Groundwork Connector Developers Guide

Guide for Writing CloudHub Virtualization Connectors for CloudHub 2.0

Introduction 4

Dependencies 4

CloudHub Source Code 4

Building, Testing and Running CloudHub 5

Building CloudHub Dependencies One Time 5

Building Online vs. Offline 5

Building CloudHub War File 6

Deploying to JBoss Server 6

Tomcat Option 6

Connector Package 7

Configure Enums and Enum Conversions 7

Implement the Configuration 7

Implement the Configuration Class 8

Implement the Connection Class 9

Implement the Configuration Provider 9

Create a Unit Test 11

Implement the Connector 11

Implementation Steps 12

Add the Connector to the ConnectorFactory 13

Create a Unit Test 13

Profiles 14

Your Profile Template 14

Metric Defaults in Java 16

Defining Synthetics 16

User Interface 17

Configuration JSP 17

Spring MVC Controller 19

New Image 19

Index.jsp 19

Home Controller 19

Profile Template JSP 20

Dynamic Properties 20

# Introduction

CloudHub is a real-time virtualization monitoring application. CloudHub interacts with the API of the virtualization management servers, periodically collecting configuration and monitoring data and storing it in the Groundwork (GWOS) Monitoring Server. CloudHub Connectors are the pluggable Java classes that connect to virtualization management systems, gather configuration and monitoring metrics, and report those metrics back to the GWOS monitoring server. Communication with the GroundWork server is through the Business Service REST API, which greatly simplifies submitting data collected from the management servers to the GroundWork Server.

This document will guide you on the step-by-step procedures to writing a CloudHub Connector. CloudHub has a pluggable architecture. Connectors can plugin to this architecture by implementing a set of interfaces.

When using this guide, follow the ordered procedure for implementing your connector.

# Dependencies

All JAR dependencies required to implement a CloudHub Connector must be provided in a Maven pom.xml format. The dependencies should not be added to the project until Groundwork engineering staff reviews the JAR dependencies. Please submit a list of dependencies required for your connector before proceeding. An XML snippet of Maven dependencies in POM format is all this is required. Submit all dependencies with version number and scope.

# CloudHub Source Code

You can check out the CloudHub source code from here:

<http://geneva/groundwork-professional/branches/monitor-platform-ec2-connector>

CloudHub is a part of the Monitor Platform project. It is found under the agents sub-project. The entire monitor-platform project is built with a Maven-3 build. The directories found under the cloudhub directory are in standard Maven directory format.

+ monitor-platform

+ agents

**+ cloudhub**

**+** src

+ main

**+ java**

+ profiles

**+ webapp**

+ profiles

+ test

**+ java**

Under the src/main/java directory is the root of the Java packages. Similarly, the root of the Java Test packages is found under src/test/java. CloudHub webapp resources are found under src/main/webapp. Example CloudHub profiles are located under src/profiles.

# Building, Testing and Running CloudHub

You can either run CloudHub in unit tests, or you can deploy it to an application server. Note: both tests and running will require that Groundwork is installed on your file system under /usr/local/groundwork. Some unit tests do not require the server to be running, while other tests do require the GWOS server to be running. For example, all tests under the org.groundwork.cloudhub.monitor and org.groundwork.cloudhub.gwos packages require that the GWOS server is running and that the GWOS Rest Services are listening.

CloudHub is built with Maven-3. In order to build CloudHub, prerequisites are:

* Java7
* Maven-3
* Install Groundwork Server (contact Groundwork staff for instructions)
* A connection to our Nexus repository or a zip of a minimal Maven local repository

## Building CloudHub Dependencies One Time

The first time you build CloudHub, you will need to build all the Groundwork dependencies to the CloudHub project. This will build not only CloudHub, but also all the essential Groundwork jars used in CloudHub.

As an example, lets checkout the entire project into your $HOME directory:

cd $HOME

svn co http://geneva/groundwork-professional/branches/monitor-platform-ec2-connector

cd monitor-platform-ec2-connector

mvn clean install -P connector-dev

## Building Online vs. Offline

When doing the one time build, you will need to have the VPN up. All of our project dependencies must be downloaded from Groundwork’s Nexus Snapshot Repository to your local repository, stored in $HOME/.m2/repository. The build will automatically download these dependencies for you.

Once you have downloaded the Snapshot dependencies, you can then build ‘offline’. Building offline means that Maven will not check for snapshot dependency updates on our Nexus repository. The default policy is to check for snapshot updates once per day. To ignore this policy, and build without the VPN up, use the offline command line switch –o :

mvn -o install

## Building CloudHub War File

CloudHub’s default target builds a war file that can be deployed to either the JBoss Application Server included with a GWOS installation or to a Tomcat server. The JBoss App Server is provided in /usr/local/groundwork/jpp when you install the Groundwork product.

### Deploying to JBoss Server

Once you have built all the one-time dependencies, then you can develop from the CloudHub root directory:

cd $HOME/monitor-platform-ec2-connector/agents/cloudhub

mvn clean install

A war file is created in the target directory. When deploying to JBoss, copy the war file into the /usr/local/groundwork/jpp/standalone/deployments directory, then touch a file named cloudhub.war.dodeploy to initiate hot deploy.

cp target/cloudhub.war /usr/local/groundwork/jpp/standalone/deployments/

touch /usr/local/groundwork/jpp/standalone/deployments/cloudhub.war

Depending on the speed of your system, give it a few seconds to deploy and then point your browser at:

<http://localhost:8080/cloudhub>

### Tomcat Option

To build for Tomcat 7, type:

cd $HOME/monitor-platform-ec2-connector/agents/cloudhub

mvn clean install -P tomcat

Note that the Tomcat war file is actually named cloudhub-tc.war. The file should be renamed to cloudhub.war when dropping into Tomcat’s webapps directory.

cp target/cloudhub-tc.war $TOMCAT\_HOME/webapps/cloudhub.war

Where $TOMCAT\_HOME is the root of your Tomcat installation. Depending on the speed of your system, give it a few seconds to deploy and then point your browser at:

<http://localhost:8080/cloudhub>

# JavaScript App Production Distributable

By default, the JavaScript App builds the production mode distributable with Web Pack. During development, this NPM task can be time consuming. To skip the production build, build with the –DskipTest=true flag:

cd $HOME/monitor-platform-ec2-connector/agents/cloudhub

mvn clean install –DskipDist=true

# OR for Tomcat

mvn clean install -P tomcat –DskipDist=true

# Connector Package

Place all of your source code in a well-defined package. For example, if your connector name is ‘amazon’ place your source code in the org.groundwork.connectors.amazon package.

# Configure Enums and Enum Conversions

Enumerated types are used to define all connectors and profile types in the system. You will want to define these before getting started on implementing your connector. Add your enums here:

**Virtual System**: the virtual system enum represents all known virtual system connectors in the system

org.groundwork.agents.monitor.VirtualSystem

**Profile Type**: profile types usually correlate to the virtual system. Profiles define the metric sampling parameters. Note that since profiles are used in both the GWOS Server and Cloudhub, the classes reside in the collagerest-common package.

org.groundwork.rs.dto.profiles.ProfileType

**Profile Type conversions**: Hold methods to convert between various enumerations and base classes:

org.groundwork.cloudhub.profile.ProfileConversion

Implement your case statement for these methods:

convertVirtualSystemToPropertyType

convertVirtualSystemToProfileClass

convertProfileTypeToVirtualSystem

convertVirtualSystemToHubType

When choosing an enumeration name for a Virtual System and Profile Type, keep the name short and follow the pattern already in place. Submit the proposed enumeration names to Groundwork engineering staff for approval.

# Implement the Configuration

CloudHub stores its configurations on the local file system in XML files under /usr/local/groundwork/config/cloudhub. These files are named as with the following format:

cloudhub-{virtualSystem}-{sequenceNumber}.xml

# 

* Note that {virtualSystem} is always converted to lowercase

Configuration files have similar XML formats, but there are always differences in the connection section. For example, some servers require username/password, while others require tokens or tenant ids.

Here is a typical skeleton for a configuration file. Configuration files are broken up into 3 sections:

1. <common> - the same elements for all configurations
2. <gwos> - the same elements for all configurations
3. A connector-specific section (in this example called <amazon>) describing connection parameters:

# 

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<vema>

<common>

…

</common>

<gwos>

</gwos>

<**amazon**>

YOUR connection elements go here

</**amazon**>

</vema>

You can find sample configuration files from other connectors under /usr/local/groundwork/config/cloudhub.

## Implement the Configuration Class

Start by implementing your Configuration class. All configurations should extend the abstract class:

org.groundwork.cloudhub.configuration.ConnectionConfiguration

and make use of JAXB for all XML mapping annotations. Here is the JAXB mapping for the base class:

@XmlType(propOrder = {**"common"**, **"gwos"**})

**public abstract class** ConnectionConfiguration **implements** MonitorConnectionConfig {

When implementing your class, use this code snippet as a template to get started (see DockerConfiguration class as an example):

@XmlRootElement(name = **"vema"**)

@XmlType(propOrder={**"connection"**})

**public class** AmazonConfiguration **extends** ConnectionConfiguration {

@Valid

**private** AmazonConnection **connection**;

**public Amazon**Configuration() {

**super**(VirtualSystem.***AMAZON***);

**connection** = **new** AmazonConnection();

}

@XmlElement(name = **"amazon"**)

**public** AmazonConnection getConnection() {

**return connection**;

}

**public void** setConnection(DockerConnection connection) {

**this**.**connection** = connection;

}

}

## Implement the Connection Class

Next, implement your Connection class. Connections hold the connector-specific connection parameters like username, password, tokens, or provisioning keys. All configurations should extend the abstract class:

org.groundwork.cloudhub.configuration.BaseMonitorConnection

Here is a skeleton for your Connection class:

@XmlRootElement(name = **"amazon"**)

@XmlType(propOrder = {**"server"**, **"username"**, **"password"**})

**public class** VmwareConnection

**extends** BaseSecureMonitorConnection **implements** SecureMonitorConnection {

@NotBlank (message=**"Server URI cannot be empty."**)

**private** String **server**;

**private** String **username**;

private String password;

**public** AmazonConnection() {

}

// implement getters setters here

}

## Implement the Configuration Provider

Next, implement your Configuration Provider class. Connection Providers provide a facade onto your connector’s configuration so that it can be easily plugged into the CloudHub engine. Configuration Providers implement the ConfigurationProvider interface and extend the BaseConfigurationProvider class.

Use this code template to get started. This example is based on the VMware configuration provider:

@Service(VMwareConfigurationProvider.***NAME***)

**public class** VMwareConfigurationProvider

**extends** BaseConfigurationProvider **implements** ConfigurationProvider {

*// Display Names*

*// choose a descriptive display name to describe your Management Server*

**public static** String *MGMT\_SERVER\_VMWARE* = **"vSphere management server"**;

// choose a descriptor display name to describe your hypervisor

**public static** String *HYPERVISOR\_VMWARE* = **"ESXi hypervisor"**;

*// Connector Constant - this is the constant used to define your connector.*

*// Must be lower case*

**public static** String *CONNECTOR\_VMWARE* = **"vmware"**;

*// Application Type - the application type constant.*

*// Work with Groundwork team to choose a name*

**public static** String *APPLICATIONTYPE\_VMWARE* = **"VEMA"**;

*// Prefixes*

**public static** String *PREFIX\_VMWARE\_MGMT\_SERVER* = **"VSS:"**;

**public static** String *PREFIX\_VMWARE\_HYPERVISOR* = **"ESX:"**;

**public static** String *PREFIX\_VMWARE\_NETWORK* = ConnectorConstants.***PREFIX\_NETWORK***;

**public static** String *PREFIX\_VMWARE\_CLUSTER* = ConnectorConstants.***PREFIX\_CLUSTER***;

**public static** String *PREFIX\_VMWARE\_STORAGE* = ConnectorConstants.***PREFIX\_STORAGE***;

**public static** String *PREFIX\_VMWARE\_DATACENTER* = ConnectorConstants.***PREFIX\_DATACENTER***;

**public static** String *PREFIX\_VMWARE\_RESOURCE\_POOL* = ConnectorConstants.***PREFIX\_POOL***;

**public static** String *PREFIX\_VMWARE\_VM\_NETWORK* = ConnectorConstants.***PREFIX\_VM\_NETWORK***;

**public static** String *PREFIX\_VMWARE\_VM\_STORAGE* = ConnectorConstants.***PREFIX\_VM\_STORAGE***;

**public static final** String ***NAME*** = **"VMwareConfigurationProvider"**;

*/\*\**

*\* Create an instance of a specific ConnectionConfiguration*

*\* This class should be properly annotated with JAXB mappings*

*\**

*\** ***@return*** *a new configuration specific to this provider*

*\*/*

@Override

**public** ConnectionConfiguration createConfiguration() {

**return new** VmwareConfiguration();

}

*/\*\**

*\* Return the class object for the implementing configuration class*

*\**

*\** ***@return*** *the implementing class for this configuration provider*

*\*/*

@Override

**public** Class getImplementingClass() {

**return** VmwareConfiguration.**class**;

}

*/\*\**

*\* Display name for Hypervisor*

*\*/*

@Override

**public** String getHypervisorDisplayName() {

**return** *HYPERVISOR\_VMWARE*;

}

*/\*\**

*\* Display name for Management Server*

*\*/*

@Override

**public** String getManagementServerDisplayName() {

**return** *MGMT\_SERVER\_VMWARE*;

}

*/\*\**

*\* Connector name, used in configuration file names.*

*\*/*

@Override

**public** String getConnectorName() {

**return** *CONNECTOR\_VMWARE*;

}

*/\*\**

*\* GWOS Application Type. Must match server side application type*

*\*/*

@Override

**public** String getApplicationType() {

**return** *APPLICATIONTYPE\_VMWARE*;

}

*/\*\**

*\* Prefixes based on Prefix Type. Special host names are prefixed according to a naming convention*

*\*/*

@Override

**public** String getPrefix(PrefixType prefixType) {

**switch** (prefixType) {

**case *ManagementServer***:

**return** *PREFIX\_VMWARE\_MGMT\_SERVER*;

**case *Hypervisor***:

**return** *PREFIX\_VMWARE\_HYPERVISOR*;

**case *Network***:

**return** *PREFIX\_VMWARE\_NETWORK*;

**case *Cluster***:

**return** *PREFIX\_VMWARE\_CLUSTER*;

**case *Storage***:

**return** *PREFIX\_VMWARE\_STORAGE*;

**case *DataCenter***:

**return** *PREFIX\_VMWARE\_DATACENTER*;

**case *ResourcePool***:

**return** *PREFIX\_VMWARE\_RESOURCE\_POOL*;

**case *VmNetwork***:

**return** *PREFIX\_VMWARE\_VM\_NETWORK*;

**case *VmStorage***:

**return** *PREFIX\_VMWARE\_VM\_STORAGE*;

}

**return null**;

}

**protected void** initPrefixMap(Map<String,InventoryType> prefixMap) {  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_HYPERVISOR*,

InventoryType.***Hypervisor***);  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_MGMT\_SERVER*,

InventoryType.***Hypervisor***);  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_NETWORK*,

InventoryType.***Network***);  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_CLUSTER*,

InventoryType.***Cluster***);  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_STORAGE*,

InventoryType.***Datastore***);  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_DATACENTER*,

InventoryType.***DataCenter***);  
 prefixMap.put(AmazonConfigurationProvider.*PREFIX\_AMAZON\_RESOURCE\_POOL*,

InventoryType.***ResourcePool***);  
 }

}

## Create a Unit Test

Write a unit test to test out reading and writing your configuration file and store the java file under the src/test/java source tree under the same package as your source code.

# Implement the Connector

The Connector’s main implementation is where you will write the code to retrieve virtualization inventories and metrics. Your connector is plugged into the CloudHub engine as a Spring component that must implement two interfaces:

1. MonitoringConnector - defines the virtualization monitoring methods to collect metrics
2. ManagementConnector - defines the virtualization inventory methods to discover hosts, services

Along with these interfaces, your connector will use a set of extension model classes to represent the monitoring and inventory data being processed. These base classes are the normalized representation of monitoring metrics in the CloudHub engine. In your implementation, you will often be populating and returning these common metric-model classes found in the org.groundwork.cloudhub.metric package:

* BaseHost - represents a host’s on a virtualization server. Hosts are often hypervisors
* BaseMetric - represents a measured metric on a Host or VM. Known as ‘services’ in GWOS
* BaseProperties - dynamic properties that are not modeled in common base model
* BaseQuery - a monitoring server specific reference or ‘query’ to a monitoring metric
* BaseSynthetic - a calculated metric, often using one or more queries to synthesize the value
* BaseVM - represents a virtual machine on a virtualization server
* MonitoringState - represents a snapshot of a single collection cycle for all monitored hosts, vms

## Implementation Steps

Get started by implementing your connector in your connector’s root package. For example:

org.groundwork.cloudhub.connectors.amazon.AmazonConnector

(1) Create a new class that implements MonitoringConnector and ManagementConnector interfaces….

For example: AmazonConnector.java

Recommend extending BaseConnector.java

(2) Create a new class in this package extending VemaBaseHost

For example AmazonHost

Provide metrics arrays

Provide config arrays

(3) Create a new class in this package extending VemaBaseVM

for example AmazonVM

Provide metrics arrays

Provide config arrays

Skeleton Code:

@Service(AmazonConnector.***NAME***)

@Scope(**"prototype"**)

**public class** AmazonConnector

**extends** BaseConnector **implements** MonitoringConnector, ManagementConnector {

**private static** Logger *log* = Logger.*getLogger*(AmazonConnector.**class**);

**public static final** String ***NAME*** = **"AmazonConnector"**;

**private** ConnectionState **connectionState** = ConnectionState.***NASCENT***;

**private** AmazonConnection **connection** = **null**;

**private** DataCenterInventory **inventory** = **null**;

*/\*\**

*\* Connect to a virtual monitoring server using configured credentials in the configuration*

*\**

*\** ***@param connection***

*\** ***@throws*** *ConnectorException*

*\*/*

**void** connect(MonitorConnection connection) **throws** ConnectorException;

*/\*\**

*\* Disconnects the connector from the virtual monitoring server, closing all resources and connections*

*\**

*\** ***@throws*** *ConnectorException*

*\*/*

**void** disconnect() **throws** ConnectorException;

*/\*\**

*\* Check the connection state of this connector to the virtual monitoring server*

*\**

*\** ***@return*** *the current state of the monitoring server*

*\*/*

ConnectionState getConnectionState();

*/\*\**

*\* Run a cycle of collecting metrics for a connector, creating a snapshot of monitored*

*\* state for all hosts, VMs, services, and metrics for this connector*

*\**

*\** ***@param priorState***

*\** ***@param hostQueries***

*\** ***@param vmQueries***

*\** ***@return*** *a snapshot of MonitoringState of all monitored objects in this data center*

*\** ***@throws*** *ConnectorException*

*\*/*

**public** MonitoringState collectMetrics(MonitoringState priorState,

List<BaseQuery> hostQueries, List<BaseQuery> vmQueries)

**throws** ConnectorException;

*/\*\**

*\* Gets the collection mode for this connector*

*\**

*\** ***@return*** *current collection mode settings*

*\*/*

CollectionMode getCollectionMode();

*/\*\**

*\* Set the collection mode for the this connector*

*\**

*\** ***@param mode*** *the new collection mode*

*\*/*

**void** setCollectionMode(CollectionMode mode);

## Add the Connector to the ConnectorFactory

In org.groundwork.cloudhub.connectors.ConnectorFactory, add case statements to the switch statements for your connector:

1. Add a case statement for your connector to the getMonitoringConnector() method
2. Add a case statement for your connector to the getManagementConnector() method
3. Add a case statement for mapToManagementServerPrefix method
4. Add a case statement for your configuration provider to the getConfigurationProvider() method

## Create a Unit Test

Write a unit test to test out your connector and store the java file under the src/test/java source tree under the same package as your source code. See any of the tests under the org.groundwork.cloudhub.connectors package. For examples see DockerTest.java or VMwareTest.java and follow the testing patterns found there.

# Profiles

Profiles store lists of measured metrics and their threshold values. These metrics are known as ‘services’ in GWOS status viewer. Profiles contain two lists of metrics:

1. Primary list of metrics - typically the hypervisor or controlling host metrics list
2. Secondary list of metrics - typically the virtual machines metrics list

For example, the VmWare profile consists of the following XML structure:

Profiles are stored on both the Groundwork Server, and on the CloudHub server. The Groundwork server profiles are actually templates. They are located under /usr/local/groundwork/core/vema/\*.xml. File names follow the format:

{virtualSystem}\_monitoring\_profile.xml

* Note that {virtualSystem} is always converted to lowercase
* Note the underscores

Profile Templates are maintained on the GWOS server. These templates have recommended default values. End users of CloudHub can associate a profile template with a configuration, and override these default values. CloudHub retrieves these profile templates over a Rest Service. Retrieving profile templates is built into CloudHub by going over this Rest API:

GET [http://gwos-server/api/profiles/cloudhub/{applicationType](http://gwos-server/api/profiles/cloudhub/%7bapplicationType)}

Where applicationType is your connector’s application type. You should not have to write any code to retrieve the profile. It’s all built into the CloudHub engine.

## Your Profile Template

You will need to coordinate with the Groundwork team to setup a profile template. Lets look at what a profile template consists of. All the default profile templates are stored in our source code repository under:

monitor-platform/agents/cloudhub/src/profiles/

Lets take a look at the VMware profile template structure:

<vema-monitoring>

<profileType>vmware</profileType>

<hypervisor>

…. Hypervisor metrics definitions go here

</hypervisor>

<vm>

…. VM metrics definitions go here

</vm>

</vema-monitoring>

The profileType must match your ProfileType enumeration in your Java code. The <hypervisor> container of metric definitions is your primary metric set, and the <vm> container of metric definitions is your secondary metric set.

There are two kinds of metric definitions:

1. Normal metric queries. The name of these queries match a monitor metric on the server
2. Synthetic metric queries. These all calculated fields. They start with “syn.” and use other metric values in their calculations. The definitions of synthetics are provided in Java code

Here is a metric definition for a hypervisor:

<metric name="summary.quickStats.overallCpuUsage"

description="Overall Hypervisor CPU Usage"

monitored="false" graphed="false"

warningThreshold="1500" criticalThreshold="2500" />

Here are synthetic metric definitions for a hypervisor:

<metric name="syn.host.cpu.used"

description="Number of CPUs used on this hypervisor"

monitored="true" graphed="true"

warningThreshold="75" criticalThreshold="95" />

<metric name="syn.host.mem.used"

description="MB of Memory used on this hypervisor"

monitored="true" graphed="true"

warningThreshold="90" criticalThreshold="95" />

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| name | The metric query name. Must match the query string in your extended BaseHost, and map to a query on the virtualization server. |
| description | Provide a description of the metric. It will be displayed in the CloudHub UI |
| monitored | true or false. Some metrics are not monitored and only used in synthetic calculations. Set this to true if you want to see the metric displayed in the Groundwork Monitor as a service |
| graphed | true or false. If you want to track the value of this metric over time in a time series database, set this value to true so that it can be graphed over time |
| warningThreshold | The Groundwork Monitor can set off events and alerts when this monitored metric reaches your specified warning level. Set this value to a number greater than or equal to zero to set off warning alerts. Setting it to -1 turns off warning alerts |
| criticalThreshold | The Groundwork Monitor can set off events and alerts when this monitored metric reaches your specified critical level. Set this value to a number greater than or equal to zero to set off critical alerts. Setting it to -1 turns off critical alerts |

## Metric Defaults in Java

In case CloudHub does not have access to a profile, you should add default values into your Java classes for all of your profile metrics. Additionally, all synthetics must be configured in your Java class.

Here is an example of a VMwareHost (hypervisor) extending BaseHost, and defining its default metrics,

**public class** VMwareHost **extends** BaseHost {

**private static final** BaseQuery[] ***baseMetricList*** =

{

…

**private static final** BaseQuery[] ***baseSyntheticList*** =

{

…

**private static** BaseSynthetic[] *baseSyntheticMaster* =

{

**new** BaseSynthetic(**"syn.host.cpu.used"**,

**"summary.quickStats.overallCpuUsage"**,

1.0,

**"summary.hardware.cpuMhz.scaled"**,

**false**,

**true**),

...

Similarly, you will want to do the same for your Virtual Machine metrics:

public class VMwareVM extends BaseVM {

…

## Defining Synthetics

Synthetics, or calculated values, must be defined in your derived BaseHost and derived BaseVM classes. Synthetics are calculated from the BaseSynthetic class:

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Example** |
| handle | The name of the synthetic. Reference this name in your profile files. | syn.host.cpu.used |
| lookup1 | The first metric to use in the calculation | summary.quickStats.overallCpuUsage |
| factor1 | A double value to multiple the value of lookup1 by. Used to scale the lookup1, as the value may need to be adjusted to MB or GB. | 1.0  1024 |
| lookup2 | The second metric to use in the calculation. After multiplying lookup1 by factor1, lookup1 is divided by lookup2 | summary.hardware.cpuMhz.scaled |
| fromTop | Boolean.  If set to true, calculate synthetic from top:  1.0 - (lookup1 / lookup2)  if (false)  (lookup1 / lookup2) | if fromTop == true  1 - ((lookup1 \*. 1.0) / lookup2)  else  ((lookup1 \*. 1.0) / lookup2) |
| toPercent | Boolean. If true, convert result from fromTop calculation to percentage | result \*= 100;  result = max(0.0, min(100.0, result ));  result = (double)((int)(result+0.49)); |

# User Interface

The CloudHub User Interface is written in Spring MVC and JSPs. The Spring MVC controllers are wired into all configurations and connectors, which are spring beans, via annotations. When writing a new controller, you will need to implement the following UI components:

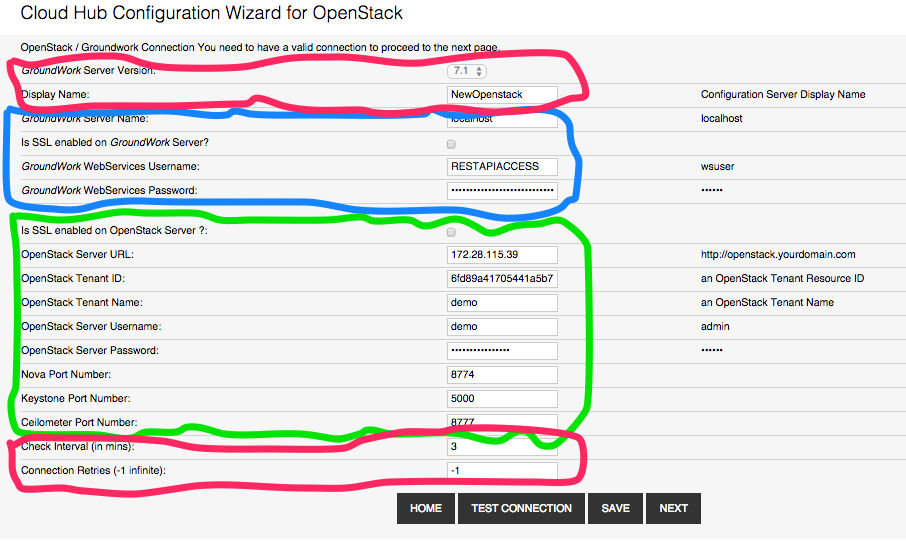
1. A configuration JSP
2. A Spring MVC Controller
3. A new image for the CloudHub List page
4. Add markup to index.jsp so uses can create new configurations from your connector
5. Add your UI to HomeController.updateConfiguration method
6. A profile template JSP

## Configuration JSP

All connectors must have a configuration page. The configuration allows you edit the three sections of a configuration file:

1. Common - common fields to all connector configurations
2. GWOS - fields specific to connecting to a GWOS server
3. Virtualization Connector specific fields

See the color-coded image below. The common(red) and GWOS (blue) fields should remain the same for your JSP. The virtualization connector fields (green) are specific to your connector, since they define how you will be connecting to the virtualization server. Some server requires passwords; others have different ways of authenticating.



A template is provided for you to get started. All of the configuration JSPs should be placed in the directory:

src/main/webapp/WEB-INF/pages/{connectorName}

To get started:

1. Create the directory for your connector under the pages directory
2. Copy the WEB-INF/pages/template/create-connection.jsp to

WEB-INF/pages/{connectorName}/

1. Edit that file, customizing your (green) fields for specific to your connection

When editing your configuration JSP:

* Case-sensitive replace all occurrences of $ConnectorName with your CamelCase Connector Name.
* Case-sensitive replace all occurrences of *$connector* with the lowercase name of your connector.

## Spring MVC Controller

Next, write a Spring MVC Controllers to handle all web actions for your configuration page. You should place your Spring MVC controller in the following package.classname:

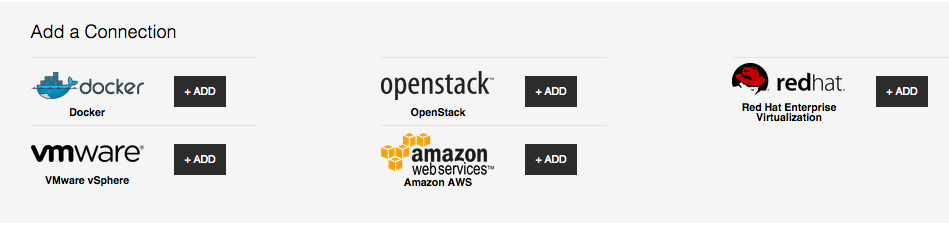
org.groundwork.cloudhub.web.{ConnectorName}Controller

To get started:

1. copy org.groundwork.cloudhub.web.ControllerTemplate.java {ConnectorName}Controller.java
2. Replace all occurrences of ‘cisco’ with your controller name
3. You will need to create a few constants in CloudHubUI.java
4. Add the logic to handle any new connection fields you may have added

## New Image

The landing page, index.jsp, is the home page of the CloudHub app. End users can create new CloudHub Connector configurations from this home by pressing the Add button. Groundwork staff will create an image for your connector and check it into the source code repository. For example, we have added Amazon AWS here:



## Index.jsp

Index.jsp is the main landing page for a CloudHub website. Groundwork staff will create your link for you:

<a class="button-l" href="/cloudhub/mvc/amazon/navigateCreateConnection">+ Add</a>

From here, you can start developing your connector.

## Home Controller

Modify the Home Controller, HomeController.java, to include logic to edit your configuration from the list view:

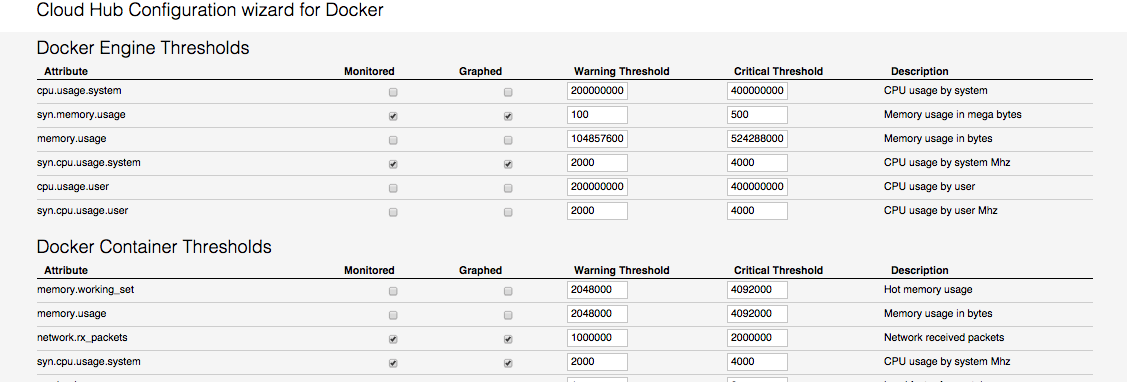
} **else if** (connectionConfig.getCommon().getVirtualSystem().equals(VirtualSystem.***AMAZON***)) {

modelAndView = **new** ModelAndView(**"amazon/create-connection"**, **"configBean"**,

connectionConfig);

## Profile Template JSP

All connectors must have a profile page. The profile page allows end users to customize the threshold values for metrics, and also enable or disable metrics for monitoring and graphing.



A template is provided for you to get started. All the configuration JSPs should be placed in the directory:

src/main/webapp/WEB-INF/pages/{connectorName}

This directory should already exist from when you configured your configuration JSP.

To get started:

1. Copy the WEB-INF/pages/template/assign-thresholds.jsp to

WEB-INF/pages/{connectorName}/

1. Edit that file, customizing your (green) fields for specific to your connection

When editing your configuration JSP:

* Case-sensitive replace all occurrences of $ConnectorName with your CamelCase Connector Name.
* Case-sensitive replace all occurrences of *$connector* with the lowercase name of your connector.

# Dynamic Properties

Here is a list of Groundwork Dynamic Properties that can be used in the system:

LastPluginOutput

LastStateChange

isAcknowledged

TimeUp

TimeDown

TimeUnreachable

LastNotificationTime

CurrentNotificationNumber

isNotificationsEnabled

isChecksEnabled

isEventHandlersEnabled

isFlapDetectionEnabled

isHostFlapping

PercentStateChange

ScheduledDowntimeDepth

isFailurePredictionEnabled

isProcessPerformanceData

RetryNumber

isAcceptPassiveChecks

isProblemAcknowledged

TimeOK

TimeUnknown

TimeWarning

TimeCritical

Latency

ExecutionTime

isServiceFlapping

isObsessOverService

ApplicationName

ApplicationCode

SubComponent

LoggerName

ErrorType

30DayMovingAvg

isPassiveChecksEnabled

AcknowledgedBy

AcknowledgeComment

Parent

Alias

RRDPath

RRDLabel

RRDCommand

CurrentAttempt

MaxAttempts

isObsessOverHost

ServiceDependencies

ExtendedInfo

Comments

CactiRRDCommand

RemoteRRDCommand

Notes

DeactivationTime

PerformanceData

ipaddress

Event\_OID\_numeric

Event\_OID\_symbolic

Event\_Name

Category

Variable\_Bindings

UpdatedBy

Location

ContactPerson

ContactNumber