HPSA Extension Pack

Equipment Connections Pool Quick Start Guide

Release v 5.1

Legal Notices

Table of Contents

Support

Support for the HP Service Activator Extended Pack product is available on the following mailing list:

ovsa.spain.support@hp.com

In This Guide

This guide will explain the configuration, installation, needed development, and functionality provided by the ECP.

Audience

The audience for this guide is the Solutions Integrator (SI). The SI has a combination of some or all of the following capabilities:

Understands and has a solid working knowledge of:

- UNIX® commands
- Windows® system administration

Understands networking concepts and language

Is able to program in Java™ and XML

Understands security issues

Understands the customer's problem domain

Conventions

The following typographical conventions are used in this guide.

Font	What the Font Represents	Example
Italic	Book or manual titles, and man page names	Refer to the *HP Service Activator — Workflows and the Workflow Manager* and the *Javadocs* man page for more information.
	Provides emphasis	You *must* follow these steps.
	Specifies a variable that you must supply when entering a command	Run the command: `InventoryBuilder` *<sourceFiles>*
	Parameters to a method	The *assigned_criteria* parameter returns an ACSE response.
Bold	New terms	The **distinguishing attribute** of this class...
`Computer`	Text and items on the computer screen	The system replies: `Press Enter`
	Command names	Use the `InventoryBuilder` command ...
	Method names	The `get_all_replies()` method does the following...
	File and directory names	Edit the file `$ACTIVATOR_ETC/config/mwfm.xml`
	Process names	Check to see if `mwfm` is running.
	Window/dialog box names	In the `Test and Track` dialog...
	XML tag references	Use the `<DBTable>` tag to...
`Computer Bold`	Text that you must type	At the prompt, type: **`ls -l`**
Keycap	Keyboard keys	Press **Return**.
[Button]	Buttons on the user interface	Click `[Delete]`. Click the `[Apply]` button.
Menu Items	A menu name followed by a colon (:) means that you select the menu, then the item. When the item is followed by an arrow (->), a cascading menu follows	Select Locate:Objects->by Comment.

Install Location Descriptors

The following names are used throughout this guide to define install locations.

Descriptor	What the Descriptor Represents
`$ACTIVATOR_OPT`	The install base location of Service Activator. The UNIX location is `/opt/OV/ServiceActivator` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\`
`$ACTIVATOR_ETC`	The install location of specific Service Activator configuration files. The UNIX location is `/etc/opt/OV/ServiceActivator` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\etc\`
`$ACTIVATOR_VAR`	The install location of specific Service Activator logging files. The UNIX location is `/var/opt/OV/ServiceActivator` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\var\`
`$ACTIVATOR_BIN`	The install location of specific Service Activator binary files. The UNIX location is `/opt/OV/ServiceActivator/bin` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\bin\`
`$ACTIVATOR_THIRD_PARTY`	The location for new Java components such as workflow nodes and modules. Third-party libraries can also be placed in this directory. The UNIX location is `/opt/OV/ServiceActivator/3rd-party` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\3rd-party\` Customized inventory files are stored in the following locations: UNIX: `$ACTIVATOR_THIRD_PARTY/inventory` Windows: `$ACTIVATOR_THIRD_PARTY\inventory`
`$JBOSS_HOME`	HOME The install location for JBoss. The UNIX location is `/opt/HP/jboss` The Windows location is `<drive>:\HP\jboss`
`$JBOSS_DEPLOY`	The install location of the Service Activator J2EE components. The UNIX location is `/opt/HP/jboss/server/default/deploy`

	The Windows location is `<drive>:\HP\jboss\server\default\deploy`
`$ACTIVATOR_DB_USER`	The database user name you define. Suggestion: `ovactivator`
`$ACTIVATOR_SSH_USER`	The Secure Shell user name you define. Suggestion: `ovactusr`
`$SOSA_HOME`	The install base location of SOSA. The default UNIX location is `/opt/OV/Sosa` The default Windows location is <drive>:\HP\OpenView\Sosa\
`$SOSA_BIN`	The install location of specific SOSA binary files. The default UNIX location is `/opt/OV/Sosa/bin` The default Windows location is `<drive>:\HP\OpenView\Sosa\bin\`
`$SOSA_ETC`	The install location of specific SOSA configuration files. The default UNIX location is `/opt/OV/Sosa/config` The default Windows location is <drive>:\HP\OpenView\Sosa\config\
`$ECP_HOME`	The install base location of Equipment Connections Pool. The default UNIX location is `/opt/OV/ECP` The default Windows location is <drive>:\HP\OpenView\ECP\
`$ECP_BIN`	The install location of specific Equipment Connections Pool binary files. The default UNIX location is `/opt/OV/ECP/bin` The default Windows location is <drive>:\HP\OpenView\ECP\bin\
`$ECP_ETC`	The install location of specific Equipment Connections Pool configuration files. The default UNIX location is `/opt/OV/ECP/conf` The default Windows location is <drive>:\HP\OpenView\ECP\conf\

1 Introduction

1.1 Purpose

This document is a manual for all ECP Module users. It gives a general view of the ECP and the initial steps needed to have it up and running.

1.2 General Description

The ECP has two main functions: Pooling TCP/IP connections and automating telnet sessions through those connections (other protocols, such as SSH or raw TCP are available).

Connection Pooling allows the user to configure how and when the ECP will connect, optimizing speed and resources use. Session automation eases for the client program the burden of sending commands to the destination and interpreting their result.

A session is automated sending to the ECP a "Commands Template", which is a java String which though a determinate syntax, describes de commands to send and how to process and interpret their result, establishing their execution logic.

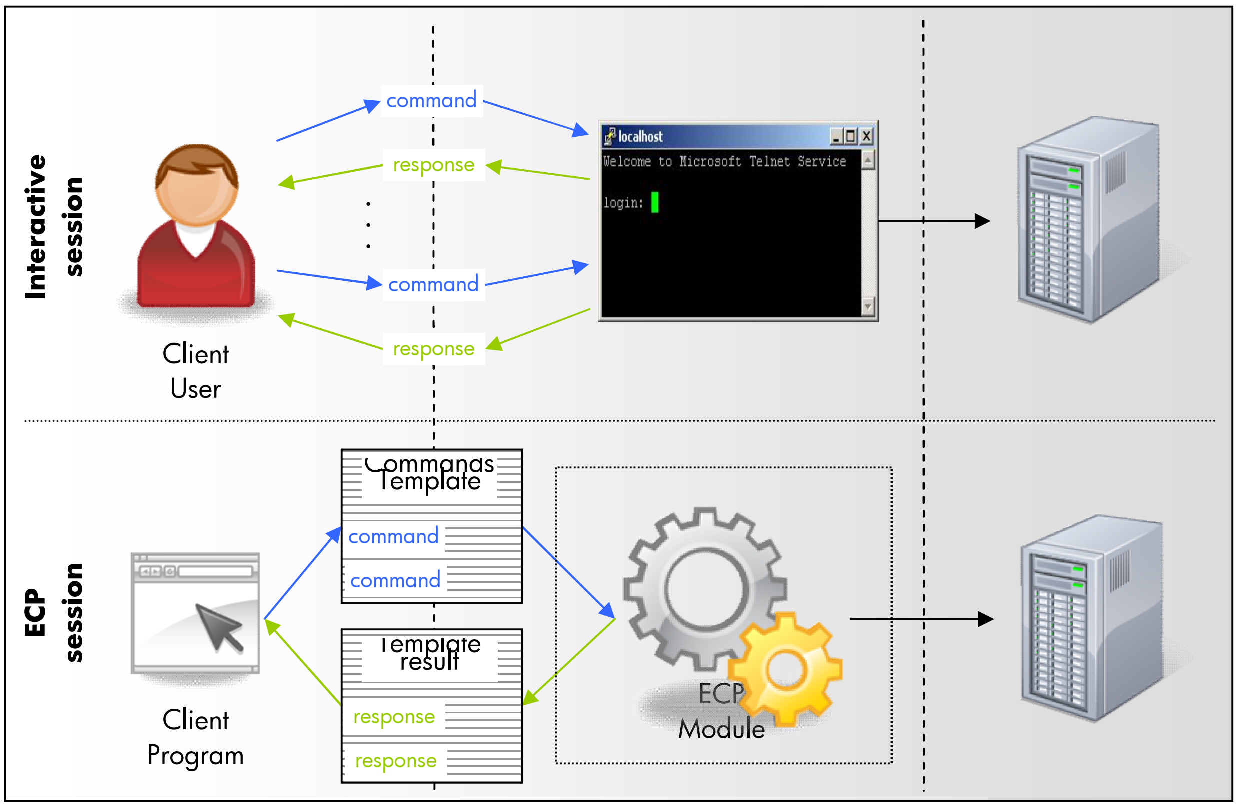


Fig. 1: Interactive sessions vs. ECP automated sessions.

The ECP will pool the connections used to execute the commands. The connections will be created and destroyed as needed, and the same connection may be used to execute different consecutive automated sessions. The ECP keeps different pools, each one managing the connections for a single destination. When commands need to be executed on a client request, the ECP will borrow a connection from the appropriate pool and use it to send the commands. When all the commands are sent, the connection will be returned to the pool. For the ECP to use a pool it must be first created and appropriately configured. A GUI is provided to administer the ECP pools.

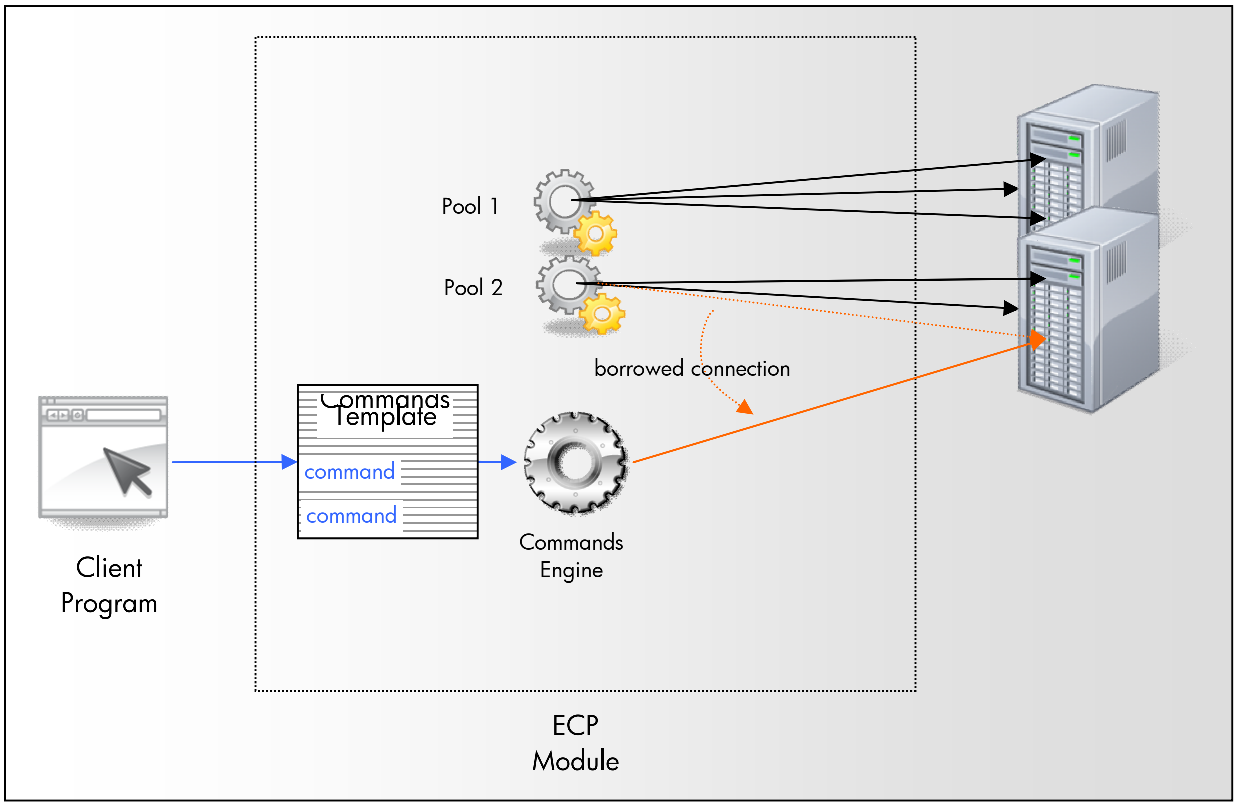


Fig. 2: ECP connection pooling.

2 Equipment Drivers

To manage the connections through which commands are sent, the ECP pools need to be able to establish, close and verify a session on the destination. Establishing a session usually implies satisfying some authentication measure imposed by the destination (such as providing a login and a password). Similarly, a clean disconnect usually must be done through certain commands (such as "`logout`" or "`exit`"). A "session verify" is performed by the ECP to check whether the session is still alive and well on the destination, and thus, can be reused. Verifying a session usually entails sending an innocuous command to the destination and checking that the appropriate answer is received, which also is a destination dependant process. These three processes are called "initialization", "finalization" and "verification" respectively.

As the three of these processes depend on the type of equipment to which the connections are established, the user must provide for each type of equipment a java class which is able to perform the "initialization", "finalization" and "verification". A class which is able to perform the "initialization", "finalization" and "verification" is called an EquipmentDriver. The ECP will create an instance of the appropriate EquipmentDriver for each connection. For example, if the solution requires some command templates to be run on a Linux system and some others to be fun on an HP-UX system, two different EquipmentDriver must be implemented: one for Linux must and another one for HP-UX.

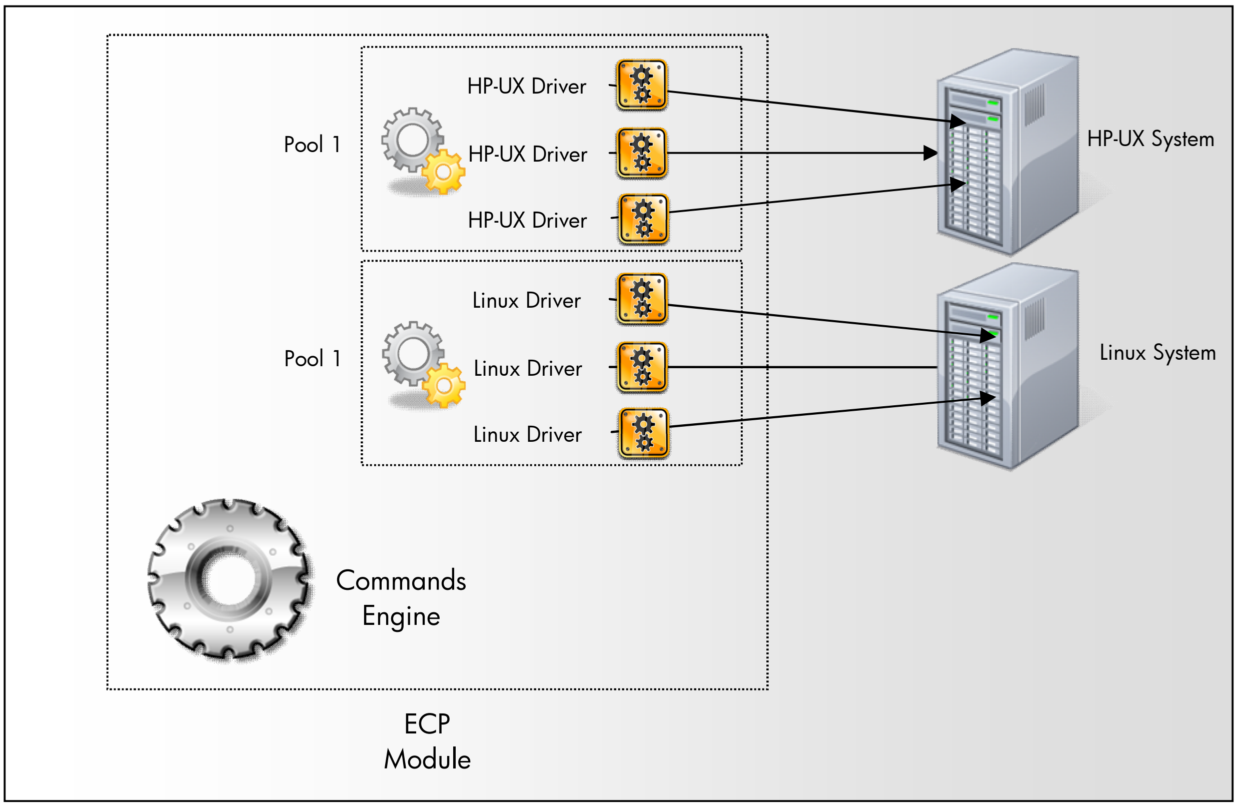


Fig. 3: EquipmentDriver classes instances in the ECP.

2.1 Default EquipmentDriver – TemplateDriver

The TemplateDriver is the default equipment driver provided by ECP. This driver is able to connect to most of the equipments that use the CLI interface.

This equipment driver is configured using the class com.hp.spain.connection.TemplateDriver. We can configure this driver adding into the DriverSpecificParameters the extra variables on properties format or referring to the Common Configuration.

The next 5 templates that can be configured into the database or into a file, finding first in database.

LOGIN_TEMPLATE: template to make the login (note: this template has sense in protocol driver that doesn'tmakes the authentication)

LOGOUT_TEMPLATE: the logout template, typically the exit command

ENTER_CONFIG_MODE_TEMPLATE: the template to configure all the sessions attributes required.

EXIT_CONFIG_MODE_TEMPLATE -> the template to unconfigure

VERIFY_TEMPLATE: template to verify if the connection is ok.

These templates will receive the parameters configured into the DriverSpecificParameters and the next parameters configured into the subpool:

USER: user name

PASSWORD: user password

PASSWORD_ENABLE: password enable

HOST: ip host value

Also, next variables can be define to make easier the templates:

LOGIN_USER_PROMPT: synchronize the driver with the login prompt.

LOGIN_PWD_PROMPT: synchronize the driver with the password prompt.

INITIAL_PROMPT: synchronize the driver with the inital prompt.

Also, this driver has the capability to add error patterns, failure patterns, non error patterns and error message to all the commands that are executed into a command template. In case, it's required to add these patterns to the connections templates (LOGIN_TEMPLATE, ENTER_CONFIG_MODE_TEMPLATE, …) the variable ADD_PATTERNS_CONNECTION_TEMPLATES has to be setted to true.

The only requirement to set these patterns is define variables with next prefix:

ENDSTRING_PATTERN

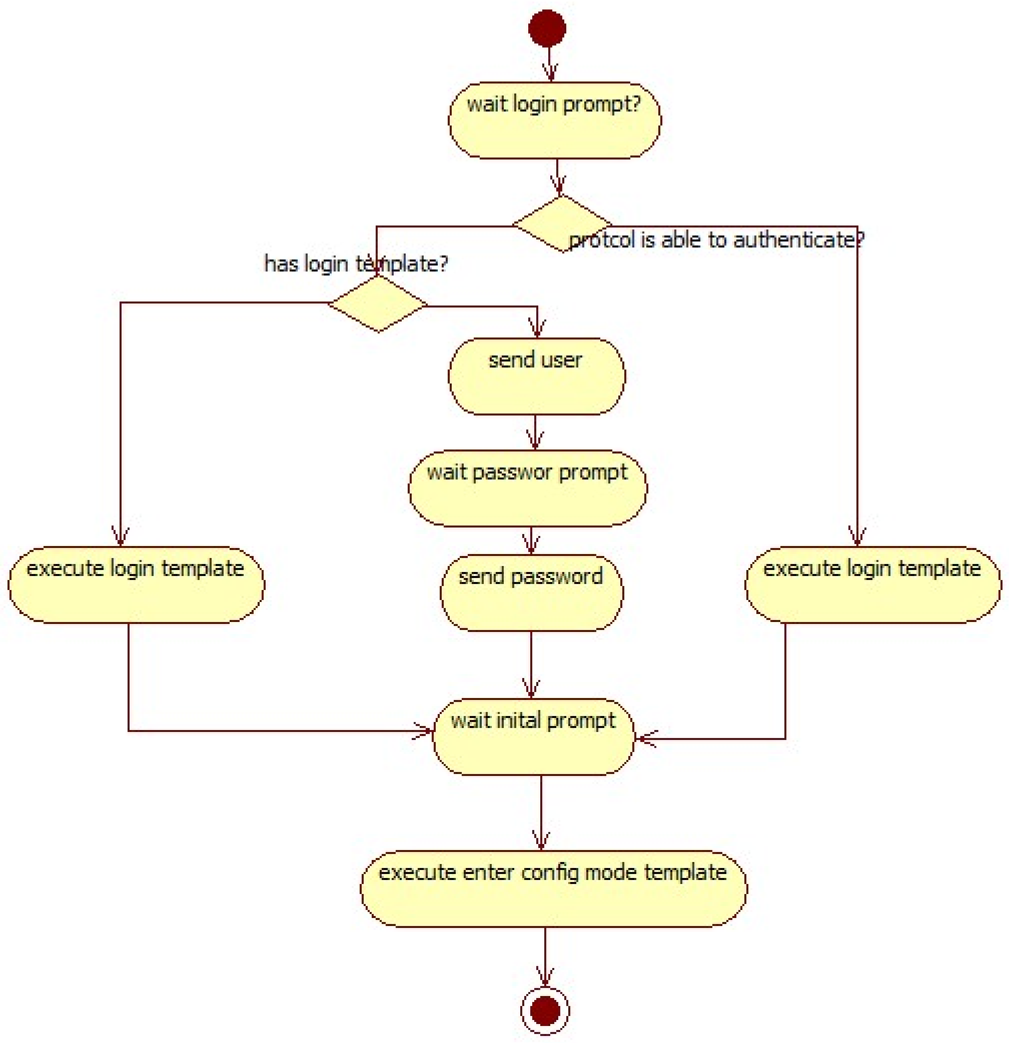
ERROR_PATTERN

FAILURE_PATTERN

NONERROR_PATTERN

ERROR_MESSAGE: in this case only can be defined one and the variable is required to have this name.

Connection flow:



When the driver starts the connection the first step is to check if the LOGIN_USER_PROMPT is configured. In that case, synchronize this prompt. After that, the LOGIN_TEMPLATE is executed if it's configured. If not and the protocol driver doesn't support authentication, send the user, synchronize the password prompt (LOGIN_PWD_PROMPT) and send the password.

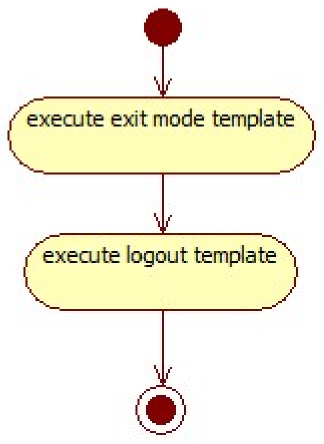
In this moment, the driver is authenticated and in case the INITIAL_PROMPT is configured the driver synchronizes the initial prompt.

Usually, when the protocol supports the authentication (for example, ssh) it's only necessary to configure the INITIAL_PROMPT and not the LOGIN_TEMPLATE and neither LOGIN_USER_PROMPT.

After synchronize the INITIAL_PROMPT the driver execute the ENTER_CONFIG_MODE_TEMPLATE and finally executes the VERIFY_TEMPLATE.

In this moment, the driver is connected and ready to be used.

Disconnection flow:



First, the driver execute the template EXIT_CONFIG_MODE_TEMPLATE and after that the LOGOUT_TEMPLATE.

2.2 EquipmentDriver States

For the ECP to be able to operate with an EquipmentDriver, the EquipmentDriver must publish its current state. An EquipmentDriver can remain in four different states: `DEACTIVATED`, `INACTIVE`, `ACTIVE` and `BUSY`. By publishing its state, the EquipmentDriver informs the ECP of the operations it is able to undertake or needs to perform.

a) `INACTIVE/DEACTIVATED`: From the EquipmentDriver point of view these two states are usually equivalent. Both signify that the EquipmentDriver has no connection established with the destination.

b) `ACTIVE/BUSY`: From the EquipmentDriver point of view these two states are usually equivalent. The connection with the destination has been established and the session is authenticated. The EquipmentDriver is ready to send commands and the session on destination ready to receive them.

The EquipmentDriver is responsible for transitioning to the appropriate states, depending on the result of the "initialization", "finalization" and "verification" processes. Each process is implemented as a different method of the EquipmentDriver that the ECP will call when appropriate. The valid state transitions and method calls are described by the state diagram which follows. Notice that the finalize call is not dependant on the current EquipmentDriver state, that is, the finalize method can be called for all the possible EquipmentDriver states (source state calls arrows have been omitted to simplify diagram):

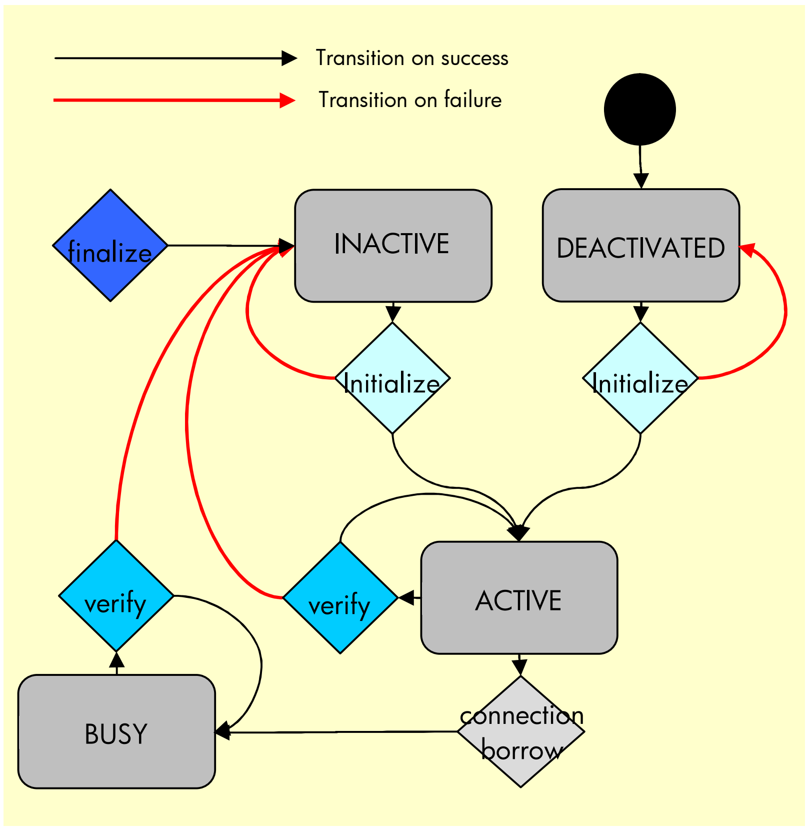


Fig. 4: EquipmentDriver states, calls and transitions.

The ECP will ensure that the calls are performed in an always mutually exclusive way. As a consequence, the EquipmentDriver implementer needs not to care about concurrency.

2.3 Implementing an EquipmentDriver

All EquipmentDriver classes must include the following jars in their classpath:

```
velocity-dep-xxx.jar
log4j-xxx.jar
equipments-connections-pool-xxx.jar
resource-manager-xxx.jar
```

Those jars are present in the $ECP_LIB directory of your installation. The ECP will always include them at runtime. They should also be included at develop/compile time.

Any additional jars your EquipmentDriver may need must be manually included in the $ECP_LIB directory of your installation to be available at runtime. The developer should make sure that any additional jar will not interfere with any of the jars present in $ECP_LIB. The ECP needs to be restarted to include new or modified jars.

As previously mentioned, an EquipmentDriver must be able to perform the "initialization", "finalization" and "verification". Those three processes are implemented by the EquipmentDriver by means of three different methods: `initialize(HashMap parameters)` , `finalize()`, `verify()` respectively.

A typical EquipmentDriver declaration (`TestConnectionResource` in the example) will be as follows:

```
package com.hp.spain.connection;

import java.util.HashMap;
import java.util.Vector;

import org.apache.log4j.Level;
import org.apache.oro.text.regex.MalformedPatternException;

import com.hp.spain.resmgr.ResourceStatus;

public class TestConnectionResource extends ConnectionResource {

  public TestConnectionResource(HashMap parameters) throws DriverException {
     super(parameters);
  }

  public void initialize(HashMap parameters) {
     //omitted
  }

  public void finalize() {
     //omitted
  }

  public ResourceStatus verify() {
     //omitted
  }
}
```

All EquipmentDriver must extend the class `com.hp.spain.connection.ConnectionResource`. As a consequence, much of the functionality needed to implement an EquipmentDriver is inherited from those classes as methods or provided by parent class members. The constructor will receive a `HashMap` containing the connection configuration (ip, port, protocol etc…). These parameters will be processed by the parent classes and a call to the parent's constructor is mandatory. The parameters may contain EquipmentDriver class specific configuration (see the Developer Reference for further details). The methods `initialize`, `finalize` and `verify` will be analyzed later.

2.3.1 EquipmentDriver Initialization

The driver initialization is performed by the `initialize(HashMap parameters)` method. This method must perform all necessary actions to start a session on the destination which is able to receive commands. This usually entails establishing a TCP/IP connection to the destination and authenticating. Some times it may include initiating a write mode on the destination through specific commands.

A typical initialize method implementation will be:

```
  private static Vector initialPrompts;
  private static EquipmentCommand loginCmd;
  private static EquipmentCommand pwdCmd;
   private static int socketTimeout = 0;

  static {
     try {
       initialPrompts = new Vector();
       initialPrompts.add("login: $");

       loginCmd = new EquipmentCommand();
       loginCmd.putEndStrPattern("password: \\**");

       pwdCmd = new EquipmentCommand();
       pwdCmd.putFailure("login: $");
```

```
      pwdCmd.putEndStrPattern("\\w\\:[^\\n\\r]*>$");
      pwdCmd.setNoEcho();
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }

  public void initialize(HashMap parameters) {
    super.initialize(parameters);

this.logger.log(Level.INFO, "Initializing Equipment Driver " + this);

    try {
      // establish the connection
      super.connectServer(this.host, this.port, socketTimeout);
      // wait for the initial prompt
      if (!super.confTerminal(initialPrompts)) {
        throw new Exception("Initial prompt was not found.");
      }
      // send login and pwd as commands
      loginCmd.putCommand(user);
      pwdCmd.putCommand(password);
      authenticate(loginCmd, pwdCmd, null);
      // everything OK, set status
      setStatus(ResourceStatus.ACTIVE);
      setSessionStatus(CONNECTED);
      this.logger.log(Level.INFO, "Finished initialization of Equipment
Driver " + this);
    } catch (Throwable t) {
      this.logger.error("Error initializing Equipmemt Driver will cause
finalization " + this, t);
      finalize();
    }
  }
```

This method will be called only if the EquipmentDriver state is `DEACTIVATED` or `INACTIVE`.

The method will receive a `HashMap` containing the connection configuration (IP, port, protocol etc…). These parameters will be processed by the parent classes and a call to the parent's `initialize(HashMap parameters)` method is mandatory. The parameters may contain EquipmentDriver class specific configuration (see the Developer Reference for further details). The received `HashMap` might be `null`, meaning that the configuration previously received through the constructor or a previous call to the `initialize` method must be used for initialization.

After parent initialization, the method will typically establish the connection with the destination, through the parent method `connectServer(String sHostname, int iPort, int iTimeOut)`.

After establishing the connection, the EquipmentDriver must wait for the initial prompt to be received by calling the method `confTerminal(Vector vInitialString)`. The method will receive a vector containing the alternative expected prompts (`initialPrompts` in the example), described through a regular expression like syntax (see the Developer Reference for further details). In case the destination requires authentication, the initial prompt will usually be a login prompt (the expression `"login: $"` in the example).

When the initial prompt has arrived, login and password may be sent as regular commands (see the Developer Reference for further details regarding how to send commands), but using the method `authenticate(EquipmentCommand tcLogin, EquipmentCommand tcPassword, HashMap hmVariables)`. That method will distinguish if the protocol using to communicate with the destination (SSH, Telnet…) defines any special rule when authenticating.

When the process is completed without errors the EquipmentDriver state may be changed to ACTIVE.

The implementer must be careful not to leave the connection opened if an error is encountered, because destinations usually limit the number of connections that can be opened simultaneously. The simplest way to do this usually is to rely on the finalization method.

This implementation could be valid to connect to the built-in telnet server of a windows machine.

2.3.2 EquipmentDriver Verification

The driver verification is performed by the `verify()` method. This method must perform all necessary actions to check whether the connection with the destination remains opened, authenticated and in the needed state to receive commands. This usually entails simply sending a command with no side effects on the destination and checking the answer received to the command.

A typical initialize method implementation will be:

```
  private static EquipmentCommand cmdVerify = new EquipmentCommand();

  static {
    try {
      cmdVerify.putCommand("");
      cmdVerify.putNonError("*");
      pwdCmd.putEndStrPattern("\\w\\:[^\\n\\r]*>$");
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }

  public ResourceStatus verify() {
    logger.log(Level.INFO, "Verifying Equipment Driver " + this);

    try {
      super.execAction(cmdVerify, true, new HashMap(), false);
      //clean the session commands log
      super.clearStdOut();
      super.clearCommandsSent();
    } catch (Throwable e) {
      this.logger.error("Error verifying Equipmet Driver " + this, e);
      finalize();
    }
    logger.log(Level.INFO, "Finished verification of Equipment Driver " +
this);
    return this.getStatus();
  }
```

This method will be called only if the EquipmentDriver state is ACTIVE or BUSY.

The example sends an empty commands (that is, simply an “enter”) and expects it to be answered with the prompt (see de Developer Reference for further details on sending commands).

After successfully sending the command and receiving its answer, the commands log is deleted. This action is needed for the verify command not to be included in the commands log sent to the client when the commands template execution ends.

If an error is encountered during verification the EquipmentDriver must transit to the INACTIVE state and ensure the connection is closed. The simplest way to do this usually is to rely on the finalization method.

This implementation could be valid to verify a connection to the built-in telnet server of a windows machine.

2.3.3 EquipmentDriver Finalization

The driver finalization is performed by the `finalize()` method. This method must perform all necessary actions to perform a clean session exit, and if it is not possible, grant that the connection is closed. This usually entails sending an exit command and closing the connection.

A typical initialize method implementation will be:

```
  private static EquipmentCommand tcExit;
  static {
    tcExit = new EquipmentCommand();
    tcExit.putCommand("exit");
    tcExit.putNonError("*");
    tcExit.putEndStr("*");
  }

  public void finalize() {
    try {
      logger.log(Level.INFO, "Finalizing Equipment Driver " + this);
      if (getStatus().equals(ResourceStatus.ACTIVE) ||
getStatus().equals(ResourceStatus.BUSY)) {
        // send exit commands
        super.execAction(tcExit, true, new HashMap(), false);
      }
    } catch (Throwable e) {
      this.logger.error("Error finalizing Equipmet Driver " + this, e);
    } finally {
      //set status
      if (!getStatus().equals(ResourceStatus.DEACTIVATED) &&
!getStatus().equals(ResourceStatus.INACTIVE)) {
        setStatus(ResourceStatus.INACTIVE);
        setSessionStatus(DISCONNECTED);
      }
      super.closeConnection();
      logger.log(Level.INFO, "Finished finalization of Equipment Driver " +
this);
    }
  }
```

This method will be called for all the EquipmentDriver states. It can even be called more than once, that is, finalizing an EquipmentDriver which has already been finalized.

The example will first try to send the exit command (`"exit"` in the example, see de Developer Reference for further details on sending commands). Whatever the result of the command is, the connection will be closed after that, even if an error is encountered, through a `finally` block.

This implementation could be valid to connect to the built-in telnet server of a windows machine.

3 Creating an Administered Pool

The easiest way to configure the ECP pools is through the GUI module. This section will show how to create an example pool that will use the EquipmentDriver shown in 2.3 Implementing an EquipmentDriver

For the full description on how to administer the ECP pools through the GUI module see the document "ECP Administration GUI - User Reference".

Before creating the example pool, the developer should copy the jar containing the example EquipmentDriver (and any jar it may need) to the $ECP_LIB directory (as described in 2.3 Implementing an EquipmentDriver) and restart the ECP.

To access Pool Creation the user should login to the Future GUI (see the document "Solution Container User Reference" for further details) and select the menu "Administrator" - "ECP" - "Pool" - "New".

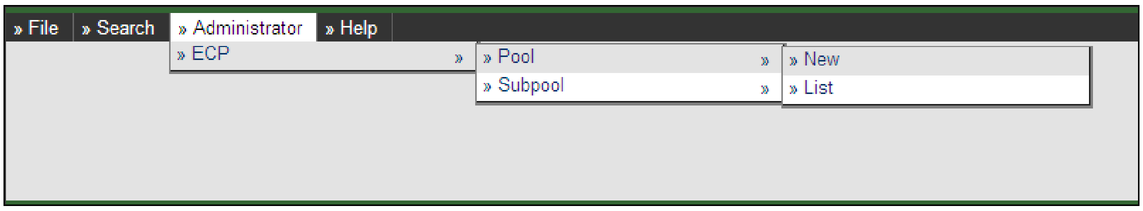
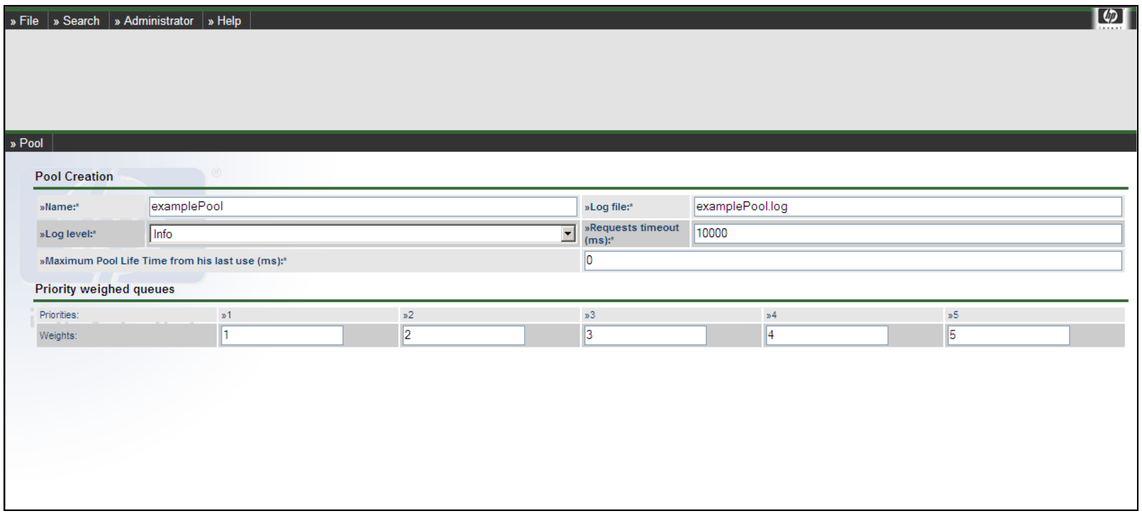


Fig. 5: View, Pool creation

When the Pool Creation Menu is selected the ECP GUI will load the "Pool Creation" screen where the user may enter the new pool configuration.



Pool Creation

»Name:*	examplePool	»Log file:*	examplePool.log
»Log level:*	Info	»Requests timeout (ms):*	10000
»Maximum Pool Life Time from his last use (ms):*		0	

Priority weighed queues

Priorities:	»1	»2	»3	»4	»5
Weights:	1	2	3	4	5

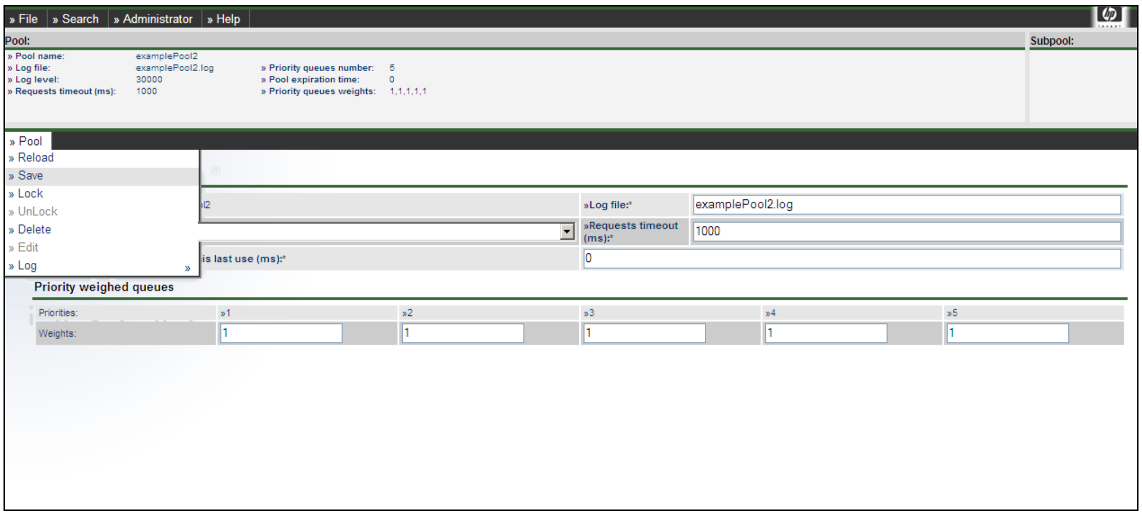
Fig. 6: Operation, Pool creation

For this example the values will be:

- Name: The pool name. It must be unique and will be used to identify the pool from now on. The example pool name will be "examplePool".
- Log File: The name of the file where the pool logs will be recorded; it must be unique and will be created at "/opt/OV/ServiceActivator/ECP/log". The example log file will be "examplePool.log".
- Log Level: The minimum level of severity of the messages to be written in the log. For developing purposes the recommended level is Debug.

- Requests timeout: 10000 in this example.
- Maximum Pool Life Time from his last use (ms): 0 in this example.
- Priority Weighed Queues: For these example the weights will be 1, 2, 3, 4 and 5 for the respective queues 1, 2, 3, 4 and 5.

Once all fields have been filled in, selecting "Pool" - "Save" in the Status Menu will save the new pool.



» Pool name:	examplePool2		
» Log file:	examplePool2.log	» Priority queues number:	5
» Log level:	30000	» Pool expiration time:	0
» Requests timeout (ms):	1000	» Priority queues weights:	1,1,1,1,1

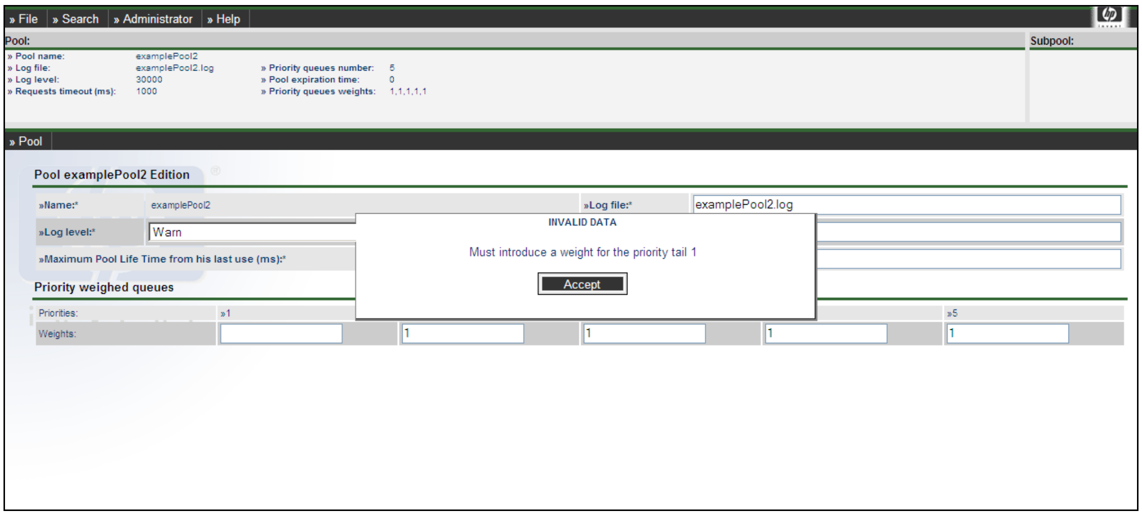
l2		»Log file:*	examplePool2.log
		»Requests timeout (ms):*	1000
is last use (ms):*			0

Priority weighed queues

Priorities:	»1	»2	»3	»4	»5
Weights:	1	1	1	1	1

Fig. 7: Status, Pool save

If any field is wrong or a mandatory field as been left empty, the ECP GUI will load a screen with a descriptive error message.



» Pool name:	examplePool2		
» Log file:	examplePool2.log	» Priority queues number:	5
» Log level:	30000	» Pool expiration time:	0
» Requests timeout (ms):	1000	» Priority queues weights:	1,1,1,1,1

Pool examplePool2 Edition

»Name:*	examplePool2	»Log file:*	examplePool2.log
»Log level:*	Warn		
»Maximum Pool Life Time from his last use (ms):*			

INVALID DATA

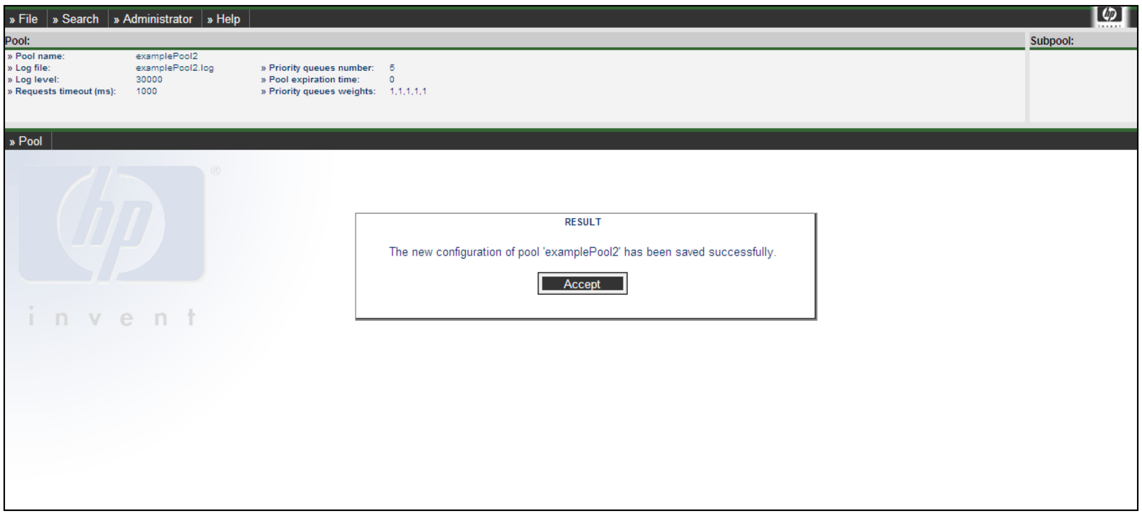
Must introduce a weight for the priority tail 1

Priority weighed queues

Priorities:	»1				»5
Weights:		1	1	1	1

Fig. 8: Operation, Pool save error

If the pool configuration is valid, the ECP GUI will show the following message.



» Pool name:	examplePool2		
» Log file:	examplePool2.log	» Priority queues number:	5
» Log level:	30000	» Pool expiration time:	0
» Requests timeout (ms):	1000	» Priority queues weights:	1,1,1,1,1

The new configuration of pool 'examplePool2' has been saved successfully.

Fig. 9: Operation, Pool saved

After creating a Pool, a SubPool belonging to that Pool must be created. Check the Developer Reference and Administrator Reference for further details on their use.

In case, the Subpool will use the default driver (this is the easiest way because there is no need to develop any EquipmentDriver) it's required to create a Common Configuration. A Common Configuration is a set of configuration to connect a type of element. In the next example, we'll configure the typical configuration to connect via telnet to a windows machine.

To access Common Configuration creation the user must select from the Views menu "Administrator" - "ECP" - "Common Configuration" - "New".

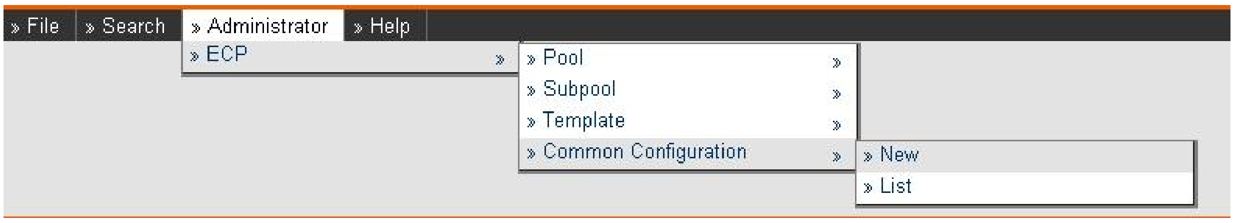
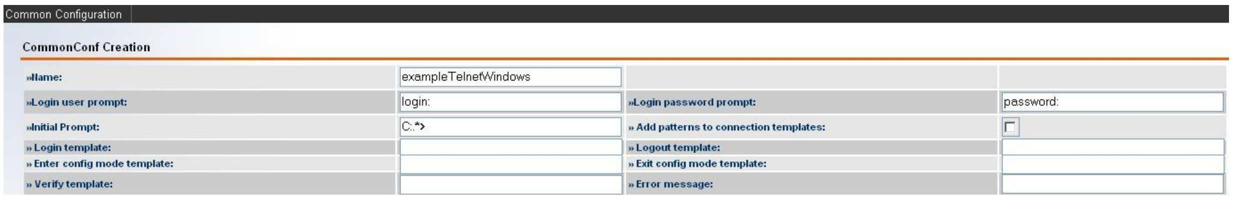


Fig. 10: Common Conf creation

The ECP GUI will show the "Common Configuration Creation" screen where the information must be entered.



CommonConf Creation

»Name:	exampleTelnetWindows		
»Login user prompt:	login:	»Login password prompt:	password:
»Initial Prompt:	C:*>	» Add patterns to connection templates:	☐
» Login template:		» Logout template:	
» Enter config mode template:		» Exit config mode template:	
» Verify template:		» Error message:	

Fig. 11: Common Conf edition

This form has next values:

- Login user prompt: in this case "login:" It's important to fill this field when the protocol cannot authenticate (telnet). In case the protocol authenticate (ssh) this field it's ignored.
- Login password prompt: in this case "password:"Same as fieldbefore in case the protocol authenticate will ignore.
- Initial prompt: "C:.*>". This is the first prompt after login.

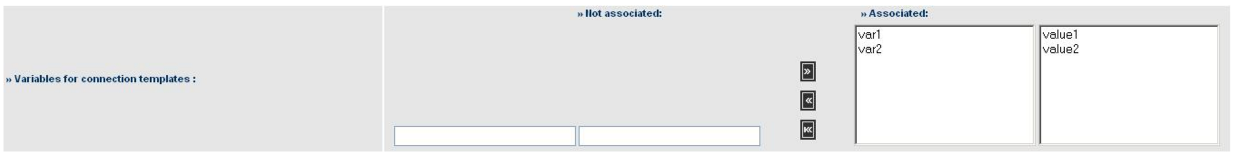
Configuring the last 3 field, in most of the case, there's no need to configure more parameters because the TemplateDriver is able to connect in most of the cases.

If the login requires sending special characters or something like that it's possible to configure the login template. This template will have the commands required by the equipment to be able to connect.

Also, after login is possible to configure a template on "Enter config mode template" to customize the session executing required commands. "Exit config mode template" and "logout template" will call to finish the session.

Finally, "verify template" will be called each to check if the session it's ok before execute any activation command.

The templates will receive the variables and values configured into the Common Configuration and also next ones: USER, PASSWORD, PASSWORD_ENABLE, HOST.



» Variables for connection templates :

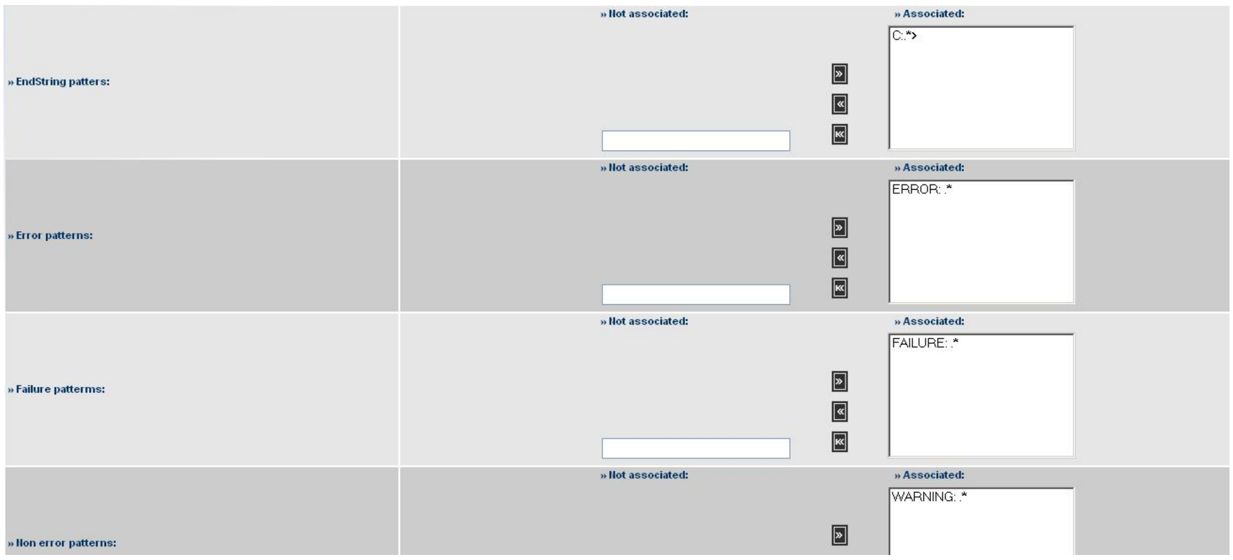
» Not associated:

» Associated:

var1	value1
var2	value2

Fig. 11: Common Conf variables

It's possible to add some patterns automatically to every command in the session. Just filling end string pattern, error pattern, failure pattern, nonerror pattern and error message.



	» Not associated:	» Associated:
» EndString patters:		C:.*>
» Error patterns:		ERROR: .*
» Failure patterms:		FAILURE: .*
» Non error patterns:		WARNING: .*

Fig. 12: Common Conf patterns

To save the Common Conf, the user must select from the “Common Conf” – “Save”

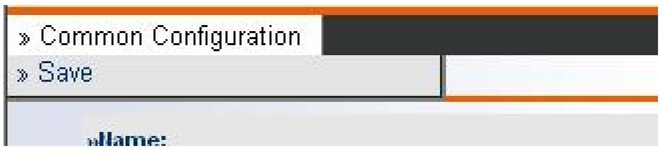


Fig. 13: Common Configuration save

To create a template to use into the Common Conf, the user must select from the Views menu "Administrator" - "ECP" - "Template" - "New".

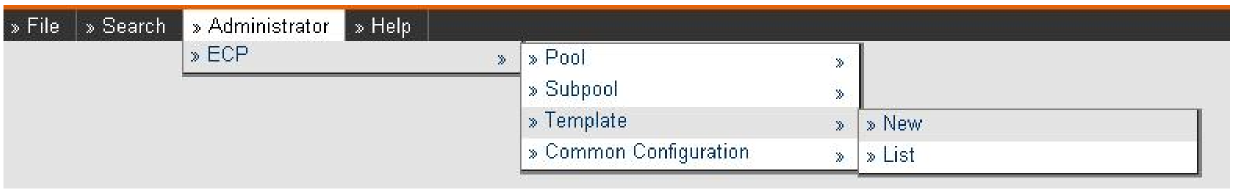
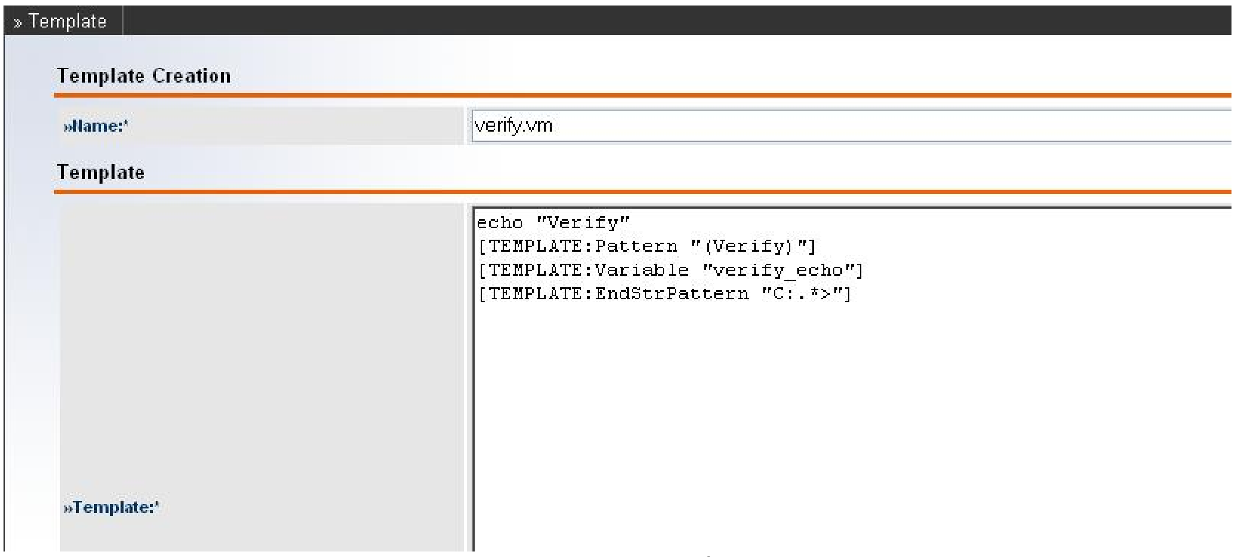


Fig. 14: Template creation

There’re just 2 field:

- Name: name of the template
- Template: the template value (commands)

In this example, a template verify it’s created. This template execute a “echo” command.



»Name:* verify.vm

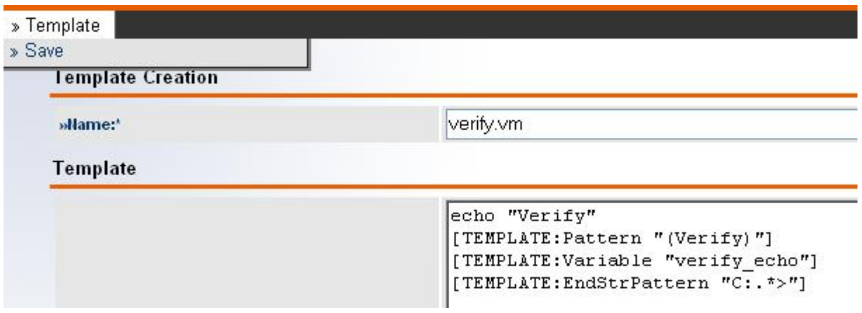
Template

»Template:*

```
echo "Verify"
[TEMPLATE:Pattern "(Verify)"]
[TEMPLATE:Variable "verify_echo"]
[TEMPLATE:EndStrPattern "C:.*>"]
```

Fig. 15: Template creation form

To save the template the user must select “Template” – “Save”.



»Name:* verify.vm

```
echo "Verify"
[TEMPLATE:Pattern "(Verify)"]
[TEMPLATE:Variable "verify_echo"]
[TEMPLATE:EndStrPattern "C:.*>"]
```

Fig. 16: Template Save

To access Subpool creation the user must select from the Views menu "Administrator" - "ECP" - "Subpool" - "New".

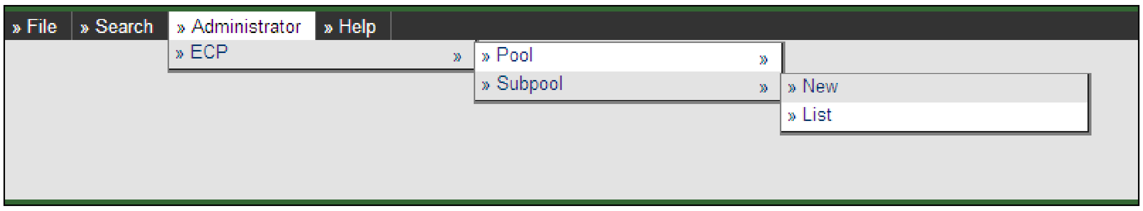
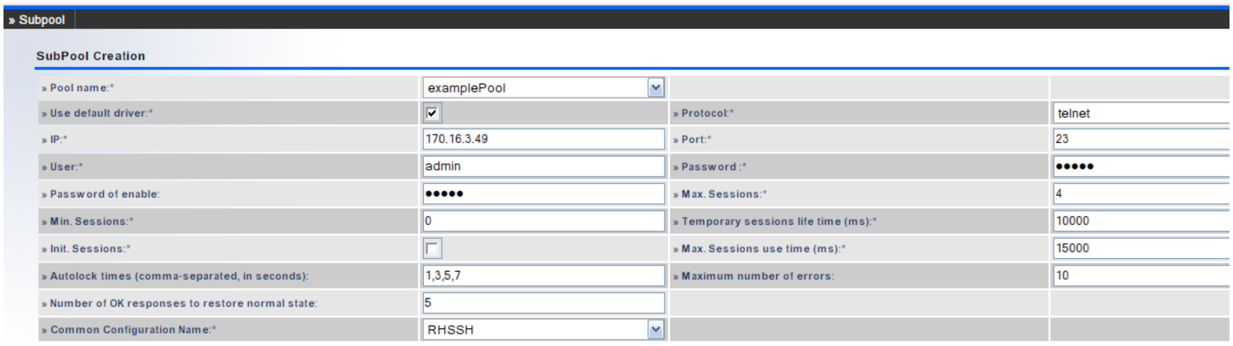


Fig. 17: View, Subpool creation

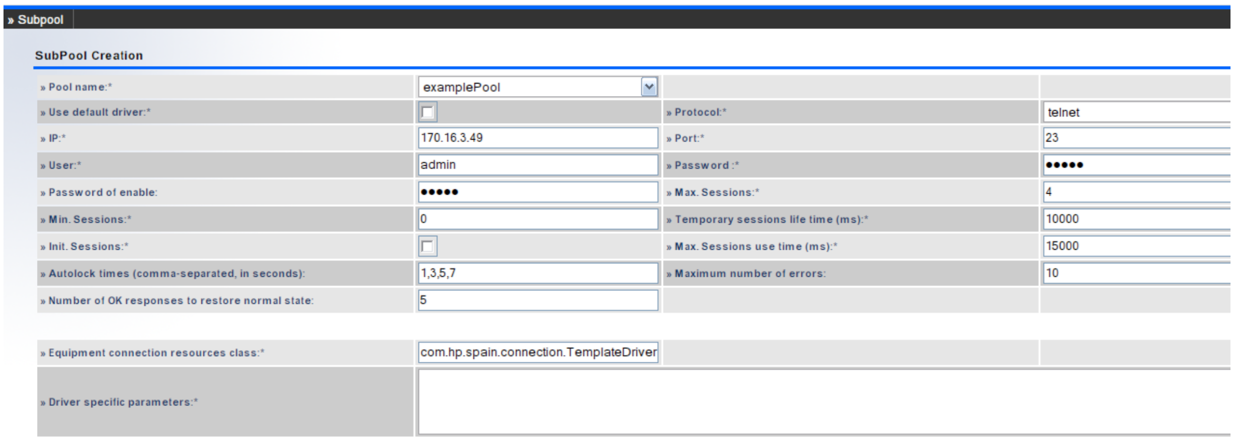
The ECP GUI will show the “SubPool Creation” screen where the new subpool information must be entered.

This form has different fields depending if the checkbox “Use default driver” is checked.



» Pool name:*	examplePool		
» Use default driver:*	✓	» Protocol:*	telnet
» IP:*	170.16.3.49	» Port:*	23
» User:*	admin	» Password :*	●●●●●
» Password of enable:	●●●●●	» Max. Sessions:*	4
» Min. Sessions:*	0	» Temporary sessions life time (ms):*	10000
» Init. Sessions:*	☐	» Max. Sessions use time (ms):*	15000
» Autolock times (comma-separated, in seconds):	1,3,5,7	» Maximum number of errors:	10
» Number of OK responses to restore normal state:	5		
» Common Configuration Name:*	RHSSH		

Fig. 18: Operation, Subpool creation default driver



» Subpool

SubPool Creation

» Pool name:*	examplePool		
» Use default driver:*	☐	» Protocol:*	telnet
» IP:*	170.16.3.49	» Port:*	23
» User:*	admin	» Password :*	●●●●●
» Password of enable:	●●●●●	» Max. Sessions:*	4
» Min. Sessions:*	0	» Temporary sessions life time (ms):*	10000
» Init. Sessions:*	☐	» Max. Sessions use time (ms):*	15000
» Autolock times (comma-separated, in seconds):	1,3,5,7	» Maximum number of errors:	10
» Number of OK responses to restore normal state:	5		
» Equipment connection resources class:*	com.hp.spain.connection.TemplateDriver		
» Driver specific parameters:*			

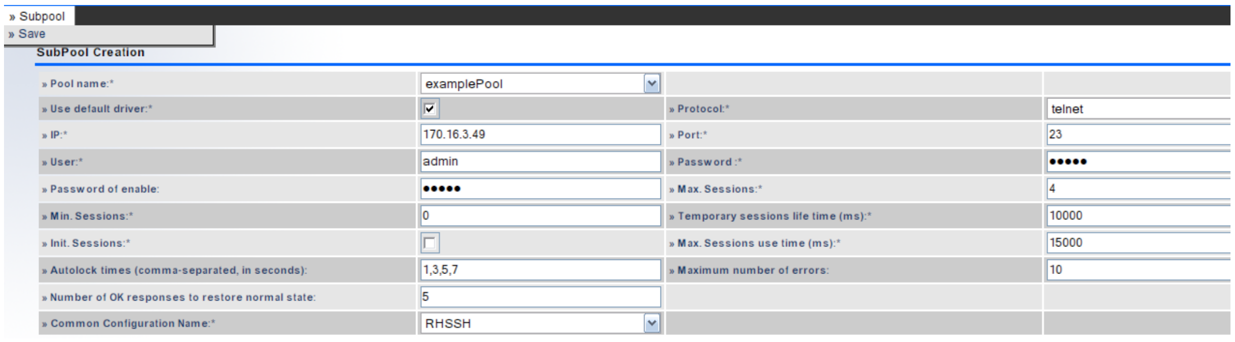
Fig. 19: Operation, Subpool creation custom driver

The example values are:

- Pool name: The name of the pool which will host the subpool. Its name must be the name of the pool created in the pool creation example: "examplePool".
- Use default driver: checkbox to set the default driver or to use a custom.
- Equipment connection resources class: The full class name of the EquipmentDriver. In this example, it will should "com.hp.spain.connection.TestConnectionResource". This field only be showed if use default driver is unchecked.
- Protocol: Will establishes the way to communicate with the equipment. This will depend on the equipment being used for the test. This example will suppose it to be "telnet".
- IP: The IP address of the destination to connect to. Will depend on the equipment being used for the test. This example will suppose it to be "172.16.2.111".
- Port: The port of the destination to connect to. Will depend on the equipment being used for the test. This example will suppose it to be "22".
- User: The username to authenticate on the destination. Will depend on the equipment being used for the test. This example will suppose it to be "root".
- Password: The login password. Will depend on the equipment being used for the test.
- Password of enable: This field will not be used. Can be left empty.
- Max. Sessions: "1" in this example.
- Min. Sessions: "1" in this example.
- Temporary sessions life time: "120000" in this example. Max life time in idle mode for sessions between min sessions and max sessions.
- Init. Sessions: "1" in this example.
- Max. Sessions use time: "200000" in this example. Maximum time that a session could be in BUSY mode (activating).
- Autolock times: a comma-separated list of times in seconds for autolocking. Leave this field blank to disable autolock for this subpool.
- Maximum number of errors: the number of erroneous responses that will trigger autolock.

- Number of OK responses to restore normal state: the number of successful responses that will reset autolock status to normal.
- DriverSpecific Parms: Special field to add particular configurations. This field only be showed if use default driver is unchecked.
- Common Conf name: the name of the common configuration in case the default driver is used. This field only be showed if use default driver is checked.

Once all fields are filled, selecting "Subpool" - "Save" from the Status menu will save the new subpool.

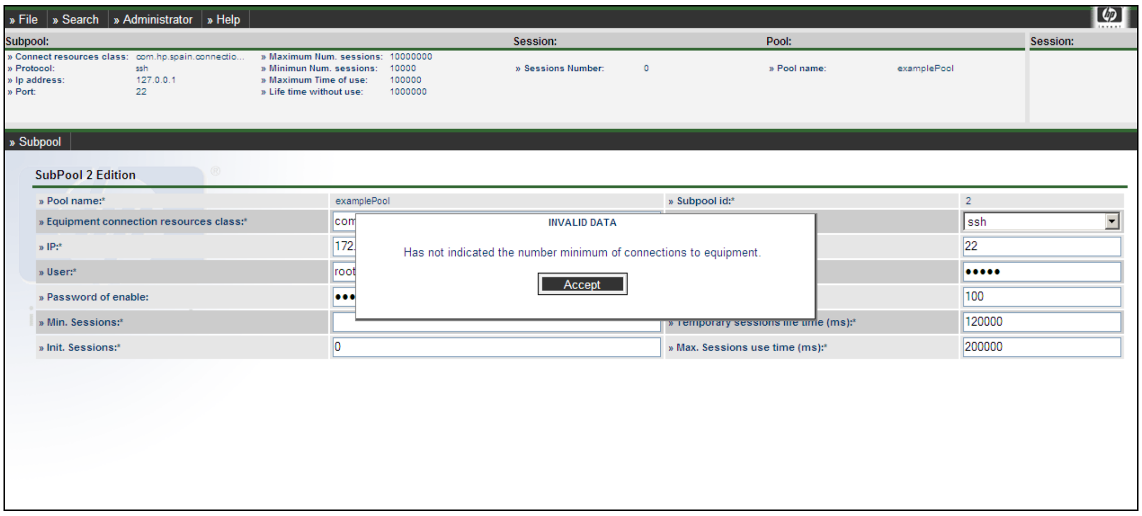


SubPool Creation

» Pool name:*	examplePool		
» Use default driver:*	✓	» Protocol:*	telnet
» IP:*	170.16.3.49	» Port:*	23
» User:*	admin	» Password :*	●●●●●
» Password of enable:	●●●●●	» Max. Sessions:*	4
» Min. Sessions:*	0	» Temporary sessions life time (ms):*	10000
» Init. Sessions:*	☐	» Max. Sessions use time (ms):*	15000
» Autolock times (comma-separated, in seconds):	1,3,5,7	» Maximum number of errors:	10
» Number of OK responses to restore normal state:	5		
» Common Configuration Name:*	RHSSH		

Fig. 20: Status, Subpool save

If any field is wrong or a mandatory field as been left empty, the ECP GUI will load a screen with a descriptive error message.



Subpool:
» Connect resources class: com.hp.spain.connectio...
» Protocol: ssh
» Ip address: 127.0.0.1
» Port: 22
» Maximum Num. sessions: 10000000
» Minimun Num. sessions: 10000
» Maximum Time of use: 100000
» Life time without use: 1000000

Session:
» Sessions Number: 0

Pool:
» Pool name: examplePool

Session:

SubPool 2 Edition

» Pool name:*	examplePool	» Subpool id:*	2
» Equipment connection resources class:*	com		ssh
» IP:*	172.		22
» User:*	root		●●●●●
» Password of enable:	●●●		100
» Min. Sessions:*		» Temporary sessions life time (ms):*	120000
» Init. Sessions:*	0	» Max. Sessions use time (ms):*	200000

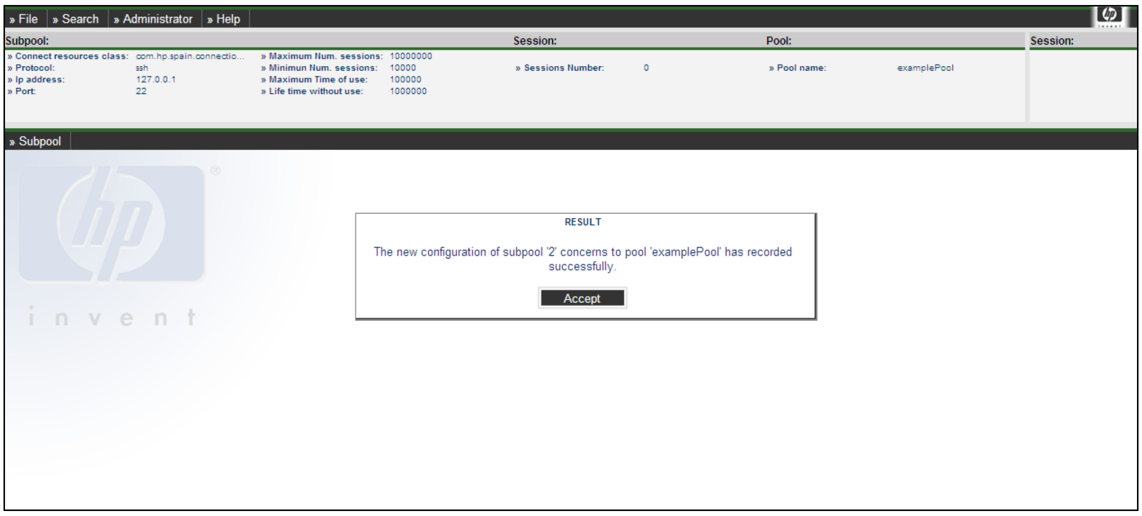
INVALID DATA

Has not indicated the number minimum of connections to equipment.

Accept

Fig. 21: Operation, Subpool save error

If the pool configuration is valid, the ECP GUI will show the following message.



Subpool:
» Connect resources class: com.hp.spain.connectio...
» Protocol: ssh
» Ip address: 127.0.0.1
» Port: 22
» Maximum Num. sessions: 10000000
» Minimun Num. sessions: 10000
» Maximum Time of use: 100000
» Life time without use: 1000000

Session:
» Sessions Number: 0

Pool:
» Pool name: examplePool

Session:

The new configuration of subpool '2' concerns to pool 'examplePool' has recorded successfully.

Fig. 22: Operation, Subpool saved

Restart the ECP and the new pool will be up and running.

4 Commands Templates

A "Commands Template" is a java String describing the commands the ECP should issue. The Command Template states the commands needed to perform a process (and usually to roll it back too), with specific information on every command, such as possible command outputs and their meaning (error, success) and the control flow which determines their execution order, among other things.

One of the main features of a Command Template is that it can be made atomic and reversible: The ECP allows the developer to include the commands needed to undo and rollback the issued commands. If and error is encountered, the ECP will start executing the specified rollback commands. Additionally, commands to trigger a commit on the destination system can be also included, if the system requires it for the changes performed to be visible and effective.

The part of a Commands Template which describes the commands to issue is called the "Do Group". The part which describes the commands to reverse the actions performed by the Do Group is called the "Undo Group". The parts which describe the commands to commit and rollback the changes are called "Commit Group" and "Rollback Group". Those four groups are the main segments of a Command Template.

The Do and Undo groups are themselves divided into numbered "Sections". Sections help organizing commands reversion. They relate a group of commands with the complementary group of commands, so that the commands in Section x of the Do Group can be undone by issuing the commands in Section x of the Undo Group.

The Commit and Rollback Groups and the Do and Undo Sections both contain Commands Statements and flow control sentences. Command Statements define how a command must be issued and its output processed. What follows is a simple Commands Template example to create/delete a user on a windows system. Lines starting with "!" are comment lines.

```
[TEMPLATE:NotUndoLastSection]
!********************************************
!******************** DO ********************
!********************************************

[TEMPLATE:Do]

!add and configure the user
  [TEMPLATE:Section 0]
      net user testUser testuserpasswd /add>nul
            [TEMPLATE:ErrorPattern *"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]
  [TEMPLATE:Section 1]
      net user testUser /homedir:c:\Users\testUser>nul
            [TEMPLATE:ErrorPattern *"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]

!create and configure the user home dir
  [TEMPLATE:Section 2]
      mkdir c:\Users\testUser
            [TEMPLATE:ErrorPattern ".*"]
            [TEMPLATE:EndStrPattern "\w\:[^\n\r]*>$"]
  [TEMPLATE:Section 3]
      ICACLS c:\Users\testUser /grant BUILTIN\Administrators:(OI)(CI)F>nul
            [TEMPLATE:ErrorPattern ".*"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]
      ICACLS c:\Users\testUser /grant testUser:(OI)(CI)(NP)F>nul
            [TEMPLATE:ErrorPattern ".*"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]
      ICACLS c:\Users\testUser /grant "NT AUTHORITY\SYSTEM":(OI)(CI)F>nul
            [TEMPLATE:ErrorPattern ".*"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]

!********************************************
!******************** UNDO ******************
!********************************************

[TEMPLATE:Undo]

!remove de user
 [TEMPLATE:Section 0]
      net user testUser /delete>nul
            [TEMPLATE:ErrorPattern *"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]
 [TEMPLATE:Section 1]

!remove de user home dir
 [TEMPLATE:Section 2]
      rmdir /s /q c:\Users\testUser>nul
            [TEMPLATE:ErrorPattern *"]
            [TEMPLATE:EndStrPattern “\w\:[^\n\r]*>$”]
 [TEMPLATE:Section 3]


[TEMPLATE:Commit]
!no commit command needed

[TEMPLATE:Rollback]
!no rollback command available
```

The same template may be executed in several ways, but the following rules always apply:

a) The commands contained in each Section and the commands in the Rollback and Commit parts are always executed in their declaration order.

b) The Sections in the Do and Undo parts are iterated in reverse order. The Sections in the Do part are always executed in their declaration order. The Sections in the Undo part are executed in the reverse order to their declaration order.

According to these rules, if no error is encountered, executing the Do part of the previous Commands Template will always result in the following commands being sent to the destination, and in this order:

```
net user testUser testuserpasswd /add>nul
net user testUser /homedir:c:\Users\testUser>nul

mkdir c:\Users\testUser

ICACLS c:\Users\testUser /grant BUILTIN\Administrators:(OI)(CI)F>nul
ICACLS c:\Users\testUser /grant testUser:(OI)(CI)(NP)F>nul
ICACLS c:\Users\testUser /grant "NT AUTHORITY\SYSTEM":(OI)(CI)F>nul
```

Executing the Undo part will always result in the following commands being sent to the destination, and in this order:

```
rmdir /s /q c:\Users\testUser>nul
net user testUser /delete>nul
```

The four different methods in which a Commands Template may be executed are `executeActivation`, `inverseActivation`, `execute` and `revert.` They allow the user to perform an action (i.e. adding a user to the system), perform the action opposite (i.e. deleting a user from the system) and grant atomicity, even if multiple Commands Templates are executed. Of course, the Commands Template must be correctly crafted.

The `executeActivation` method will try to run the Do Sections commands followed by the Commit commands. As previously described, the Sections commands will be executed in the order in which they are declared in the Commands Template and the Do Sections will be iterated in the order in which they are declared the template. If an error which can be handled is encountered during the command execution, the engine will run the commands in the Undo Sections followed by the Rollback commands. Again, following the previously stated rules, the Sections commands will be executed in the order in which they are declared in the Commands Template, and the Undo Sections will be iterated in the reverse order to their declaration order in the template, starting from same section as the failed Do Section -1.

The `inverseActivation` method will try to run the Undo Sections commands followed by the Commit commands. As previously described, the Sections commands will be executed in the order in which they are declared in the Commands Template and the Undo Sections will be iterated in the reverse order to their declaration order in the template. If an error which can be handled is encountered during the command execution, the engine will run the commands in the Do Sections followed by the Rollback commands. Again, following the previously stated rules, the Sections commands will be executed in the order in which they are declared in the Commands Template, and the Do Sections will be iterated in the order in which they are declared the template, starting from same section as the failed Undo Section +1.

The `execute` method will try to run the Do Sections commands followed by the Commit commands. As previously described, the Sections commands will be executed in the order in which they are declared in the Commands Template and the Do Sections will be iterated in the order in which they are declared the template. If an error is encountered during the command execution, the engine will end execution.

The `revert` method will try to run the Undo Sections commands followed by the Commit commands. As previously described, the Sections commands will be executed in the order in which they are declared in the Commands Template and the Undo Sections will be iterated in the reverse order to their declaration order in the template. If an error is encountered during the command execution, the engine will end execution.

In the previous Command Template example, should it fail during the execution of the Do Section 2, performing an `executeActivation` method will result in the following commands being sent to the destination, and in this order (the failed command is signaled in red):

```
net user testUser testuserpasswd /add>nul
net user testUser /homedir:c:\Users\testUser>nul

mkdir c:\Users\testUser

net user testUser /delete>nul
```

Performing an `execute` method will result in the following commands being sent to the destination, and in this order, should it fail during the execution of the Do Section 2 (the failed command is signaled in red):

```
net user testUser testuserpasswd /add>nul
net user testUser /homedir:c:\Users\testUser>nul

mkdir c:\Users\testUser
```

Also, should the previous Command Template example fail during the execution of the Undo Section 0, performing an `executeActivation` method will result in the following commands being sent to the destination, and in this order (the failed command is signaled in red):

```
rmdir /s /q c:\Users\testUser>nul

net user testUser /delete>nul
net user testUser /homedir:c:\Users\testUser>nul
mkdir c:\Users\testUser

ICACLS c:\Users\testUser /grant BUILTIN\Administrators:(OI)(CI)F>nul
ICACLS c:\Users\testUser /grant testUser:(OI)(CI)(NP)F>nul
ICACLS c:\Users\testUser /grant "NT AUTHORITY\SYSTEM":(OI)(CI)F>nul
```

Performing an `revert` method will result in the following commands being sent to the destination, and in this order, should it fail during the execution of the Do Section 0 (the failed command is signaled in red):

```
rmdir /s /q c:\Users\testUser>nul

net user testUser /delete>nul
```

5 ECP Node – ECPCall

The easiest way to execute a template using ECP is to use ECPCall node provided into the installation.

There're only two mandatory parameters. This is the simplest way to execute a template given the static pool name:

- pool_name : name of the pool to call. Could be static pool or dynamic pool
- template : template value or template file absolute path name.

There're a lof of optional parameters to use all the feature available into a template execution.

Next variables are to get outputs after execution:

- output_parameters : Variable hashmap where the output parameters will be saved.
- stdout : variable where the output of the terminal will be saved
- commands_sent : variable where the command sent will be saved
- error_miscellaneous_parameters : In case of error where hashmap misscellanous will be save
- error_message : In case of error where error message will be save
- error_action : In case of error where error action will be save
- error_command : In case of error where error command will be save

Next variables to define where ecp is running:

- ecp_host : the ip address where ecp is running. By default 127.0.0.1
- ecp_port : the port address where ecp is listening. By default 1200

Next variables to execute velocity parser:

- compose_template: (true/false) Setting this parameter to true the node will compose the template. Also, if compose_paramters is not null.
- compose_parameters : In case the template has to be composed (velocity parsed), the hashmap with the variables. If this variable is not null then the node will compose the template.
-

Next variables to define the type of execution:

- initial_section : Number of the initial section
- priority : Priority number of this execution
- revert : (true/false) Call revert action instead of executeActivation
- inverse : (true/false) Call inverseActivation action instead of executeActivation
- execute : (true/false) Call execute action instead of executeActivation

Next variables is to define a dynamic call:

- dynamic_ip : The target ip address in case of connection by dynamic pool. If this variable is setted then the call will be dynamic.
- dynamic_port : The target port in case of connection by dynamic pool.
- dynamic_protocol : Type of protocol (telnet, tcp, ssh, ssh_deprecated, …) in case of connection by dynamic pool.
- dynamic_user : User name in case of connection by dynamic pool.
- dynamic_password : Password in case of connection by dynamic pool.
- dynamic_password_enable : Password enable in case of connection by dynamic pool.
- dynamic_resource_class_name : The resource class name in case of connection by dynamic pool. Default value is com.hp.spain.connection.TemplateDriver
- dynamic_specific_params : Specific params in case of connection by dynamic pool.
- dynamic_maximum_connection : Maximum connections allowed in case of connection by dynamic pool.
- dynamic_minimum_connection : Minimum connections in case of connection by dynamic pool.
- dynamic_init_on_create : Initialize sesions on create in case of connection by dynamic pool.
- dynamic_temporary_resource_timeout : Temporary resoucer timeout in milisegonds in case of connection by dynamic pool.
- dynamic_resource_timeout : Resource timeout in milisegonds in case of connection by dynamic pool.
- dynamic_not_used_maxtime_life : Not used maximum time in milisegonds of this pool in case of connection by dynamic pool.

In case a dynamic call is used and dynamic_resource_class_name is setted to com.hp.spain.connection.TemplateDriver, next variables is to configure the TemplateDriver:

- driver_common_configuration_name: the name of the common configuration (only for TemplateDriver)
- driver_add_patterns_connection_templates : add patterns to connection templates (only for TemplateDriver)
- driver_endstring_pattern : end string patterns that will be added to all commands (only for TemplateDriver)
- driver_error_pattern : error patterns that will be added to all commands (only for TemplateDriver)
- driver_error_message : error message that will be added to all commands(only for TemplateDriver)
- driver_failure_pattern : failure patterns that will be added to all commands(only for TemplateDriver)
- driver_nonerror_pattern : nonerror patterns that will be added to all commands(only for TemplateDriver)
- driver_login_template : login template file path or name templates saved into database (configured by web administration)(only for TemplateDriver)

- driver_logout_template : logout template file path or name templates saved into database (configured by web administration)(only for TemplateDriver)
- driver_enter_config_mode_template : enter config mode template file path or name templates saved into database (configured by web administration)(only for TemplateDriver)
- driver_exit_config_mode_template : exit config mode template file path or name templates saved into database (configured by web administration)(only for TemplateDriver)
- driver_verify_template : verify template file path or name templates saved into database (configured by web administration)(only for TemplateDriver)
- driver_login_user_prompt : login user prompt(only for TemplateDriver)
- driver_login_pwd_prompt : login password prompt(only for TemplateDriver)
- driver_inital_prompt : initial prompt(only for TemplateDriver)

Next variables are used to save into ddbb each execution:

- save_ecp_action_by_jms : If true jms message will be listen to save the ECP action on database. Default false
- job_id : job_id variable. Default value is JOB_ID
- service : In case save_ecp_action_by_jms is true, service value will be saved into database into the ECP action
- action : In case save_ecp_action_by_jms is true, action value will be saved into database into the ECP action
- ecp_jms_module : Name of the jms module. Default value is EcpJmsModule

To call the ECP in background and the free the worked, the node will be waiting in askfor. It's mandatory to configure the background module into the mwfm.xml:

- background_call: If true tha activation will be done by BackgroundCallModule in background.
- background_call_module: Name of the BackgroundCallModule. Default EcpBackgroundModule.

6 ECP Plug-in for HPSA

The example which follows is an ECP plug-in which may execute (doing or undoing) an arbitrary Commands Template using a connection from an arbitrary pool. There're two task one for static and another for dynamic. These tasks receive a parameter named template. This template cannot be a file path name else the content. This plugin doesn't make the compose of velocity. Then, there's a node called ECPParseTemplate. This node has next paremeters:

- template : Template value or template file path.
- compose_parameters : In case the template has to be composed (velocity parsed), the hashmap with the variables. If this variable is not null then the node will compose the template.
- compose_template : (true/false) Setting this parameter to true the node will compose the template. Default value, true.
- generated_template : The variable where parsed template will be saved.

Next there's the plugin example:

```
package com.hp.spain.connection.plugin;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.HashMap;
import java.util.Properties;

import com.hp.ov.activator.resmgr.ExecutionDescriptor;
import com.hp.ov.activator.resmgr.par.PARPlugin;
import com.hp.ov.activator.resmgr.par.PluginException;
import com.hp.ov.activator.util.AttributeTable;
import com.hp.spain.connection.CLICommands;
import com.hp.spain.connection.CLIConstants;
import com.hp.spain.connection.CLIExecutionException;
import com.hp.spain.connection.TemplateDriver;
import com.hp.spain.connection.TemplateParser;
import com.hp.spain.connection.TemplateParserException;
import com.hp.spain.connection.pool.DynamicEcpProperties;

/**
 * <p>
 * ecp plugin
 *
 * @preprov <i>General pre-provisioning information</i>
 * @platform <i>Platform here</i>
 *
 * @author   HP OpenView Service Activator ServiceBuilder
 * @version  1.0.0
 * (c) Copyright 2007 Hewlett-Packard Development Company, L.P.
 */
public class ECP extends PARPlugin
{
  public static final String ECP_SERVICE_IP = "ECP_SERVICE_IP";
  public static final String ECP_SERVICE_PORT = "ECP_SERVICE_PORT";

  private String ecpServiceIp = null;

  private String ecpServicePort = null;

  public void init(AttributeTable config) throws PluginException {
       super.init(config);

       if (config.containsAttribute(ECP_SERVICE_IP)){
             this.ecpServiceIp = config.getAttribute(ECP_SERVICE_IP);
       }
       if (config.containsAttribute(ECP_SERVICE_PORT)){
             this.ecpServicePort = config.getAttribute(ECP_SERVICE_PORT);
       }
  }

  public void destroy() {
  }

  /**
   * <p>
   *
   * @param poolName
   * @param template
   * @param outputParamsVariable
   *
   * @do_and_check   <i>DO_AND_CHECK description</i>
   * @undo_and_check <i>UNDO_AND_CHECK description</i>
   * @preprov <i>Pre-provisioning requirements</i>
   * @warning <i>Additional warnings</i>
   * @dependency <i>Dependencies with other elements</i>
   */
  public ExecutionDescriptor task_staticCall (int op, String poolName, String
template) throws PluginException
  {
             HashMap hmVariables = new HashMap();
       CLICommands cliCommands = null;

       TemplateParser parser = new TemplateParser();

       try {
             cliCommands = parser.parseTemplate(template);
             if (ecpServiceIp!=null && !"".equals(ecpServiceIp))
cliCommands.setRMIHostName(ecpServiceIp); // IP de ECP
             if (ecpServicePort!=null &&
!"".equals(ecpServicePort))cliCommands.setRMIPort(ecpServicePort); // Puerto
de ECP

             switch (op) {
                   case DO_AND_CHECK:
                         hmVariables =
cliCommands.executeActivation(poolName);
                         break;
                   case UNDO_AND_CHECK:
                         hmVariables = cliCommands.revert(poolName);
                         break;
                   default:
                        throw new PluginException("Operation not
supported");
            }
            context.uploadData("ecp_outputparams", hmVariables);
            return new ExecutionDescriptor(ExecutionDescriptor.OK,
ExecutionDescriptor.NONE, "template executed", "", "");

      } catch (CLIExecutionException cliee) {
            if (cliee.getAction().equals(CLIConstants.ROLLBACK_ERROR) ||
cliee.getAction().equals(CLIConstants.IO_ERROR)
                || (cliee.getAction().equals(CLIConstants.CONNECTION_ERROR)
&& cliee.getSectionNumber() != null)) {
                  return new ExecutionDescriptor(ExecutionDescriptor.ERROR,
ExecutionDescriptor.INCONSISTENT, cliCommands.getStdOut(),
cliee.getMessage(), cliee.getCommand()+ "\n" + cliee.getMiscellaneous());
            } else {
                  return new ExecutionDescriptor(ExecutionDescriptor.ERROR,
ExecutionDescriptor.CONSISTENT, cliCommands.getStdOut(), cliee.getMessage(),
cliee.getCommand()+ "\n" + cliee.getMiscellaneous());
            }
      } catch (TemplateParserException e) {
            return new ExecutionDescriptor(ExecutionDescriptor.ERROR,
ExecutionDescriptor.CONSISTENT, e.getMessage(), e.getMessage(),
e.getMessage());
      }

  }

  /**
   * <p>
   *
   * @param poolName
   *
   * @do_and_check   <i>DO_AND_CHECK description</i>
   * @undo_and_check <i>UNDO_AND_CHECK description</i>
   * @preprov <i>Pre-provisioning requirements</i>
   * @warning <i>Additional warnings</i>
   * @dependency <i>Dependencies with other elements</i>
   */
  public ExecutionDescriptor task_dynamicCall(int op, String poolName, String
template,
            String driverProtocol, String driver, String ip, String port,
String user, String password, String passwordEnable, String
driverSpecificParams,
            String maxConnections, String minConnections, String
initOnCreate,
            String temporaryResourceTimeout, String reourceTimeout, String
notUsedMaxTime, String driverCommonConfigurationName) throws PluginException
{

      HashMap hmVariables = new HashMap();
      CLICommands cliCommands = null;

      TemplateParser parser = new TemplateParser();
      DynamicEcpProperties dynamicEcpProperties = null;
      try {
            if (driver == null || "".equals(driver)){
                  driver = "com.hp.spain.connection.TemplateDriver";
            }
            boolean isDefaultDriver = false;
            if ("com.hp.spain.connection.TemplateDriver".equals(driver)){
                  isDefaultDriver=true;
            }

            if (driverProtocol != null &&
driverProtocol.toLowerCase().equals("ssh_deprecated")){
                  driverProtocol="ssh";
            }else if (driverProtocol == null ||
driverProtocol.toLowerCase().equals("ssh")){
                  driverProtocol="sshex";
            }

            dynamicEcpProperties = new DynamicEcpProperties(poolName,
driverProtocol, driver, ip , new Integer(port).intValue()
                        , user, password, passwordEnable);

            dynamicEcpProperties.setPoolConfiguration(poolName, new
Integer(maxConnections).intValue(),
                        new Integer(minConnections).intValue(),
"true".equalsIgnoreCase(initOnCreate),
                        new Integer(temporaryResourceTimeout).intValue(),
new Integer(reourceTimeout),
                        new Integer(notUsedMaxTime));

            if (isDefaultDriver){
                  Properties properties = new Properties();
                  if (driverSpecificParams!=null &&
!"".equals(driverSpecificParams)){
                        try {
                              properties.load(new
ByteArrayInputStream(driverSpecificParams.getBytes()));
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                  }
                  properties.put(TemplateDriver.COMMON_CONFIGURATION_NAME,
driverCommonConfigurationName);
            ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
            try {
                        properties.store(outputStream, "Template Driver
parameters");
                  } catch (IOException e) {
                  }

  dynamicEcpProperties.setSpecificParameters(outputStream.toString());

            }else{

  dynamicEcpProperties.setSpecificParameters(driverSpecificParams);
            }

            cliCommands = parser.parseTemplate(template);
            if (ecpServiceIp!=null && !"".equals(ecpServiceIp))
cliCommands.setRMIHostName(ecpServiceIp); // IP de ECP
            if (ecpServicePort!=null &&
!"".equals(ecpServicePort))cliCommands.setRMIPort(ecpServicePort); // Puerto
de ECP
```

```
            switch (op) {
                  case DO_AND_CHECK:
                        hmVariables =
cliCommands.executeActivation(dynamicEcpProperties);
                        break;
                  case UNDO_AND_CHECK:
                        hmVariables =
cliCommands.revert(dynamicEcpProperties);
                        break;
                  default:
                        throw new PluginException("Operation not
supported");
            }
            context.uploadData("ecp_outputparams", hmVariables);
            return new ExecutionDescriptor(ExecutionDescriptor.OK,
ExecutionDescriptor.NONE, "template executed", "", "");

      } catch (CLIExecutionException cliee) {
            if (cliee.getAction().equals(CLIConstants.ROLLBACK_ERROR) ||
cliee.getAction().equals(CLIConstants.IO_ERROR)
                || (cliee.getAction().equals(CLIConstants.CONNECTION_ERROR)
&& cliee.getSectionNumber() != null)) {
                  return new ExecutionDescriptor(ExecutionDescriptor.ERROR,
ExecutionDescriptor.INCONSISTENT, cliCommands.getStdOut(),
cliee.getMessage(), cliee.getCommand()+ "\n" + cliee.getMiscellaneous());
            } else {
                  return new ExecutionDescriptor(ExecutionDescriptor.ERROR,
ExecutionDescriptor.CONSISTENT, cliCommands.getStdOut(), cliee.getMessage(),
cliee.getCommand()+ "\n" + cliee.getMiscellaneous());
            }
      } catch (TemplateParserException e) {
            return new ExecutionDescriptor(ExecutionDescriptor.ERROR,
ExecutionDescriptor.CONSISTENT, e.getMessage(), e.getMessage(),
e.getMessage());
      }

  }
}
```

The execution of a Commands Template involves a parser instance and an engine instance. The Commands Template must be first parsed using a `TemplateParser` class instance and calling its method `parseTemplate(String sTemplate)`, which receives the template as a `String`. As a result of the parsing process, the TemplateParser will either throw a `TemplateParserException` if the parsing was not possible, or return a `CLICommands` engine instance.

If the parsing is successful, the returned CLICommands instance may be used to execute the Commands Template. The four Commands Template execution methods are available through the functions `executeActivation(String sPoolName)`, `inverseActivation(String sPoolName)`, `execute(String sPoolName)` and `revert(String sPoolName)`. The developer may usually recognize whether the execution has not been atomic by checking the exception action type and the failed section. The exception action type indicates the part of the Commands Template where the error referred by the Exception occurred.

If the error is an `IOException` which is encountered during Do, Undo, Commit or Rollback, as `IOException` errors cannot be handled, the special action type `CLIConstants.IO_ERROR` is declared by the exception. This error will usually indicate (depending on the specific Commands

Template) an inconsistent execution, that is, atomicity could not be provided and the plug-in should return an `ExecutionDescriptor.INCONSISTENT` result.

If a non `IOException` error is encountered during error handling, the action type `CLIConstants.ROLLBACK_ERROR` is declared by the exception. Again, this error will usually indicate (depending on the specific Commands Template) an inconsistent execution, that is, atomicity could not be provided and the plug-in should return an `ExecutionDescriptor.INCONSISTENT` result.

If a non `IOException` error is encountered during error handling, the action type `CLIConstants.ROLLBACK_ERROR` is declared by the exception. Again, this error will usually indicate (depending on the specific Commands Template) an inconsistent execution, that is, atomicity could not be provided and the plug-in should return an `ExecutionDescriptor.INCONSISTENT` result.

If a SocketException error is encountered, the action type `CLIConstants.CONNECTION_ERROR` is declared by the exception. This error can't be handled by the ECP, and as a consequence if this error is thrown on Do, Undo, Commit or Rollback no error handling is performed and as a consequence atomicity can't be provided and (depending on the specific Commands Template) the plug-in should return an `ExecutionDescriptor.INCONSISTENT` result. To identify if the error has been thrown on Do, Undo, Commit or Rollback, the user should check whether a section is present in the error.