up – based on builder definition we will call cleanupConfiguration function in H3C_Cleanup_Parser_Script.pl Perl script.

```
What is happening? Driver is removing More prompts, also “display current-configuration” command (which should not be part of configuration file), remove potential syslog messages and do some new line manipulations so results are exactly the same as when getting config via file transfer.

backup_running_config_ftp.tcl and ftp_trans_file.tcl are implementing FTP and even it looks complicated it is due a lot of error checks. In fact this is very similar to TFTP – save configuration to file name and push it to iMC via CLI command.

I am going to stop here with adapter description. Please consult iMC documentation for some more details and the best resource is to learn from existing adapters. You can implement your adapters to work with OS management (it is a bit more complicated if you want to capture all difficulties such as check free space, remove older files, check dependencies ... but if you stick with basics you can make it easily). Let’s move on to create our own primitive adapter for device not supported in iMC – in our case virtual Mikrotik device.

During development and testing there are two important things to note. For performance reasons ICC (Configuration Center) adapters are cached in memory. If you make some changes you need to restart process imccfgbakdm – this forces iMC to cache new version of your adapter. Also you might face some error situations – for troubleshooting look into following log file (there is txt file for current day and zip archives for past days):

```
/IMC/server/conf/log/imccfgbakdm.2014-07-07.txt
```

For our example make sure Mikrotik virtual devices has been added to iMC and new Device Vendor is created (as explained earlier in this guide).

First create new vendor folder:

```
/IMC/server/conf/adapters/ICC/Mikrotik
```

Now create adapter-index.xml in this folder. We will create definition for single adapter that will be referencing single device model. You can use multiple sysoid statements as well as wildcards or create more adapters if CLI syntax differ between models.

```xml
<?xml version="1.0"?>
<!--sysoid adapt adapter-->
<adapters>
  <type name="CLI">
    <adapter name="RouterOS6">
      <description>Mikrotik demo adapter for RouterOS 6</description>
      <sysoid>1.3.6.1.4.1.14988.1</sysoid>
    </adapter>
  </type>
</adapters>
```

Next step is to create subfolder that exactly matches adapter name, in our case:

```
/IMC/server/conf/adapters/ICC/Mikrotik/RouterOS6
```

In this folder create adapter.xml file. We will start simple so only common and configuration backup service will be defined:

```xml
<adapter name="RouterOS6">
  <version>1.0.0</version>
  <services>
    <service name="CLICommon">
```

Create Mikrotik_Common_CLI.xml file and for simplicity let’s define just two modes – initialize and exec:

```xml
<?xml version="1.0"?>
<common>
  <mode name="initialize" method="cli">
    <require-mode>connect</require-mode>
    <enter>
      initialize.tcl
    </enter>
  </mode>
  <mode name="exec" method="cli">
    <error>Failed to get to exec mode. </error>
    <require-mode>initialize</require-mode>
    <enter>
      enter_exec.tcl
    </enter>
    <exit>
      exit_exec.tcl
    </exit>
  </mode>
</common>
```

First let’s create file initialize.tcl and declare some basic variables:

```tcl
set standard_timeout 10
set long_timeout 120
set very_long_timeout 1800
set username_prompt ogin:
set password_prompt password:
set login_as 
set info_prompt "continue!"
set exec_prompt "] >"
set timeout $standard_timeout
set ERROR_RESULT false
set ERROR_MESSAGE ""
```

Now we will define script to enter exec mode in file enter_exec.tcl:

```tcl
set IGNORE_DELAY true
set loop true
```
set timeout $standard_timeout

while {$loop == "true"} {
    expect {
        $password_prompt {
            send "$password"
            sleep 1
            send "\r"
        } $username_prompt {
            send "$username"
            sleep 1
            send "\r"
        } $login_as {
            send "$username"
            sleep 1
            send "\r"
        } $info_prompt {
            send "\r"
        } "Store key in cache" {
            send "y\r"
            set loop false
        } -re $exec_prompt {
            set loop false
        } "incorrect username or password" {
            set ERROR_MESSAGE "Authentication failed"
            set ERROR_RESULT true
            return
        } "Access denied" {
            set ERROR_MESSAGE "Authentication failed"
            set ERROR_RESULT true
            return
        }
    }
}

set IGNORE_DELAY false

And for exit we create file exit_exec.tcl:

send "quit\r"

By now we have prepared basic structures for real work. Let’s create Mikrotik_Config_backup_Builder.xml. We are defining single command (backup configuration) and since we are using CLI screen-capturing method we also need to specify cleanup parser:

<?xml version="1.0"?>
<definition>
    <action name="backup_startup_config">
        <step>
            <command name="backup_startup_config_cli" description="Backup startup via CLI">
                <result name="startup_configuration">
                    <parser parserName="CleanupParser" parserRet="configuration" parserScript="cleanupConfiguration"/>
                </result>
            </command>
        </step>
    </action>
</definition>

Implementation of backup action is in Mikrotik_Config_backup_Builder_Script.xml, so let’s create it:

<?xml version="1.0"?>
And here is actual TCL script `backup_startup_config_cli.tcl`. After we send export command (that list configuration) we expect to get # (so Mikrotik started to print configuration already) and wait for end when we will be back in exec prompt:

```tcl
set timeout $standard_timeout
sleep 1
expect -re $exec_prompt { } send "export\r"
expect -re #
expect -re $exec_prompt { }
```

Lastly we will define our parser in `Mikrotik_Cleanup_Parser.xml`:

```xml
<?xml version="1.0"?>
<parser>
  <perlfile>Mikrotik_Cleanup_Parser_Script.pl</perlfile>
  <script name="cleanupConfiguration">
    <callfunction name="cleanupConfiguration">
      <parameterspec name="0" type="data_string"/>
      <result type="single">configuration</result>
    </callfunction>
  </script>
</parser>
```

And here is Perl script – at first we will not do anything here since we do not know yet what is going to be needed. `Mikrotik_Cleanup_Parser_Script.pl`:

```perl
#!/usr/local/bin/perl
sub cleanupConfiguration
{
  my($config) = @_;
  return $config;
}
```

We are ready to test our driver now. Make sure Mikrotik is managed by iMC and proper Device Vendor has been created. Then go to Configuration Center, Configuration Center and open Configuration Management for your Mikrotik device.
Click backup and see process. For troubleshooting you might want to tail (in Linux install or similar method in Windows) log file (find proper one based on actual date):

```bash
```

Here is GUI output:

We have not implemented backup of Running configuration (as Mikrotik does not have such concept), but we successfully backed up Startup configuration. Go back and check results file:

```plaintext
Device configuration files manually backed up. Total: 1, backed up: 0, failures: 1.
```

```plaintext
Device Name  Device Model  File Type  Time          Result                    Configuration File Name                      Details
10.10.10.44(10.10.10.44)  Mikrotik virtual device  Startup  2014-07-14 12:15:02  ✓ Backup configuration files. Total: 1, startup files: 1, running files: 0. 10.10.10.44_startup_20140714115453.cfg
10.10.10.44(10.10.10.44)  Mikrotik virtual device  Running  2014-07-14 12:14:30  ✓ Starting to backup configuration files. --
```
There are couple of problems with this. First – we have captured actual call of export command as well as exec prompt on end. Also configuration starts with timestamp, which is problem, because iMC will think configuration has changed everytime you initiate backup. This is job for cleanup parser. We will do three things here: filter syslog lines if that happens (if Mikrotik generates event during export command it gets printed on screen), filter timestamp and filter exec prompt itself. Let’s modify our Mikrotik_Cleanup_Parser_Script.pl:

```
#!/usr/local/bin/perl

sub cleanupConfiguration
{
  my($config) = 8_;

  # CLI sometimes leaks in some syslog messages.. remove them _first_
  # $config =~ s/(^|\n)%.*//g;
  $start = index($config, "# software id = ");
  if ($start == -1)
  {
    $start = 0;
  }

  $cleanConfig = substr($config, $start);

  # Get rid of the exec prompt at the end, and any blank lines
  $cleanConfig =~ s/\ \.*//n/;
  $cleanConfig =~ s/\n+$/;

  # Add a trailing newline to match file transfer results
  $cleanConfig = $cleanConfig . "\n";

  return $cleanConfig;
}

Restart imccfgbakdm process and initiate new backup and check results:
Here we go. In this example we have used CLI capture method, but with more typical networking devices you should rather use file transfers (TFTP, FTP, SFTP, SCP) as it is much faster and cleaner with less overhead. Also you can extend driver to support restore of configuration. You might also want to write support for OS backup and restore, but make sure some corner cases are handled (such as not enough space on flash etc.).

Here is full adapter:

Mikrotik ICC driver.zip

Creating custom batch device and interface operations
iMC comes with predefined set of batch operations under Resource, Batch Operation:
Last two items are used for more generic configuration wizards and there are some predefined operations there. Nevertheless you can extend iMC with your own custom operations that include GUI components and actual implementation. Compared to CLI scripts this has many advantages:

- GUI wizard to enter parameters of your choice with variety of GUI options such as check box, check box group, input box, combo box or even calendar tool
- GUI components can be populated statically or even dynamically load values (via TCL scripts directly from device or via iMC REST API) – for example your wizard can list currently configured items such as VLANs, ACLs, ...
- Scripting implementation allows for very complex scenarios including checks, parsing and editing existing configurations (for example remove configured items with undo/no statements)
- Implementation can leverage iMC REST APIs for your tasks
- Operations have defined dependencies on devices capabilities and GUI is parsing those accordingly, so you can only select devices that your operation actually supports

First let’s create GUI part. This is managed by creating this folder:

/iMC/client/web/apps/imc/gencfg/register/custom

If you want to see how iMC implements built-in wizards (for reference) you can check

/iMC/client/web/apps/imc/gencfg/register/gencfg

There are wizard categories defined for “Device configuration guide” and “Interface configuration guide” under Batch Operation. Those are listed in

/iMC/client/web/apps/imc/gencfg/register/categoryDef.xml:
It is expected you will use predefined Custom category. You can create your own, but be warned that this particular file will be replaced during iMC upgrades (so you need to modify it again after upgrade). In our example we will not create new category, we will use “custom”.

Let’s go back to custom directory we have created as a first step. We will need to implement three files here. First create custom_devCapabilities.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DevCapabilities>
  <Capability name="ComwareNTP">
    <Sysoid>1.3.6.1.4.1.25506</Sysoid>
  </Capability>
</DevCapabilities>
```

What is happening here? Our custom batch operations will have specific needs that not all devices iMC manage might support. For example we will be implementing custom operation for configuring NTP, but for simplicity we will create adapters only for Comware devices. Even you batch operation can call various adapters for multiple vendors we are not implementing those adapters in our example. More over some devices will even not support NTP at all. Definition of “capability” defines what devices can batch operation run on. When selecting your wizard user will not be able to select devices that does not conform to defined capabilities (GUI will filter unsupported ones).

Next step is to create custom_operations.xml file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Operations>
  <Operation id="ComwareNTP" label="Configure NTP" categoryId="custom"
    devCapability="ComwareNTP" actionId="ComwareNTPaction" actionType="2"
    operLevel="1">
    <Asn1Action id="ComwareNTPaction" serviceName="ComwareNTP"
      actionName="ComwareNTPaction" operType="1" operMode="2" execMode="1">
      <Parameter name="Authentication" label="Enable MD5 authentication"
        dataType="boolean" required="true">
```
Please refer to iMC documentation for more details – chapter Extending and customizing functions to support third-party devices, Extending a customized function.

We will now quickly introduce to structure of this file. We are defining Operation here. There is id, label (how operation is visible in GUI), devCapability (as explained previously) and some other IDs. Very important is operLevel – if you set this to 1 this wizard will be based on devices. If you set 2, GUI will select interfaces (so your adapter will be called for selected ports which might span multiple devices). Next critical item is actionType. We are using 2 here, which indicates CLI operation (adapter is called as we will explain later), but you can also use 1 that is REST action (basically you can leverage iMC eAPI to do things).

Next definition is action (Asn1Action). serviceName and actionName are very important as this is how adapter is called (strings will need to match what we will do later in adapters). operMode 1 indicates CLI (SNMP adapters are also possible, but not documented), operType is 1 (deploying configuration changes) and execMode is 1 (synchronous operation).

Last definitions are about parameters which will be passed to your adapter. You can pass built-in variables to your scripts, for example when dealing with interface wizards you will most likely define something like this:

```xml
<Parameter name="ifDesc" value="${IF_DESC}" />
```

So your script will get ifDesc variable where interface will be submitted (so your script can do things like send "interface $ifDesc\r").

Full list is part of iMC documentation:
Certainly you will need more than built-in variables and you will probably want to have interaction with user. You can define parameters with certain View type, which actually mean GUI component. Options include things like check box, input box, combo box or even calendar. Some of those items (such as combo box) can be filled statically (so you define values of your choice) or even dynamically (you can define action in your adapter or REST action that will poll device for that list – for example you can list existing configured BGP peers). There are some attributes you can use with your parameters, for example check input against simple RegEx (we are using this to check whether input looks like IP address), or set some fields as required.

Now create empty file custom.properties. In our example we have hardcoded GUI strings directly into operations file, but you can place those into custom.properties file and reference it in operation definition (which is good practice) – but for simplicity we have not done so, but file needs to exist in order for custom batch operations to work.
For troubleshooting issue in your custom files you can check
\[/IMC/client/log/imcforeground.log\]


Select Device Configuration Guide

Click Add operation and note there is new Custom category of operations.
Add devices and note that only Comware-based devices are listed.

Click Next and you will see GUI asking for parameters.
You can now test our RegEx on IP address field.

For now do not continue as we have not created adapters to support this new wizard.

Also note that there are more ways to initiate this. For example you can do it from devices list by selecting some and clicking on More. Interface Configuration Guide can be called from interface list.

You can also initiate wizard from topology view.
GUI is working as expected, we now need to implement actual adapters. You can find existing adapters in folder

/IMC/server/conf/adapters/GenericConfig/

Structure is very similar to configuration backup drivers described before in this guide. There are vendor folders with adapter index files that point to actual adapters. You might create your own vendors in similar way as was described previously. We will actually extend existing H3C adapter. Please note that files we will add will survive iMC upgrades, but changes done to adapter.xml will be lost. First modify following file:

/IMC/server/conf/adapters/GenericConfig/H3C/H3CCommonCLI/adapter.xml

We will add new service called ComwareNTP (that is reference from our GUI components) – make sure you place it within services section:

  <service name="ComwareNTP">
    <item type="builder_definition">H3CCommonCLI_NTP_Builder.xml</item>
    <item type="tcl_script">H3CCommonCLI_NTP_Builder_Script.xml</item>
  </service>
We have defined new service and pointed it to builder definition and scripts definition. First create H3CCommonCLI_NTP_Builder.xml file:

```xml
<?xml version="1.0"?>
<definition>

  <action name="ComwareNTPaction">
    <step>
      <command name="setup_ntp" description="configure ntp">
      </command>
    </step>
  </action>

</definition>
```

We have defined new action where name is actual reference from GUI components and created steps and commands within. Commands are defined in H3CCommonCLI_NTP_Builder_Script.xml, so let's create this file:

```xml
<?xml version="1.0"?>
<scripts>

  <command name="setup_ntp" method="CLI">
    <error>Failed to setup ntp.</error>
    <require-mode>enable</require-mode>
    <script>
      setup_ntp.tcl
    </script>
  </command>

</scripts>
```

We require adapter to enter enable mode first and then execute setup_ntp.tcl script. Last thing to do is create actual script file:

```tcl
expect -re $enable_prompt

send "ntp-service enable\r"

expect -re $enable_prompt

if {$Authentication} {
```
send "ntp-service authentication enable\r"
expect -re $enable_prompt
send "ntp-service authentication-keyid $keyid authentication-mode md5 simple $key\r"
expect -re $enable_prompt
send "ntp-service reliable authentication-keyid $keyid\r"
expect -re $enable_prompt
send "ntp-service unicast-server $server authentication-keyid $keyid\r"
} else {
  send "ntp-service unicast-server $server\r"
}

expect -re $enable_prompt

if {$Commit} {
  send "save force\r"
}

expect -re $enable_prompt

What is happening there? Note we are referencing variables defined in GUI. We start by enabling ntp-service and that create if statement to check whether user wants authenticated service. If yes we configure actual commands and reference variables such as key, keyed and server ip address.

Now we are ready to test this. First you need to restart imcmdmgrdm service.
For troubleshooting check following log file (you might want to use tail command or equivalent in Windows environment) where last part of the file is date stamp.


Let’s now enter GUI and fill in required parameters.
After short time you should see your task finished.

You can check Result view

Click on question mark to see actual sequence of commands
Also you can look into logs on server where you will something similar to this:

2014-07-23 07:03:25.656 [INFO (0)] [THREAD(3726637936)] [CScriptProcessor::batchExec()] Finished, result:

ServiceName = ComwareNTP
ActionName = ComwareNTPaction
InputParam =
Authentication=true?Commit=true?key=hp?keyid=1?server=10.10.10.123
OutputParam =
CmdResp =

Configuration is saved to device successfully.
We are going to stop here with this simple example and note you can send this to any number of devices you like in batch (adapter is called for each device selected in GUI so if you implement this for ProVision and Cisco you can mix all different types of devices in this batch operation). Also note that there some more work to do if you want – for example you might want to clean up previous configurations (your script would need to check configuration for ntp-service lines and undo those), extend script and GUI with support for multiple ntp servers configuration, use combo box with some predefined servers to choose from etc.

Here is full demo solution:

Custom batch operation.zip

Creating and using custom cloud-based SMS gateway

iMC and UAM comes with very useful features in guest management with possibility to send information over SMS or forward alarms to SMS. Built-in support requires usage of hardware SMS gateway connected to iMC server, therefore need to have such hardware, SIM card and contract with operator. This might not be preffered method for some customers and can be limitation during PoCs, testing or lab environment. There are many cloud-based SMS gateways available on the market.

iMC UAM does have capability to extend its Java API with custom implementation. This document describes implementation for one selected provider (www.textmagic.com) and can be extended to others. As this company is offering free trial accounts for 10 messages it is well suited for PoCs and operates in more than 200 countries.

First we need to configure iMC to know about our Java API implementation. Simply copy file acmExternalCfg.xml to folder <IMCfolder>/client/conf/

Second step is to modify our application settings file to fit your needs. Open imcSmsGateway.conf and change username and password based on your registration details with www.textmagic.com. Currently no other provider is implemented, but you might extend source code accordingly. Debug flag should not be used in production. Once you are ready copy this file to <IMCfolder>/client/conf/
Third step is to copy our Java application that implements iMCs Java API for sending messages. Copy file imcSmsGateway.jar to your <iMCfolder>/client/repository/imc/jars/ 

Now please restart your iMC server.

Configure iMC to use our custom implementation in System, System Configuration, SMSC Settings by choosing Other SMS Sender:

![System > SMSC Settings](image)

Repeat the same for UAM under User, User Access Policy, System Parameters, System Settings, SMS Settings:

![User > User Access Policy > Service Parameters > System Settings > SMS Settings](image)

You can check operations in log file:

imc/client/log/imcSmsGateway.log

Demo extension provided does support only one provider (Textmagic). You can implement other services if you want. If you decide to add new provider follow the same logic and extend application with support for other providers and share with community. Easiest way to start is to import existing project from attached zip file into your Eclipse. If you start your own you need to add external JAR with iMC Java API declaration (you can get it from iMC repository folder and file you are looking for is imc_acm.jar).

Code is documented and should be self explanatory.