4.1.x Composite Applications

Stratos is able to start up or terminate instances (single tenant) according to the demand. In Stratos terms this is referred to as subscription and unsubscription. Subscription will bring up a cluster, with the minimum number of instances, and unsubscription will terminate the instances. Prior to subscription, it is required to define and deploy the relevant cartridge in order to communicate to Stratos the details about the service that you will be providing, the IaaS that will be used, etc.

In previous Stratos releases, each cluster corresponding to a subscription was isolated and there was no connection between multiple clusters. However, in a real world use cases, there are requirements to have multiple clusters in a grouped manner, where one or more clusters can depend on some other clusters. Therefore, in this release composite applications support has been introduced and cartridge subscription has been made obsolete.

Composite application support, which is also referred to as cartridge grouping, provides the ability to deploy an application that has different service runtimes with their relationship and dependencies. Further, each of the service runtimes in the application can scale independently or jointly with the dependent services. Thereby, users can defined applications with its required service runtimes, deployment and auto scaling policies, artifact repositories and all dependencies in simple structured JSON file that Stratos can understand and provision all the required runtimes in a defined manner.

> Composite Application = Information to Create Multiple Clusters + Dependency Details

The various terminologies used with composite application support, which are groups, dependency, startup order, group scaling, dependence scaling, termination behaviors and metadata service, have been explained in-depth in the following sub-sections.

Cartridge groups

A cartridge group, is a metadata that you can define by grouping several cartridges together. Composite applications in Stratos, support nested groups within a group as inter-dependencies among group members. Cartridge groups define the relationship among a set of groups and a set of cartridges. The relationship among the children of a group can be start up order, termination behavior and any scalable dependencies. Writing a group definition provides the ability to re-use the same frequently used group as needed in different composite applications. See diagram - 01 for a sample group.

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

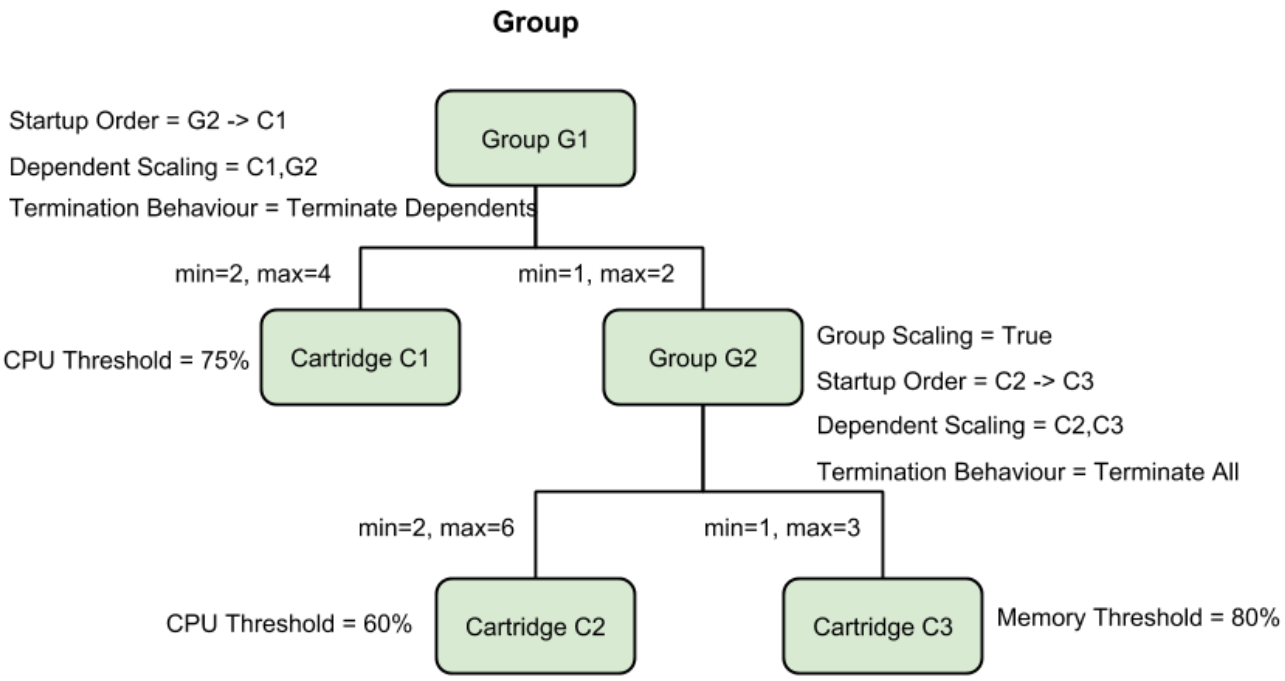


diagram - 01

The following is the structure of a basic service group JSON definition:

- `name`
- `groupscartridges`
  - `name`
  - `groups`
  - `cartridges`
  - `groupScalingEnabled`
  - `dependencies`
    - `startupOrders`
    - `scalingDependants`
    - `terminationBehaviour`
- `groupScalingEnabled`
- `dependencies`
  - `startupOrders`
  - `scalingDependants`
  - `terminationBehaviour`

Related Links

For information on all the properties that can be used in a cartridge group definition, see the Cartridge Group Resource Definition.

Startup order

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

Within a group or an application you can define the startup order that needs to be maintained between two or more dependent cartridges or groups.

Example: Based on diagram-01:

- Group G1 has two members, cartridge C1 and group G2, and the startup order among these two members are G2->C1, which means that group G2 has to wait until cartridge C1 is created first and comes into active mode.
- In group G2, the startup order is C2->C3, which means cartridge C2 has to wait until cartridge C3 is available.

Termination behavior

Termination behavior allows you to define the action that needs to be taken if any dependency runtime failure occurs in an application. Failures are identified as not satisfying the required minimum member count. The following are the three termination behaviors that are supported:

- Termination all
- Termination dependents
- Termination none

Example: Based on diagram-01:

- As the termination behavior for group G2 is terminate all, if either cartridge C2 or C3 failures, all members will be terminated and they will be re-created with the defined startup order.
- As the termination behavior is terminate dependents in group G1, if cartridge C1 fails, group G2 will be terminated, because group G2 is dependent on C1 in the startup order. However, if group G2 fails, cartridge C1 will not be terminated, because it is not dependent on G2.

Dependent scaling

When dependent scaling is defined among members (cartridge or a group), and scaling (scaling up or down) is taking place for any of the members, all other dependent members will also scale in order to maintain the defined ratio.

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

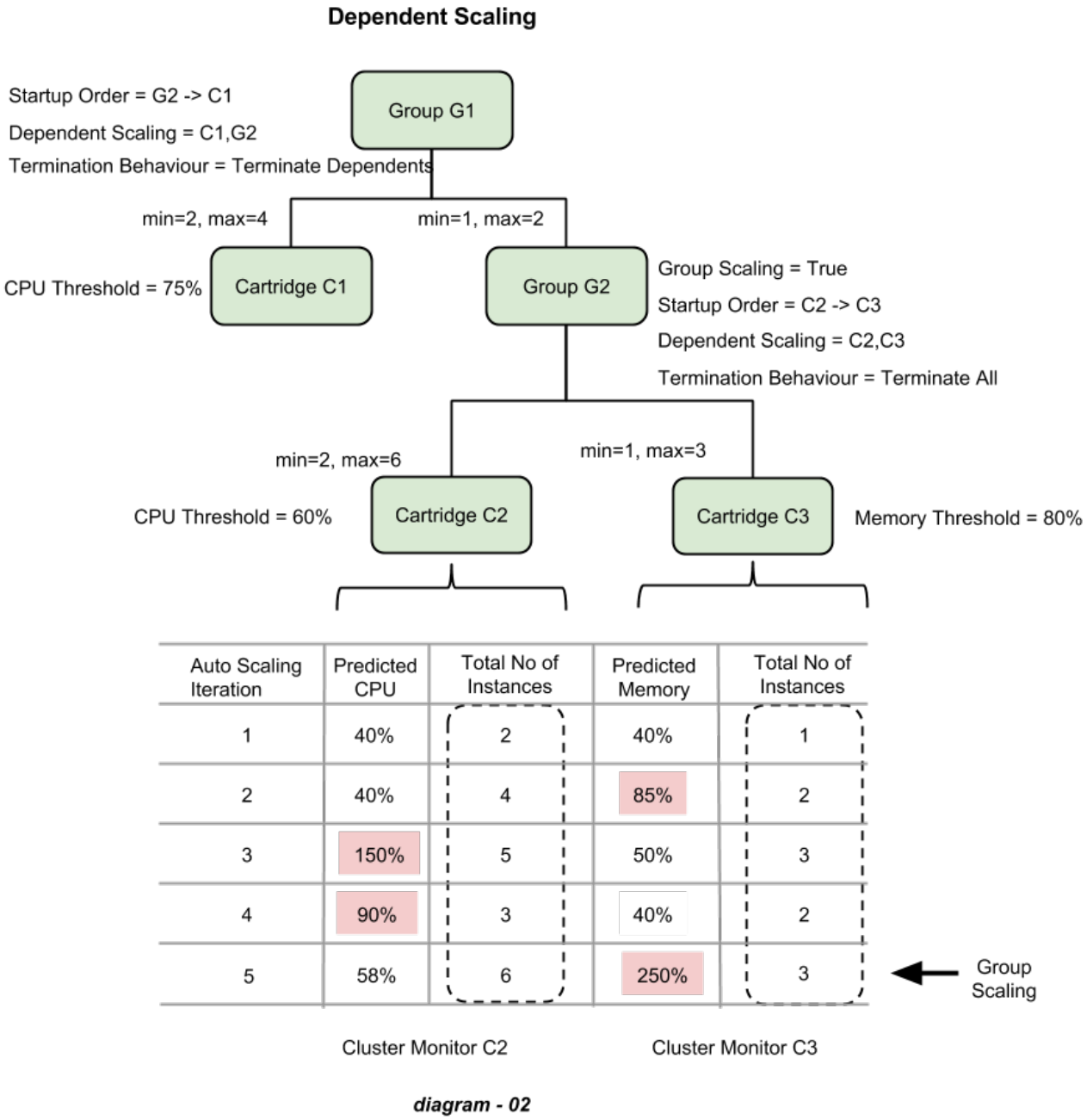


diagram - 02

Example: Based on iterations 1 to 4 in diagram-02.

Group G2, cartridge C2 and cartridge C3 have been defined in the dependent scaling list. Four auto-scaling iterations are considered in this example to illustrate how scaling up and scaling down takes place with dependent scaling. Note that cartridge C2 and C3 have two different auto-scaling policies, one is based on the CPU utilization and other is based on memory consumption. Furthermore, in cartridge C2 the number of minimum instances in the cluster is 2 and the maximum number of instances are 6; while. in cartridge C3 the minimum number of instances is 1 and the maximum number of instances are 3.

- **Iteration 1 :** Cartridge C2 and C3 are both below the threshold values. Therefore, the minimum instance count that is defined, which C2=2 and C3=1, is maintained.
- **Iteration 2 :** In cartridge C3 the predicted memory consumption is 85%, which exceeds the threshold. The following equation needs to be used to calculate the required C3 cartridge instances that are required to handle this predicted value.

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

Required instances count formula

$$\text{required Instances} = \frac{\text{predicted value}}{\text{threshold value}} \times \text{defined minimum Instance}$$

The required C3 instances = 85/80 * 1 = 1.06. When dealing with instances, the minimum required instance count is one. Therefore, in situations like this we need to round off the required instance count to the nearest instance, which means that we need 2 C3 instances to handle the predicted load.

As C2 depends on C3 with the a 2:1 (C2 minimum instances : C3 minimum instances) ratio and the C3's new instance count is 2, Stratos will create 2 new C2 instances, to make a total of 4 C2 instances in order to maintain the defined ratio. Furthermore, note that at the time C2 was under the threshold value, but priority is always given to dependent scaling decisions.

- **Iteration 3:** In this case, C2's predicted CPU consumption is 150%. When the above instances count formula is applied, the required instances count for C2 is 150/60 * 2 = 5. Since C3 is dependent on C2 with a ratio 2: 1, C3 will increase its instance count to 3.

- **Iteration 4:** In this scenario, C3 is under the defined threshold, but C2 has exceeded the threshold. When the above instances count formula is applied, the required instances count for C2 is 90/60 * 2 = 3. This means that we need to scale down C2's instances to 3 instances. Based on the dependent ratio, C3 should be in scaled down to 2 instances. Since C3 is below the auto-scaling threshold scaling down takes place.

Group scaling

If group scaling is defined as true within a group definition, this means that the group itself can be scaled in a situation when a member reaches its maximum instance limit and also based on the dependancy.
Example: Based on iteration 5 in diagram - 02.

- **Iteration 5 :** In this scenario, C3's predicted value is 250. When the above instances count formula is applied the required instances count for C3 is 250/80 * 1 = 3.125, which means 4 C3 instances are needed. However, as C3's maximum number of instances count is 3, it is not possible to scale more than 3 C3 instances within the G2 group. In this situation, as group scaling is enabled, Stratos will create a new G2 group instance, which has 2 instances of C2 and one instance of C3. As a result, there will be two G2 group instances G2-1 and G2-2. This has been illustrated further in the diagram below:

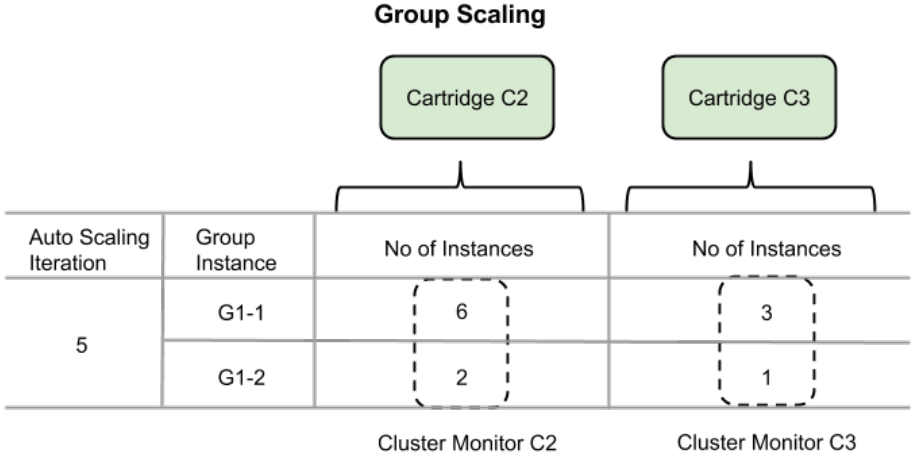


diagram - 03

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

Application JSON

An application JSON is a structured JSON, that you can define the runtime of your application by using, cartridges, cartridge groups, dependencies, artifact repositories and auto-scaling policies. The application JSON can be converted into an application template, that can be reused to deploy the same application with different deployment patterns. The deployment policy is the way to define the deployment pattern (e.g., high availability, disaster recovery, cloud bursting, multi-cloud with 4 nines or 5 nines etc.).

The following is the structure of a basic application JSON definition:

- `applicationId`
- `alias`
- `components`
    - `groups`
        - `alias`
        - `min/max`
        - `group scaling enable/disable`
        - `cartridges`
            - `min/max`
            - `subscribable info`
        - `groups`
            - `alias`
            - `min/max`
            - `group scaling enable/disable`
            - `cartridges`
                - `min/max`
                - `subscribable info`
    - `cartridges`
        - `min/max`
        - `subscribable info`
    - `dependencies`
        - `startup order`
        - `termination behavior`
        - `dependent scaling`

Application deployment policies

In Stratos, deployment policies are used to define the deployment patterns that need to be used. In grouping, deployment policies are supported at the group level or at the cluster level. Global deployment policies provide a single place to define the deployment policies that correspond to the children (nested groups). Thereby, each application will have a single deployment policy, which defines all the children deployment policies as well. There are many advantages of defining global deployment policies, which are as follows:

- The same applications can be deployed in high availability (HA) or in a burst manner using different deployment policies.
    - This enabled the actual VMs to be started after deploying the deployment policy rather than starting it once the application has been deployed.
    - The deployment policy will always be coupled with an application.
- It is not needed to define multiple deployment policies at the cluster level or group level.
- The partition definition details can also be defined in the deployment policy itself.

The following is the basic structure of an application deployment policy:

- `id`
- `applicationPolicy[1..1]`
    - appId
    - networkPartition[1..n]
        - id

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

        - activeByDefault
        - partition[1..n]
            - `id`
            - `provider`
            - `properties[1..n]`
- `childPolicies[1..n]`
    - childId (Group alias or cartridge alias)
    - networkPartition[1..n]
        - id
        - min
        - partition[1..n]
            - id
            - max

Descriptions on the terminology used in deployment policy definitions are as follows:

- **Network Partition**
  A network partition is a logical partition that you can define, in a network bound manner. Thereby, Stratos will map the network partitions with the IaaS regions. The abbreviation for a network partition is as follows: NP
  Example:
  `NP1: EC2-US-WEST`
  `NP2: EC2-US-EAST`
  `NP3: OPENSTACK-NY-RegionOne`

- **Partition**
  Partitions are also an individual or a logical group within a network partition in a fine grained manner. The abbreviation for a partition is as follows: P
  Example:
  `NP1:P1 -> US-WEST -> us-west-1 (N. California)`
  `NP1:P2 -> US-WEST -> us-west-2 (Oregon)`

- **applicationPolicy**
  The application policy will have the definition of all the network partitions and partitions that will be used throughout the application.

- **activeByDefault**
  If this property is true, it means that the network partition will be used by default. However, if this property is false, it means that the network partition can be used when all the resources are exhausted (in bursting).
- **childPolicies**
  Each child policy will refer the network partition and the relevant partition from the applicationPolicy to define their own deployment pattern. Please note that, if you define a childPolicy by referring to group, the underlying cluster (cartridge runtime)/group will inherit the same policy.

- **max**

  Maximum no of instances that can be handled by a partition.

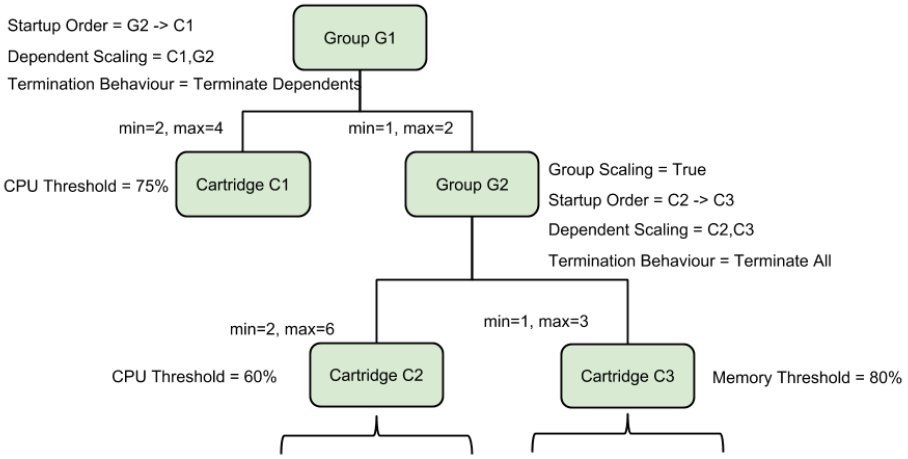
  In terms of a group: Maximum group instances that can be in a partition.
  In terms of a cluster: Maximum number of members that can be maintained for a cluster instance in a partition.
- **algorithm**
  Stratos support the following two algorithms: round robin and one after another. These algorithms can be used depending on the required scenarios.
  Example: Based on diagram-04, which has two child policies applied in the cartridge runtime level.

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

Child policy : sample1
Partitions : P1, P2
P1 Max : 4
P2 Max : 3
Algorithm : Round robin

Child policy : sample2
Partitions : P3, P4
P3 Max : 2
P4 Max : 3

Algorithm : One after another



Auto Scaling Iteration	Predicted CPU	Instances		Predicted Memory	Instances	
		P1	P2		P3	P4
1	40%	1	1	40%	1	0
2	40%	2	2	85%	2	0
3	150%	3	2	50%	2	1
4	90%	2	1	40%	2	0

diagram - 04

In diagram-04 you can see that C2 has applied the sample1 child policy and all C2 instances are created in P1 and P2 in a round robin manner. C3 has applied the sample2 child policy and as the defined algorithm is one after another, all C3 instances are created in P3 until P3 reaches its maximum instance count, which is 2. Thereafter, C3 instances are created in P4.

- **Iteration 1 :** Cartridges C2 and C3 are both below the threshold values. Therefore, the minimum instance count will be maintained. As C2 is based on the round robin algorithm and because its minimum instance count is 2, one instance will be created in P1 and another instance will be created in P2. As C3 is based on the one after the other algorithm and because its minimum instance count is 1, the instance will be created in P3.
- **Iteration 2 :** In cartridge C3 the predicted memory consumption is 85%, which exceeds the threshold. When the above instances count formula is applied, the required instances count for C3 is 85/80 * 1 = 1.06. Therefore, P3 will increase its count to 2. As C2 is dependent on C3 with a 2:1 (C2 minimum instances : C3 minimum instances) ratio, C2 will need to increase its instance count to 4. As C2 is based on the round robin algorithm, the instance count will be increased by 1 in P1 and P2 .

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

- **Iteration 3:** In this case, C2's predicted CPU consumption is 150%. When the above instances count formula is applied, the required instances count for C2 is 150/60 * 2 = 5. As C2 is based on the round robin algorithm and because the instance count was last adjusted in the P2 partition, the instance count in P1 will be increased by 1. Based on the dependent ratio, the instance count in C3 needs to be increased to 3. As P3 has reached its maximum instance count limit, one instance will be created in P4.

- **Iteration 4:** In this scenario, C3 is under the defined threshold. However, C2 has exceeded the threshold. When the above instances count formula is applied, the required instances count for C2 is 90/60 * 2 = 3. This means that we need to scale down C2's instances to 3 instances. As C2 is based on the round robin algorithm and because the instance count was last adjusted in the P1 partition, the instance count in P2 needs to be scaled down to 1 instance. Based on the dependent ratio, C3 should be in scaled down to 2 instances. As C3 is based on one after the other algorithm, the instance in P4 will be terminated.

Metadata Service

As a composite application has a collection of cartridges, there will be multiple cartridge instances that will boot up. Thereby, the information needs to be shared among the cartridges within a composite application. The metadata service is a web app in Stratos that is responsible for data sharing among the runtime services that may require a composite application. The metadata service acts as a central placeholder for all cartridge instance related metadata. Thereby, cartridges can publish metadata to the metadata service, so that other interested parties can retrieve the data when required.
The metadata are stored as key value pairs in the metadata service. Multiple values may be associated with one key; however, each key needs to have at least one corresponding value. When you create an application, Stratos Manager will create a JW (JSON web token) OAuth token that includes the application ID as a claim. This token will be passed into the cartridge runtime instance or container as a payload. The cartridge agent uses this token to authenticate and authorize against the application whenever data is retrieved from or published to a metadata service.

Many REST APIs have been exposed to add, fetch and delete metadata in the metadata service. Therefore, when there are multiple cartridges all the instances will invoke REST APIs to publish their details to the metadata service. When information sharing is required between the cartridge instances in a composite application, the respective cartridge instance will invoke a REST API to retrieve the required metadata.

Metadata client

The metadata client is used when third party Stratos components (e.g., Cloud Controller, Auto-scaler, Stratos Manager) need to communicate with the metadata service. However, other clients (e.g., Cartridge Agent) that have the JW OAuth token can communicate with metadata service without the use of the metadata client. The metadata client is a Java API that wraps the client to transform the requests and responses. Therefore, third party Stratos components will be able to use the metadata client to add, fetch and delete metadata in the metadata service. For example, the metadata client will be used when the Git repository URL, which is in the Cloud Controller, needs to be added to the metadata table.

Related Links

- See Metadata REST API Reference to learn how to work with the metadata REST API.
- See Sample 1 to see how the metadata service works.
- See the hangout on the metadata service to learn more on the metadata service design and whats under the hood.

How it works?

The following flow explains how to get a composite application working with Stratos:

1. Add a network partition.

   For more information on how to add a network partition via REST API, CLI or UI, see Working with Network Partitions , and see Network Partition Resource Definition to learn about all the properties that can be defined in a network partition.

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

2. Add a deployment policy.

   For more information on how to add a deployment policy via REST API, CLI or UI, see Working with Deployment Policies , and see Deployment Policy Resource Definition to learn about all the properties that can be defined in a deployment policy.

3. Add an auto-scaling policy.

   For more information on how to add an auto-scaling policy via REST API, CLI or UI, see Working with Auto-scaling Policies , and see Auto-scaling Policy Resource Definition to learn about all the properties that can be defined in an auto-scaling policy.

4. Add a cartridge.

   For more information on how to add a cartridge via REST API, CLI or UI, see Working with Cartridges , and see Cartridge Resource Definition to learn about all the properties that can be defined in a cartridge.

5. Add a cartridge group.

   For more information on how to add a cartridge group via REST API, CLI or UI, see Working with Cartridge Groups , and see C artridge Group Resource Definition to learn about all the properties that can be defined in a cartridge group.

6. Add the application.

   For more information on how to add a application via REST API, CLI or UI, see Working with Applications, and see Application Resource Definition to learn about all the properties that can be defined in an application.

7. Add the application policy.

   For more information on how to add a application policies via REST API, CLI or UI, see Working with Application Policies , and see Application Policy Resource Definition to learn about all the properties that can be defined in an application policy.

8. Deploy the application.

   For more information on how to deploy an application via REST API, CLI or UI, see Working with Application Deployment.

After the composite application is deployed, all the clusters that belong to the composite application are brought up by Stratos, based on the dependency information provided in the cartridge group definition and in the application definition.

Copyright © 2015 The Apache Software Foundation, Licensed under the Apache License, Version 2.0