vFabric Hyperic HQU Plug-in Development

VMware vFabric Hyperic 5.0

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-000964-00

You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

Copyright © 2013 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Table of Contents

1 About vFabric Hyperic HQU Plug-in Development 6

Intended Audience 6

2 HQU Overview 7

What is HQU? 7

What is HQU Good For? 7

HQU Plugins and the HQ UI 7

Overview of HQU Plugin Development 7

3 HQU Anatomy and Components 9

HQU Design Pattern 9

Plugin Files 9

Plugin Directories 10

Key HQU Framework Files 10

Plugins and the HQ Portal 11

HQU Deployment 11

Invoking an HQU Plugin as a Web Service 11

HQU Request Processing 11

4 Planning an HQU Plugin 13

Plugin Mission and Type 13

Access and Authorization 13

Required HQ Data and Functions 13

User Supplied Parameters 13

View Plugin Requirements 13

Integration Plugin Requirements 14

Automation Plugins 14

5 Developing an HQU Plugin 15

Step 1 - Create Plugin Scaffolding 15

Step 2 - Build the Plugin 17

Step 3 - Deploy and Run the Plugin 18

Step 4 - Set Location Properties for a View Plugin 18

Step 4 - Program the Plugin Behavior 20

Step 5 - Localize the Plugin 20

6 Writing Plugin Controller Logic 22

Accessing HQ Data and Functions 22

Defining User Interaction 23

Rendering 24

Miscellaneous Controller Topics 26

7 Groovy Tips 27

Links to Groovy Topics 27
Groovy def 28
Property Accessor Methods 28
Accessibility Modifiers 28
Inferred Types 29

8 Plugin Scaffolding 30

Plugin.groovy 30
ControllerName Controller.groovy 30
index.gsp 31
plugin_name _i18n.properties 31
plugin.properties 31

9 HQ Groovy APIs 32

Coding Conventions 32
Obtaining a Helper Instance 32
BaseController Utility Methods 32
MetaClasses and Categories 33
ResourceHelper.groovy 33
AlertHelper.groovy 34
AuditHelper.groovy 34
BaseHelper.groovy 34
MetricHelper.groovy 34
AgentHelper.groovy 34

10 Sample Plugin 35

Currently Down View 35
SystemsdownController.groovy 35

- About vFabric Hyperic HQU Plug-in Development (see page 6)
- HQU Overview (see page 7)
- HQU Anatomy and Components (see page 9)
- Planning an HQU Plugin (see page 13)
- Developing an HQU Plugin (see page 15)
- Writing Plugin Controller Logic (see page 22)
- Groovy Tips (see page 27)
- Plugin Scaffolding (see page 30)
- HQ Groovy APIs (see page 32)
- Sample Plugin (see page 35)

About vFabric Hyperic HQU Plug-in Development

1

vFabric Hyperic HQU Plug-in Development is a guide to using the HQU Framework to extend the VMware® vFabric™ Hyperic® and Hyperic HQ user interface and to implement extended management functionality.

Intended Audience

This guide is intended for Groovy developers that are knowledgeable about Hyperic internals.

HQU Overview 2

These topic briefly describe HQU functionality and usage.

What is HQU?

HQU is a framework within HQ for creating and deploying extensions to HQ. It provides tools and APIs for extending the HQ Portal, integrating HQ with other systems, and automating HQ administrative tasks. The HQU framework is based on Groovy, a scripting language with syntax similar to Java. Groovy runs in the JVM and can interact with regular Java classes and libraries.

What is HQU Good For?

HQU helps you tailor HQ to your environment and work flows. You can use it to:

- Extend the HQ Portal—HQU Plugins can add Views to the HQ user interface. HQU has been used to develop some of the regular HQ Portal pages. Examples include the Alert Center, a single page view of triggered alerts across an HQ deployment; the Event Center, a filterable view of HQ events such as log entries, configuration changes, and Alerts; and the Systems Down page.
- Automate HQ functions—HQU plugins can automate HQ functions that you perform frequently or in high volume. For example, the HQU Mass plugin allows you to list or approve multiple Platforms in the auto-discovery queue, using regular expressions.
- Integrate HQ with other systems—HQU plugins can expose Web services for exchanging data with other systems. For example, the HQU OpenNMS plugin enables export of Platform and Alert data from HQ to OpenNMS. HQU Web services plugins can use the XML or the JavaScript Object Notation (JSON) interchange format, useful if the consumer of the web services is a browser.

Other HQU plugins that ship with HQU include the the Groovy Console, HQ Health page, and LiveExec Views.

HQU Plugins and the HQ UI

As desired, an HQU plugin can be made available from several locations in the HQ Portal:

- The View tab for a Resource
- As items on the Resources and Analyze menus
- In the Plugins section of the Administration page

Overview of HQU Plugin Development

Creating an HQU plugin involves these basic steps.

1. Generate the plugin files. HQU provides a scaffold tool that creates a plugin directory structure with the files required for a functional plugin.
2. Program the core plugin functionality. HQU provides helper APIs for accessing HQ data and functions. You can use the Groovy Console in the HQ Portal to execute Groovy code and HQ methods within the context of the HQ Server.
3. Plug it in. If your plugin is designed to render content to a browser, you integrate it into the HQU Portal user interface by specifying its attachment point.
4. Deploy the plugin. Add the plugin to the HQ Portal web application (HQ.war); it will be automatically hot-deployed. You can use the HQU plugin manager for debugging and for removing plugin Views from HQ Portal pages.

HQU Anatomy and Components 3

This section describes how HQU plugins fit into, and operate with, the rest of your HQ environment.

HQU Design Pattern

Like HQ, HQU is based on a Model View Controller (MVC) design pattern. In the HQU framework, MVC functionality is distributed in this fashion:

- Model—The model, which represents HQ data and the rules for accessing and updating it, consists of a set of HQ Groovy APIs. The Groovy APIs provide the HQ data for the plugin's View, JSON, or XML output. Groovy APIs provide simple access to backend functionality.
- View—For an HQU plugin that extends the HQ UI, the view that renders the model is one or more Groovy Server Pages (GSPs). For an integration plugin, the view is the plugin's XML or JSON output.
- Controller—The controller is a Groovy script.

Plugin Files

This table describes the required components of an HQU Plugin.

Plugin.groovy

Plugin.groovy is the class that HQ uses to interact with the plugin. This file declares the plugin's default controller and the default action (method) to invoke when the plugin is run. If your plugin extends the HQ Portal user interface, you define the attachment location in Plugin.groovy.

plugin.properties

Descriptor that HQ uses to identify the plugin. It contains the plugin name, plugin version number, and the HQ version that supports the plugin.

Controller_Name Controller.groovy

The plugin controller is a Groovy script. A controller file name is the concatenation of the name you assign the controller, the string "Controller", and the file extension .groovy.
The controller accepts user requests, generates the view (in .gsp, xml, or json format), and instructs the Groovy APIs and the view to perform actions in response to user input. The controller translates interactions (GET and POST HTTP requests) with the view into actions to be performed using the Groovy APIs. Based on user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.
A plugin developer defines the plugin behavior in this file. The plugin controller extends BaseController.groovy, which imports the Groovy helper APIs—enabling access to each helper API via its getter method. The base controller is invoked by the dispatcher when it detects that a controller method is being requested.
A plugin must have at least one controller, and may have multiple controllers.

index.gsp

For plugins that extend the HQ Portal, this is the default plugin view—a Groovy Server Page. It is created by the controller, and renders the the model. Analogous to a .jsp, a .gsp contains HTML, Javascript, and Groovy control elements; it can use variables passed by the plugin controller to generate dynamic portions of the view. Views can use HQU utility methods to generate tables and provide AJAX support.
The view, responsible for delivery or presentation of HQ data as an HTML response to the user, takes the form of a Groovy Server Page (GSP). A GSP contains HTML and Groovy control elements. In an integration plugin, the view would be the plugin's XML or JSON output.
An HQU plugin may have multiple views.

plugin_name _i18n.properties

This file contains localization strings. While not strictly required, many parts of the framework use it frequently. It is used by Controllers and Views to display localized text and provides the localized name of the plugin to be displayed at attach points.

Plugin Directories

hq
/ui_plugins
/***plugin*** home directory for the plugin, with same name as the plugin
/app plugin controller file(s)
/etc localization file
/lib non-HQ Java support libraries that the plugin uses, as applicable. JAR files in this directory will automatically be added to the plugin's classpath.
/public Additional HTML, images, and CSS that the plugin depends on
/views GSPs
/templates optional templates (*.gsp) of markup that you wish to reuse for different actions

Key HQU Framework Files

HQU Framework files are found in the hq-rendit module. Some of the key files that are relevant to plugin development are shown below.

File	Purpose

BaseController.groovy	The base controller is invoked by the dispatcher when it detects that a controller method is being requested. BaseController imports the Groovy helper APIs, and a plugin controller extends BaseController, enabling access to each helper API via its getter method.
DojoUtil.groovy	Provides utility methods for creating complex content which interacts with HQ.
RenderFrame.groovy	Renders output to the browser.
HQUPlugin.groovy	The plugin deployer, deals with requests from the HQ. It can respond to queries for the plugin name, descriptions, and supported HQ versions.

Plugins and the HQ Portal

These are the places where you can make a View plugin available in the HQ Portal.

- Masthead menus—You can make a plugin available as an item on either the Resources or the Analyze menu. Plugins can emit any HTML to render the menu item (meaning they can be dynamic). Views attached to the masthead display content in the area below the orange bar under the Masthead
- Administration Attach Point—There is a section within the Administration tab entirely devoted to plugins. Like masthead attachments, Administration attachments are viewed under the orange bar.
- Resource Attach Point—Resource attach points are special because they exist within the context of a resource. Attaching here makes sense when a plugin needs to provide particular screens for resource types (such as a custom SQL tool which interacts with your Oracle instance, or a real-time log viewer for Apache). HQ passes this context information to the plugin for easy access.

HQU Deployment

Copy your HQU plugin to:
TOMCAT_HOME/webapps/ROOT/*hqu.*

A directory watcher periodically checks this directory for new additions and removals and loads and unloads plugins as necessary. Based on the attachment method you choose, HQ will automatically create a item in the appropriate menu, or in the View tab of a resource.

Invoking an HQU Plugin as a Web Service

HQU plugins can be invoked as a web service, via a regular HTTP request:

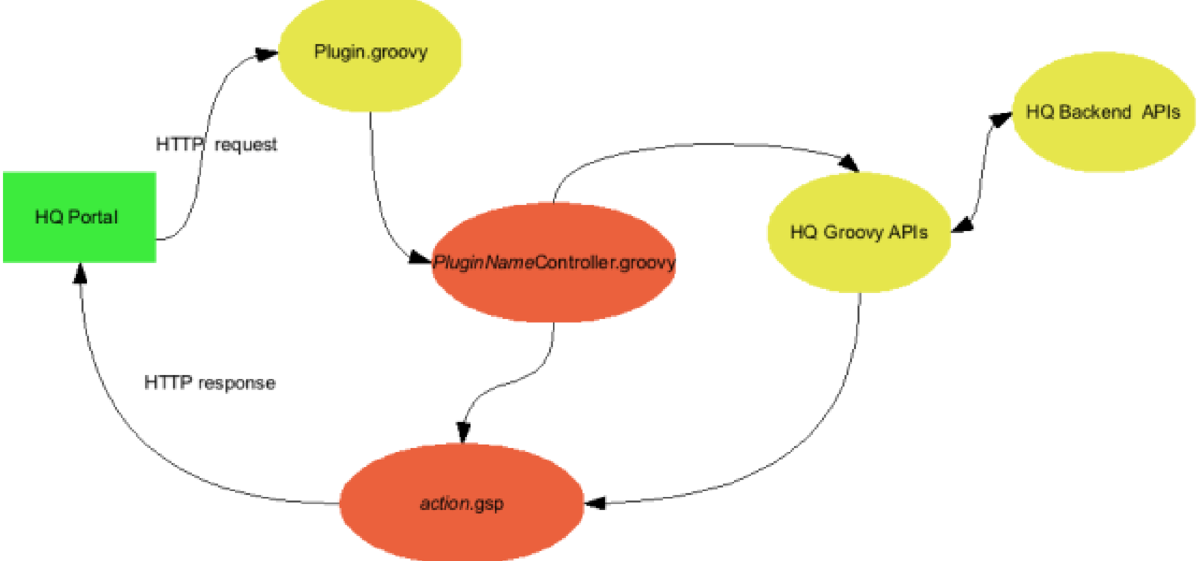
```
curl https://hqadmin:hqadmin@localhost:7443/hqu/plugin_name/Controller_Name/Action.hqu
```

where:

- 7443 is the port where Server listens for https traffic.
- hqadmin is the HQ root user, the original HQ administrator.
- hqadmin@localhost is the "From" address on alert notification emails sent from HQ. Note that most mail servers will not deliver mail without a valid domain name in the From field.
- *plugin_name* is the the plugin name
- *Controller_Name* is the controller name
- *Action* is the Action, or method, in the controller to invoke

HQU Request Processing

The diagram below illustrates HQU plugin components in the context of a request-response cycle, given user request of a plugin that is a View attached to the masthead.



1. When user selects the plugin from a Masthead menu, an internal URL is issued:

```
http://localhost:7080/mastheadattach.do?typeID=nnnnn
```

2. HQU looks up the view, and calls the plugin controller index action:

```
http://localhost:7080/hqu/plugin_name/Controller_Name/index.hqu
```

3. The controller object is instantiated. It renders the view, index.gsp, calling the render() method provided by RenderFrame.groovy.
4. The view uses the HQ Groovy APIs to obtain the HQ data to present, and provides the HTTP response to the Portal user.

Note: A controller object is instantiated only for the lifetime of a request. Each new request gets a new instance of the controller. Local variables within the controller only exist for the duration of the request.

Planning an HQU Plugin

4

- The topics in this section are intended to give a new HQU plugin developer a starting point for defining the requirements for an HQU plugin, and for mapping HQU capabilities to those requirements.

WIP Note: This section is a work in progress. In the next version of this guide, these topics will be enhanced, and cross-referenced to related "how to" topics.

Plugin Mission and Type

What is the purpose of your plugin? Is it a:

- View plugin that adds one or more new pages to the HQ Portal user interface?
- Integration plugin that creates or consumes an export file, for example one that defines inventory resources, such such as Platform?
- Automation plugin that automates an HQ inventory or administration task?

A plugin can combine these capabilities. For instance, a plugin might consume an XML import file, automatically create resources, and present a new view in the Portal user interface.

Access and Authorization

- What user roles can use the plugin? Do the plugin requirements vary by user or role?
- Do current HQ Roles support the plugin's access control requirements?

Required HQ Data and Functions

- What HQ data do you need to access or modify? This will determine what API methods you need to use.
- What filtering options are required?
- Do current Groups support the plugin's access control requirements?
- Any manipulation/processing to be performed on the data returned?

User Supplied Parameters

- What parameters can the user supply when running the plugin?

View Plugin Requirements

- View Layout—what should the presented page look like?
- What components/widgets will the view contain? Tables, graphical presentation, static content?
- Where will it attach to the Portal user interface?

- What text string will you use for the plugin menu option or link presented in the Portal?

Integration Plugin Requirements

- Does the plugin export or consume HQ data?
- Should it produce XML or JSON?
- What structure must be produced?
- Will the plugin be accessible from the Portal?

Automation Plugins

- What are the tasks to be automated?

Developing an HQU Plugin

5

- Step 1 - Create Plugin Scaffolding (see page 15)
  - Scaffolding Syntax and Arguments (see page 15)
  - Files and Directories Created by Scaffolding Task (see page 16)
  - Contents of the Generated Plugin Files (see page 17)
- Step 2 - Build the Plugin (see page 17)
- Step 3 - Deploy and Run the Plugin (see page 18)
- Step 4 - Set Location Properties for a View Plugin (see page 18)
  - Excerpt from Plugin.groovy as Created (see page 18)
  - Understanding Attachment Properties and Options (see page 18)
  - Specify View Location in Plugin.groovy (see page 19)
    - Attach Plugin to a Masthead Menu (see page 19)
    - Attach Plugin to Administration Page (see page 19)
    - Attach Plugin to Resource>View page (see page 19)
  - Detaching and Removing HQU Plugins (see page 20)
- Step 4 - Program the Plugin Behavior (see page 20)
- Step 5 - Localize the Plugin (see page 20)
  - Contents of the I18n Properties as Created (see page 20)
  - Understanding How Plugin Views Obtain Localization Strings (see page 20)

This section has instructions for the key steps in developing an HQU plugin.

Step 1 - Create Plugin Scaffolding

You must have access to Hyperic source code to run the scaffolding task.

Generate the directory structure and the required component files for the plugin using the scaffold maven archetype in the hq source directory.

Scaffolding Syntax and Arguments

To generate a new plugin using the scaffold archetype:

```
$ cd hq-uiplugin
$ mvn archetype:update-local-catalog
$ mvn archetype:generate -DarchetypeCatalog=local
[INFO] No archetype defined. Using maven-archetype-quickstart
(org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
1: local -> scaffold-archetype (scaffold-archetype)
Choose a number: : 1
Define value for property 'groupId': : org.myco
Define value for property 'artifactId': : cool
Define value for property 'version': 1.0-SNAPSHOT:
Define value for property 'package': org.myco
Define value for property 'controller': : Freezer
Define value for property 'controllerDir': : freezer
Confirm properties configuration:
groupId: org.myco
artifactId: cool
version: 1.0-SNAPSHOT
package: org.myco
controller: Freezer
controllerDir: freezer
Y: Y
```

The table below defines the properties you supply when you run the scaffolding task.

Properties	Description
`groupId`	The group ID used in the generated `pom.xml` of of the plugin.
`artifactId`	The name of the plugin; the string you supply will be the name of the plugin's root directory. Supply a lower case string.
`version`	The plugin version
`package`	The base package structure for generated controllers
`controller`	The name of the controller for which the scaffolding will create a controller file. The task will create a controller file, whose name is the string "Controller" concatenated with *Controller_Name*, as in: `Controller_NameController.groovy`. Supply a value with leading character in upper case.
`controllerDir`	The name of the directory that contains views for *Controller_Name*; the string you supply will be the name of the directory in which views are created. Supply a lower case string.

Files and Directories Created by Scaffolding Task

The scaffold task creates the directory structure and files shown below in the `hq-uiplugin` directory of the source tree.

```
pluginname

/*
where "pluginname" is the value of "artifactID"
*/

     src/main/groovy/
           Plugin.groovy
           app/
                ControllerNameController.groovy
                /*
                where "ControlName" is the value of "controller"
                */
           src/main/resources
                plugin.properties
                views/
                     controller_views/
                          index.gsp
                etc/
                     pluginname_i18n.properties
                     /*
                     where "pluginname" is the value of "artifactID"
                     */
```

The maven target creates a single controller file and the default view generated by that controller, index.gsp. Note that if your requirements dictate, a single controller can render multiple views. To accomplish this, you would add an additional action that renders the other view to the controller.

Note also that a plugin can have multiple controllers. To accomplish this, you would write an additional controller and put it in the /app directory, and add a subdirectory for that controller's views under the /app directory.

Contents of the Generated Plugin Files

The plugin files that are created are a starting point for the plugin development process. The newly created plugin is functional, to the extent that you can invoke it and see it render a view to the browser. In following steps, you edit the plugin files to define its behavior.

The table below describes the contents of the plugin files upon creation.

Plugin File	**Contents When Created**
Plugin.groovy	Imports the plugin controller file; contains commented out property definitions for attaching the plugin to the HQ Portal.
plugin.properties	Sets the property values for plugin name, help tag, and version information.
AlertController.groovy	Imports BaseController.groovy; limits access to users with Super User role, and contains a sample action (method), called "index", and shows usage of the render method to render the plugin name and user name.
index.gsp	Contains sample text for the view, and shows the use of variables.
joesalertcenter_i18n.properties	Example usage of the the property that contains the plugin description, and a localization string.

Step 2 - Build the Plugin

You should now be able to package your new plugin for deployment to HQ:

```
$ cd hq-uiplugin/_plugin_name_
$ mvn clean assembly:directory
```

Step 3 - Deploy and Run the Plugin

To deploy the plugin:

```
$ cd hq-uiplugin/_plugin_name_
$ cp -r target/plugin_name-VERSION-server/plugin_name/* TOMCAT_HOME/webapps/ROOT/hqu/plugin_name
```

NOTE: To hot deploy updates to the plugin, remove the plugin folder completely from the hqu folder to undeploy and then paste the updated folder back in to redeploy.

To invoke the plugin from a browser:

```
http://localhost:7080/hqu/plugin_name/Controller_Name/index.hqu
```

For example, given the values used for *plugin_name* and *Controller_Name* in the example task execution above:

```
http://localhost:7080/hqu/joesalertcenter/Alert/index.hqu
```

When you invoke the plugin, its default action "index" is executed.

Step 4 - Set Location Properties for a View Plugin

In this step, you define how the plugin attaches to the HQ Portal, and the controller and action that will be invoked when the plugin is run. This behavior is defined in Plugin.groovy. If you do not wish to attach the plugin to the HQ Portal, this step is optional.

Excerpt from Plugin.groovy as Created

This is the portion of Plugin.groovy you edit:

```
/*
        addView(description:  'A Groovy HQU-_plugin_name_',
                attachType:   'masthead',
                controller:   _Controller_Name_Controller,
                action:       'index',
                category:     'tracker')
         */
    }
}
```

Understanding Attachment Properties and Options

This table defines the properties defined in `Plugin.groovy` that govern plugin attachment and invocation behavior.

Property	Behavior and Options	Default
description	Brief description of the view displayed as the View title in the Portal.	'A Groovy HQU-plugin_name'
attachType	Specifies where the plugin will be attached. 'masthead'—attach to a menu on the Portal masthead. 'admin'—attach to Administration page. 'resource'—attach to View page of a Resource.	'masthead'
resourceType	If attachType is set to 'resource' use this property to identify the type of the resource whose **View** page should show the plugin.	The Plugin.groovy file generated by the scaffolding does not contain this property, you must add it yourself to use it.
controller	Specifies the filename of the plugin controller to invoke when the the plugin is run.	Specifies the controller file created by the scaffolding process.
action	The method, in the controller file specified by the controller property, to invoke when the plugin is run. If you change the value of action, remember that each action defined must have a similarly named .gsp file in views/*controller_views*.	Specifies the sample index ()method contained in the controller file.
category	Defines the view plugin's Portal attachment point, when attachType is 'masthead'. It may be defined as: 'tracker'——plugin will appear on Analyze menu. 'resource'—plugin will appear on Resources menu. (blank)—set to blank unless the attachType is 'masthead'.	'tracker'

Specify View Location in Plugin.groovy

Uncomment the addView method in Plugin.groovy.

Attach Plugin to a Masthead Menu

1. Leave the attachType property set to 'masthead'.
2. Set the category property to blank.
3. Edit the values of description, controller, and action, as desired.

Attach Plugin to Administration Page

1. Set the value of attachType property set to 'admin'.
2. Set the category property blank.
3. Edit the values of description, controller, and action, as desired.

Attach Plugin to Resource>View page

To add an item to the 'Views' tab of a resource, and gain access to resource context, you can attach a view to a resource type:

1. Set the value of attachType property set to 'resource'.
2. Set the category property blank.
3. Add, and set the resourceType property to a valid Resource type, enclosed in single quotes and square brackets, for instance ['Linux'].
4. Edit the values of description, controller, and action, as desired.

Detaching and Removing HQU Plugins

HQ has a built-in HQU plugin, called Plugin Mon, for managing other HQU plugins. To detach plugin views and remove a plugin from the database, use Administration -> HQU Plugin Management.

Step 4 - Program the Plugin Behavior

The real work of developing an HQU plugin is defining its behavior, in the plugin controller file. For controller development topics, see Writing Plugin Controller Logic (see page 22).

Step 5 - Localize the Plugin

HQU supports the I18N internationalization model. The scaffolding tool creates a PropertyResourceBundle file for localization strings properties, `plugin_name_i18n.properties` in `/ui_plugins/{plugin/etc`.

Contents of the I18n Properties as Created

Upon creation `plugin_name_i18n.properties` contains two key-value pairs.
plugin_name.description=[new plugin] plugin_name
Congrats=Congratulations
The first line defines the link name for plugins attached to the portal, where:

- *plugin_name* is the name you assigned to the plugin when running the scaffolding task.
- new plugin is an arbitrary string that will be prefixed to *plugin_name*

The menu item the is created is:
new plugin *plugin_name*
The second line defines the string to present in Views for the text item whose key is Congrats.

Understanding How Plugin Views Obtain Localization Strings

A special variable in the .gsp rendered for an action allows it to obtain localized resource bundles from `etc/myplugin_i18n.properties`. The variable, a lower-case letter 'l', is of the type 'BundleMapFacade' which provides a map-like facade, allowing you to access resource bundle attributes like regular Map entries. For example, in a line from a `.gsp`, in either of these forms:

```
${l.Congrats}
```

```
$l['Congrats']
```

The variable `l.Congrats` points at `etc/myplugin_i18n.properties:Congrats`

For information about how a plugin controller can obtain localized resource bundles, see Obtaining Localized Strings (see page 24).

Writing Plugin Controller Logic 6

- Accessing HQ Data and Functions (see page 22)
  - HQU APIs (see page 22)
  - Backend APIs (see page 22)
- Defining User Interaction (see page 23)
  - Defining Parameters (see page 23)
  - Creating Dojo Tables (see page 24)
- Rendering (see page 24)
  - Obtaining Localized Strings (see page 24)
  - Rendering Ajax (see page 25)
  - Returning XML from Controllers (see page 25)
  - Returning JSON from Controllers (see page 25)
- Miscellaneous Controller Topics (see page 26)
  - Groovy Console* (see page 26)
  - Re-using Markup with Groovy Templates (see page 26)
  - Using Variables in Controllers (see page 26)
  - Authorization (see page 26)
  - Snippets (see page 26)

Accessing HQ Data and Functions

HQU APIs

HQU provides a set of Groovy APIs for accessing HQ data and functions, located in the hq-rendit jar. The Groovy APIs provide a simpler, more intuitive interface to HQ which will be supported and deprecated as API versions periodically change. Although an HQU plugin can call any HQ class, Hyperic recommends the use of the Groovy APIs. Note however, in this release of the HQU Framework, not all backend functions are available through the Groovy APIs.

Backend APIs

HQU plugins can use HQ's backend APIs, which can perform any HQ function. **This is not officially supported and may change in any future release of Hyperic.**

HQ Manager APIs are POJOs that provide transactional functionality, for example:
hq-server/src/main/java/org/hyperic/hq/SUBSYSTEM/server/session/SomeManagerImpl.java

Invocation of a manager operation is simple:

```
import org.hyperic.hq.context.Bootstrap;
import org.hyperic.hq.appdef.shared.PlatformManager;

private getPlat10001() {
    Bootstrap.getBean(PlatformManager.class).findPlatformById(10001)
}
```

The Bootstrap class is a wrapper around the Spring application context that returns a singleton instance of a manager POJO.

Defining User Interaction

Defining Parameters

Controller actions are executed with parameters from the HTTP request. One, or multiple, values may be obtained for a parameter.

For example, this request provides a single timeout value and multiple id values:

```
http://localhost:7080/hqu/pinger/madcast/execute.hqu?timeout=300&id=3&id=4
```

To define the timeout parameter, for which a single value is provided:

```
def timeout params.getOne('timeout')
```

To define the id parameter, for which a two value are provided:

```
def id params.get('id')
```

The listing of MadcastController.groovy below, illustrates both usages, allowing for a single value of timeout and multiple values of id to be obtained from the HTTP request.

```
import org.hyperic.hq.hqu.rendit.BaseController

class MadcastController extends BaseController {

    def execute(params) {
        def timeout = params.getOne('timeout', 60 // default timeout if not in URL)

        def resources = lookupAndValidateResources(timeout, params.ids)

        def start  = now
        def errors = ExternalProgram.execute(timeout, resources)
        log.info "Mastcasted to $\{resources.size()\} resources in $\{now - start\}ms"
        render(locals:\[
            castTime:now - start,
            isAdministrator: user.hasAdminPrivilege(),
            errors: errors\])
    }

    private getNow() {
        System.currentTimeMillis()
    }

    private lookupAndValidateResources(timeout, ids) {
        ....
    }
}
```

Creating Dojo Tables

DojoUtil.groovy provides utility methods for creating complex content which interacts with HQ.

- dojoTable()—A versatile, AJAX-based table that can deal with paging, sorting, and localization.
- processTableRequest()—Processes table requests from dojoTables
- dojoTabContainer()—A tabbed container, with hidden tabs.

An example of processTableRequest:

```
def data(params) {
        def json = DojoUtil.processTableRequest(SYSTEMSDOWN_SCHEMA , params)
        render(inline:"/* ${json} */", contentType:'text/json-comment-filtered')
    }
```

Rendering

Obtaining Localized Strings

The controller can obtain localized values for a controller method. This is useful if the controller needs to generate text that will be displayed in the rendered view, or write localized strings to a log or other file.

Controller files can use the getLocaleBundle() method of BaseController.groovy, which returns a Map facade over a ResourceBundle.

```
def getName(u) {
        "${localeBundle.NameIs}: ${u.name}"
    }
```

Rendering Ajax

If you want to render AJAX, see `RenderFrame.groovy` documentation at "
https://fisheye.springsource.org/browse/hq/hq-rendit/src/main/groovy/org/hyperic/hq/hqu/rendit/render/RenderFrame.groovy
".

Returning XML from Controllers

Groovy provides the `groovy.xml.MarkupBuilder` class for generating XML.
To return XML from your controller, specify the methods whose results should be formatted as XML, using setXMLMethods().

```
class EdatorController extends BaseController {
    EdatorController() {
        setXMLMethods(['findUsers'])
    }
    def findUsers(xmlBuilder, params) {
        xmlBuilder.users {
            for (u in UserManager.getUsers()) {
                user(Name:u.name, Id:id)
            }
        }
        xmlBuilder
    }
}
```

HQU creates a groovy.xml.MarkupBuilder and passes it as the first argument to your controller method. The controller should return the builder object, or some other XML that it wants to render.

For more information, see "http://groovy.codehaus.org/Using+MarkupBuilder+for+Agile+XML+creation".

Returning JSON from Controllers

Requests are sent to HQ via a regular HTTP request:

```
curl https://hqadmin:hqadmin@localhost:7443/hqu/myplugin/edator/viewUser.hqu?name=JoeBob
```

and result in JSON objects:
{"Name": "JoeBob", "Id": 32321 }

To return JSON to the browser, specify the methods whose results should be formatted as JSON, using setJSONMethods.
The result of the action must be a map (key-value pairs), but can contain values of most types.

```
class EdatorController extends BaseController {
    EdatorController() {
        setJSONMethods(['viewUser', 'newUser'])
    }
    def viewUser(params) {
        def user = myFindUser(params)
        [ Name: user.name, Id: user.id ]
    }
    def newUser(params){ ... }
}
```

Miscellaneous Controller Topics

Groovy Console*

Re-using Markup with Groovy Templates

Templates provide a way to reuse markup for different actions. If you find yourself using a fragment of Groovy code frequently, for instance an HQL query, you can create a template for it.

To create a template, copy it into your gconsoleTemplates directory

```
$ cp myTemplate.groovy $TOMCAT_HOME/webapps/ROOT/WEB-INF/gconsoleTemplates
```

If you are building and deploying HQ from source, you can also place it in ~/.hq/gconsoleTemplates and it will be automatically picked up by the build.

Templates will appear as menu items in the Groovy Console.

Using Variables in Controllers

Authorization

Snippets

Groovy Tips 7

This section contains key facts about Groovy that may be useful to an HQU developer, and links to sites with good Groovy reference material.

Links to Groovy Topics

What's There	Link
Groovy home page	"http://groovy.codehaus.org/"
Groovy JDK API specification, provides a list of Groovy extensions to standard JDK classes	"http://groovy.codehaus.org/groovy-jdk/"
Discusses Groovy's dynamic features with how-to on implementing the GroovyObject interface and using ExpandoMetaClass, a expandable MetaClass that allows adding of methods, properties and constructors.	"http://docs.codehaus.org/display/GROOVY/Dynamic+Groovy"
"Cool Things You Can do with the Groovy Dynamic Language"	"developers.sun.com/learning/javaoneonline/2007/pdf/TS-1742.pdf (http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-1742.pdf)

More on Groovy dynamic features, with examples of : ▪ adding methods onto interfaces ▪ overriding invokeMethod, getProperty and setProperty to provide "missing method" behavior ▪ "borrowing" methods from other classes ▪ constructing method property names at runtime	" http://graemerocher.blogspot.com/2007/06/dynamic-groovy-groovys-equivalent-to.html "
Groovy Server Pages; short page of information.	"http://docs.codehaus.org/display/GROOVY/GSP"
Hyperic HQU Plugins for download.	http://support.hyperic.com/display/hyperforge/HQU+Plugins
Groovy Markup	"http://groovy.codehaus.org/GroovyMarkup (http://groovy.codehaus.org/GroovyMarkup)
"Practically Groovy: Reduce code noise with Groovy", an article on benefits of Groovy concise syntax, compared to Java.	"http://www.ibm.com/developerworks/java/library/j-pg09196.html"

Groovy def

In Java, methods must exist within a class object, and you must define behavior within the context of a class. In Groovy, you can define behavior within a function, which can be defined outside a class definition.
You define a Groovy functions with the def keyword. defs do not require any type declarations for parameters, nor do defs require a return statement.

Property Accessor Methods

In a Groovy class, it is not necessary to explicitly use a public property's getter and setter methods. Instead of:
s.getProp()
you can have:
s.prop
In Groovy, property accessors are generated for you, unless you explicitly declare the properties to be private.
For more information, see "Practically Groovy: Reduce code noise with Groovy", at " http://www.ibm.com/developerworks/java/library/j-pg09196.html. (http://www.ibm.com/developerworks/java/library/j-pg09196.html)

Accessibility Modifiers

In a Groovy class, the accessibility modifier for a member or type is assumed to be public; you don't need to specify the modifier unless you wish to declare the item private or protected.

Inferred Types

Groovy infers the data type of a property.

Plugin Scaffolding 8

The contents of the files created by the HQU scaffolding task are included here for reference.

Plugin.groovy

```
import org.hyperic.hq.hqu.rendit.HQUPlugin

import _ControllerName_Controller

class Plugin extends HQUPlugin {
    Plugin() {
        /**
         * The following can be uncommented to have the plugin's view rendered in HQ.
         *
         * description:  The brief name of the view (e.g.: "Fast Executor")
         * attachType:   one of \['masthead', 'admin'\]
         * controller:   The controller to invoke when the view is to be generated
         * action:       The method within 'controller' to invoke
         * category:     (optional)  If set, specifies either 'tracker' or 'resource' menu
         */
        /*
        addView(description:  'A Groovy HQU-{_}plugin_name{_}',
                attachType:   'masthead',
                controller:   _ControllerName{_}Controller,
                action:       'index',
                category:     'tracker')
         */
    }
}
```

ControllerName Controller.groovy

```
import org.hyperic.hq.hqu.rendit.BaseController

class _ControllerName{_}Controller
 extends BaseController
{
    protected void init() {
        onlyAllowSuperUsers()
    }

    def index(params) {
        // By default, this sends views/{_}ControllerName{_}/index.gsp to
        // the browser, providing 'plugin' and 'userName' locals to it
        //
        // The name of the currently-executed action dictates which .gsp file
        // to render (in this case, index.gsp).
        //
        // If you want to render AJAX, read RenderFrame.groovy for parameters.
     render(locals:\[ plugin :  getPlugin(),
                     userName: user.name\])
    }
}
```

index.gsp

```
Congrats!
|Plugin Name| |
|Description| |
|Version| |
Your username is ${userName}
The method named 'index' in app/_ControllerName_Controller.groovy was invoked to render this page.
It then rendered views/_ControllerName_/index.gsp which you are reading.
You'll also want to change the following files:
/Users/mmcgarry/hq/ui_plugins/_plugin_name_/etc/_plugin_name__i18n.properties contains the
description
```

plugin_name _i18n.properties

```
# The following property is the localized description of the plugin, which
# is typically used as the link name in an attach point
_plugin_name_.description=[new plugin] _plugin_name_
Congrats=Congratulations
```

plugin.properties

```
plugin.name=_plugin_name_
plugin.helpTag=_plugin_name_.Help
plugin.version=_project.version_
```

HQ Groovy APIs

9

These topics provide information about the HQ's Groovy APIs and how to use them.
WIP Note: This section is a work in progress. In the next version of this guide, topics will be added to provide this information:

- What the Groovy APIs can and cannot do. In a future release, the Groovy APIs will provide access to all backend HQ data and functionality. Currently, not all backend data and functionality can be access by the Groovy APIs.
- More complete API documentation. Some API documentation is missing or incomplete.

Coding Conventions

Many times, methods take only a single argument; a Map of parameters. For example:

```
def findSomething(p) {
   def name = p['name']
   def auto = p.get('auto', true)
}
```

In this example, the method takes two arguments, the second of which, "auto" defaults to true.

Obtaining a Helper Instance

To get an instance to a helper in your method, use its getter, available from BaseController.groovy:

```
import org.hyperic.hq.hqu.rendit.BaseController
class _ControllerName_Controller extends BaseController {
    def _ControllerName_Controller() {    }
    def index(params) {
       def myPlatform = getResourceHelper().find(platform:'My Platform') // or more succinctly

       def yourPlat = resourceHelper.find(platform:10001)
    }
}
```

BaseController Utility Methods

All controllers extend BaseController and therefore have access to all of its utility methods.

```
def MyController extends BaseController {
   MyController() {
       // Have a closure execute before every action is invoked.
       // Also provides getUser() to fetch the current AuthzSubject
       // making the request
       addBeforeFilter({ log.info "Current request from ${user}"})
   }
   def index(params) {
       log.info 'Hello world!'  // built in logging via log4j.
       // InvokeArgs provides both HttpServletRequest / Response
       log.info "Request from: ${invokeArgs.request.serverName()}"
   }
```

- urlFor(): Uses HtmlUtil.urlFor to generate URLs
- linkTo(): Generates a link to another action or resource
- now(): System.currentTimeMillis()
- escapeHtml(): Escapes a string of HTML

BaseController provides wrappers around a few methods from the Util classes found here:
hq-rendit/src/main/groovy/org/hyperic/hq/hqu/rendit/util
hq-rendit/src/main/groovy/org/hyperic/hq/hqu/rendit/render
hq-rendit/src/main/groovy/org/hyperic/hq/hqu/rendit/html

MetaClasses and Categories

Groovy allows dynamic insertion of methods into objects dynamically at runtime. HQU takes advantage of this to make methods on objects seem more meaningful.

```
private getPlatformToDelete(id) {
        AuthzSubject curUser = getUser()
        log.info "${curUser.isSuperUser()}"
    }
```

The class AuthzSubject exists in HQ, however, it does not contain the method 'isSuperUser()'. AuthzSubjectCategory.groovy adds it at runtime. See this excerpt from AuthzSubjectCategory.groovy:

```
class AuthzSubjectCategory {
    static boolean isSuperUser(AuthzSubject subject) {
        PermissionManagerFactory
      .getInstance()
            .hasAdminPermission(subject.id)
    }
}
```

Static methods in the category class tell Groovy to insert those methods for their associated object class. The first parameter is the recipient of the invocation, and all subsequent parameters are passed normally.

ResourceHelper.groovy

General purpose utility method for finding resources and resource counts. It provides methods for finding:

- the counts of platforms, servers, or services
- platforms, servers, or services

- a subset of all platforms, servers, or services
- all platforms, servers, or services, sorted by name
- all resource groups
- a prototype by name
- all prototypes of platforms, servers, and services a group by ID

AlertHelper.groovy

Provides methods to find:

- Alerts within a specified timerange, greater or equal to a given priority
- Recent alerts with at least the specified severity level
- Group alerts within a specified timerange, greater or equal to a given priority
- Recent alerts with at least the specified severity level
- All alert definitions
- Type-based alert definitions; these are the templates for the individual resource definitions.
- Group alert definition

AuditHelper.groovy

BaseHelper.groovy

MetricHelper.groovy

AgentHelper.groovy

Sample Plugin

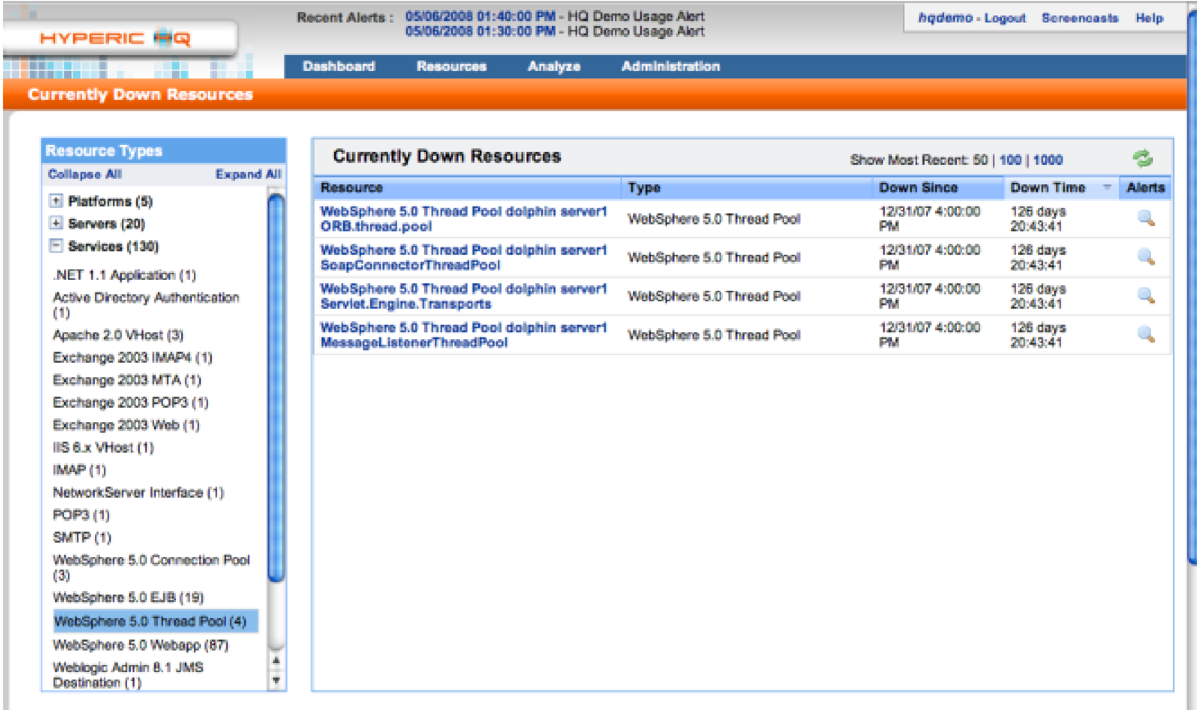
10

This section describes HQ's Currently Down view, an HQU Plugin, and provides a listing of its controller file.

Currently Down View

This screen allows users to centrally view all the managed resources (platforms, servers, and services) that are currently unavailable and when they went down. The screen automatically refreshes every minute, but users can also refresh it manually.

- Resource Types—Presents down resources by inventory level and resource type. Users can expand or collapse the view, and listall down resources of a particular resource type by clicking the type here.
- Availability—Lists the down resources of the resource type selected in "Resource Types." The default sort order is by Down Time (from longest to shortest). Users cannot page through the list of down resources; they must instead select a number of resources to be displayed. This section also provides a link to the resource's alerts (where any alerts triggered by the resource's unavailability will be listed).
- Resource: Name of the down resource
- Type: The resource's platform, server, or service type
- Down Since: The time at which the resource became unavailable
- Down Time: The length of time the resource has been unavailable
- Alerts: The icon links to a list of alerts for this resource



SystemsdownController.groovy

```
import java.text.DateFormat
import java.util.Locale
import org.hyperic.util.units.FormatSpecifics
import org.hyperic.util.units.UnitsConstants
import org.hyperic.util.units.UnitsFormat
import org.hyperic.util.units.UnitNumber
import org.hyperic.hq.hqu.rendit.html.DojoUtil
import org.hyperic.hq.hqu.rendit.BaseController
import org.hyperic.hq.appdef.server.session.DownResSortField

class SystemsdownController extends BaseController
{
    private final DateFormat df =
        DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.MEDIUM)

    private getAlertListImg() {
        def imgUrl = urlFor(asset:'images') +
            "/icon_zoom.gif"
        """<img src="${imgUrl}" width="16" height="16" border="0"
                class="alertListIcon" title="Click to go to the alert list for this resource">"""
    }

    private final SYSTEMSDOWN_SCHEMA = [
        getData: {pageInfo, params ->
            resourceHelper.getDownResources(params.getOne('typeId'), pageInfo)
        },
        defaultSort: DownResSortField.DOWNTIME,
        defaultSortOrder: 1,  // descending
        columns: [
            [field:DownResSortField.RESOURCE, width:'37%',
             label:{linkTo(it.name,
                           [resource:it.resource.entityId])}],
            [field:DownResSortField.TYPE, width:'30%',
             label:{it.type}],
            [field:DownResSortField.SINCE, width:'15%',
             label:{df.format(it.timestamp)}],
            [field:DownResSortField.DOWNTIME, width:'13%',
             //label:{formatDuration(it.duration)}
              label:{formatDuration(it.duration)}
              ],
            [field:DownResSortField.ALERTS, width:'5%',
             label:{
             linkTo(getAlertListImg(), [resource:it,rawLabel:true])
             }],
        ]
    ]

   def SystemsdownController() {
        setTemplate('standard')
    }

    def getNow() {
        System.currentTimeMillis()
    }

    def formatDuration(d) {
        return UnitsFormat.format(new UnitNumber(d, UnitsConstants.UNIT_DURATION,
                                                 UnitsConstants.SCALE_MILLI),
                                  Locale.getDefault(), null).toString()
    }
```

```
    def index(params) {
     render(locals:[ systemsDownSchema : SYSTEMSDOWN_SCHEMA, numRows : params.numRows])
    }

    def data(params) {
        def json = DojoUtil.processTableRequest(SYSTEMSDOWN_SCHEMA , params)
  render(inline:"/* ${json} */", contentType:'text/json-comment-filtered')
    }

    def getTypeJSON(type, count) {
        def json = ""
        if (type != null) {
            json += "{name: \"" + type.name + "\", id: \""
            json += type.appdefType + ":" + type.id + "\", count: " + count + "}"
        }
        return json
    }

    def summary(params) {
        def map = resourceHelper.downResourcesMap.entrySet()

        def json = "[\n"

        def appdefType = 1
        def first = true
        map.each { entry ->
            def list = entry.value

            if (list.size() > 0) {
             if (first) {
              first = false
             }
             else {
                 json += ",\n"
             }

                json += "{parent: \"" + entry.key + "\",\n" +
                        "id: " + appdefType + ",\n" +
                        "count: " + list.size() + ",\n" +
                        "children:[\n"

                def previous = null
                def count = 0

                list.each { type ->
                    if (previous == null || previous.name != type.name) {
                        json += getTypeJSON(previous, count)

                        if (previous != null) {
                            json += ",\n"
                        }

                        previous = type
                        count = 0
                    }

                    count++
                }

                json += getTypeJSON(previous, count)
                json += "]\n}"
            }
            appdefType++
```

```
        }

        json += "]"
  render(inline:"/* ${json} */", contentType:'text/json-comment-filtered')
    }
}
```