

Configuration and Deployment

This document explains the related concepts of “Configuration” and “Deployment”. “Configuration” is how you connect the Vantiq servers on your systems into a network, and “Deployment” is how you can provision these systems with the appropriate set of Vantiq resources (Types, Rules, Sources, etc.)

Connecting systems using Nodes

A key feature of the Vantiq server is its ability to connect systems into a federated network. Each connection is defined by an instance of the **Node** object. A Node object tells system A about the existence of system B, as well as how it can be accessed and what capabilities it can provide. Once one system “knows” about another system then all of its resources can be accessed remotely; virtually any resource (for which you are authorized) can be read, written or deleted.

To accomplish this, Node objects contain a URI which provides an endpoint address for the server as well as valid credentials used to authenticate requests. This information is used to “log in” to the server, which means that system A has access to the same resources of system B they **would** have if they logged in locally. (And of course, the same restrictions.) For convenience you can also assign an arbitrary human-readable “name” which can be used to identify the server.

Note that this relationship only goes one way; if you define a node on system A describing access to system B this does **not** mean that system B knows about system A. If you want the relationship to be bi-directional, then system B must contain a Node of its own describing system A.

Nodes are a resource like any other and can be created or updated using the REST API or using the Vantiq IDE. (There is a “Nodes” menu item under “Deploy”.)

This simple model makes it possible to build a network of Vantiq servers into any topology that makes sense for your environment.

Tagging systems with properties

A Node can also contain information about the capabilities or characteristics of the system. It contains a field called **ars_properties** which can hold an object that conveys any suitable information which describes the system in your environment. For example, you might define it like this:

```
{
  "type": "Factory",
  "state": "CA",
  "size": "Large"
}
```

The **ars_properties** object can contain any information which is meaningful in order to categorize the system in various ways. This is useful because any VAIL code you write will probably **not** want to refer to systems by their URI. Instead you will want to do things like “Execute a Procedure on all the Factory servers in California”. If new Nodes are added which meet your criteria, they will automatically be included in the affected set without the code changing at all.

This is accomplished by the existence of the **PROCESSED BY** clause on various statements in the VAIL language; something like this

```
EXECUTE MyProcedure(p1,p2)
PROCESSED BY ((ars_properties.type == "Factory") && (ars_properties.state == "CA"))
```

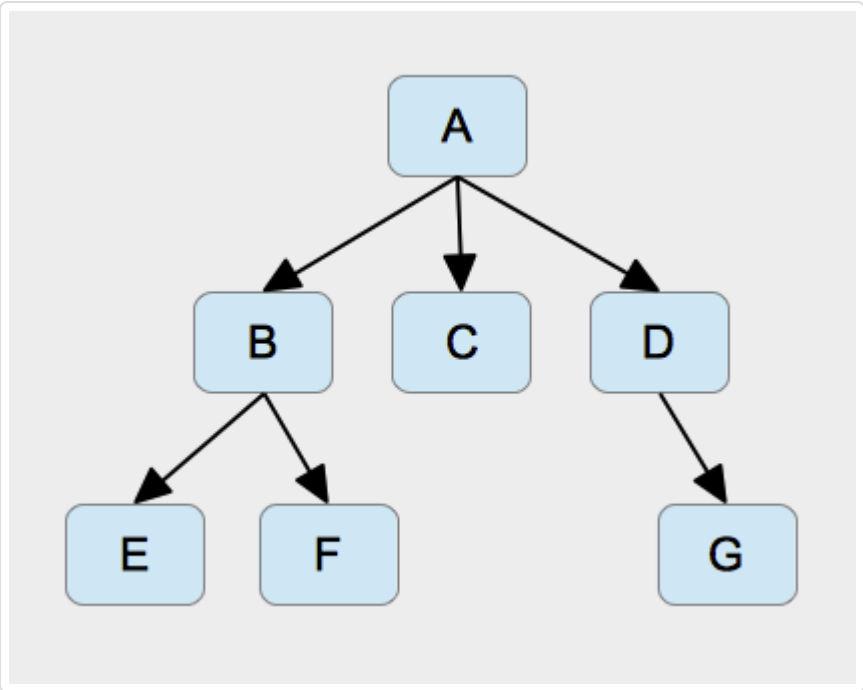
PROCESSED BY is like a “WHERE” clause that queries against all the Nodes defined on the local system. Any Node which matches your criteria will be told to execute *MyProcedure* with the given parameters. To specify that the procedure should be executed by all remote nodes replace the PROCESSED BY constraint with the keyword “ALL” like this:

```
EXECUTE MyProcedure(p1,p2) PROCESSED BY ALL
```

You can also use *tag values* in a PROCESSED BY clause. For example, if you had a tag defined on your node with value “tagB”, you could use a PROCESSED BY like this (note that the system adds a leading # to tag values):

```
EXECUTE MyProcedure(p1,p2)
PROCESSED BY ars_properties.tags == "#tagB"
```

It is important to note that this **only** applies to Nodes that are defined on your local system. For example, suppose your systems were arranged in a “hierarchical” topology like this:



In this case on system A you have defined 3 Nodes which describe B, C and D. In addition system B knows about E and F, and system D knows about G. For the purposes of this example assume **all** the nodes are defined as “California Factories”. If you ran code on system A like the “EXECUTE” example above it would execute MyProcedure on **only** systems B, C and D. PROCESSED BY operations are not automatically “relayed” down the chain.

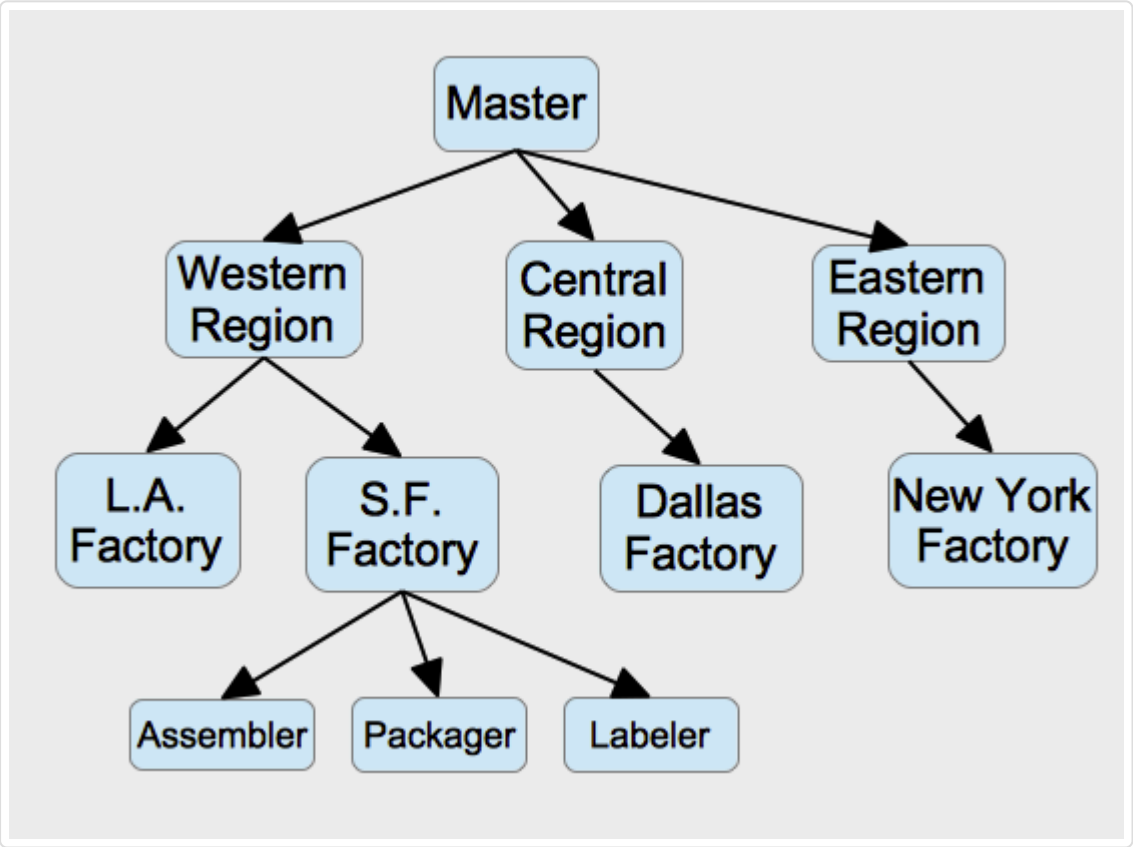
The PROCESSED BY clause is available on other VAIL statements as well, such as SELECT, INSERT, UPDATE, DELETE, CREATE and DROP.

Building networks

By defining Nodes on Vantiq servers you can create networks such as the one shown above. Remember that each connecting line with an arrowhead means that a Node has been defined on the system it emanates from.

Hierarchical (“Tree”) Network

The most common topology is probably a “tree” or “hierarchical” arrangement such as the example below.



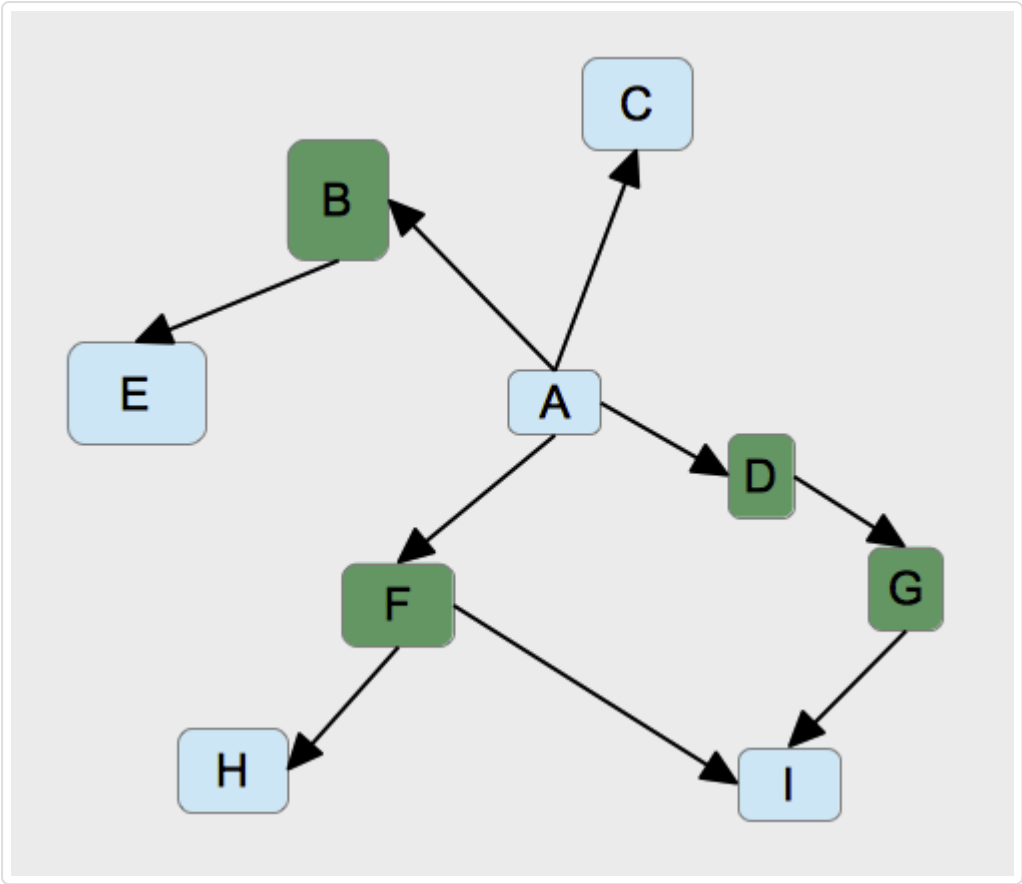
In this example your systems are arranged in a hierarchy where each “level” contains some known “components” inside it. For example the “Master” server connects to 3 Regional servers. Each Regional server owns one or more “Factory” servers. And each Factory owns the various kinds of “manufacturing machines” inside it. (Only one Factory is shown for simplicity.)

Each Node only needs to know about the Nodes it controls and is directly connected to. The Master server doesn’t need to know about how many “Assemblers” live below it, only the Regions it owns.

In this example the Nodes do not know about their “parents” (i.e. there are no lines with an arrowhead pointing to the level above). But there is no reason you could not build that way if there was a good reason to do so.

“Mesh” Network

Another less common topology is sometimes called a “mesh” network where there are various types of servers arranged with no particular order imposed on how they are connected, something like this:



Perhaps this might be a collection of “Blue Kiosks” and “Green Kiosks” that perform different functions but are connected to whatever node happens to be closest (rather than out of some sort of organizational scheme). In this case there is no predicting what type of server any given type might be connected to.

Configurations

Whatever the topology, each Vantiq server in your installation must have access to the set of “resources” that it needs to perform its function. This set is called a “Configuration” and can consist of zero or more resources of the following types:

- Types
- Rules
- Sources
- Procedures
- Topics
- Configurations

You can create a Configuration using the REST API or the Vantiq IDE. Note that a Configuration does not “contain” the things on this list, just a **reference** to them (i.e. just their name).

The process of getting the needed resources to the proper Nodes in your environment is called “deployment”. The first step in doing a deployment is building one or more Configurations that describe the “list of parts” needed by each Vantiq server. The way you approach this depends on the organization of your network.

Deploying a Hierarchical (Tree) Configuration

For a hierarchical environment (such as the example above) that probably means you build one Configuration for each kind of “thing” in the network. That means 5 different kinds of Configurations, where each Configuration contains the Configurations for the parts inside it:

Region Configuration

- Factory Configuration
 - Assembler Configuration
 - Packager Configuration
 - Labeler Configuration

For example, the Factory Configuration would contain all the parts needed by the Factory, **as well as** the Assembler, Packager and Labeler Configurations (each of which contains parts of their own.) The “union” of all the Configurations makes up the “payload” for the deployment process.

Note that there does not need to be a “Master” Configuration; the “Master” system is simply the place the deployment starts; it is simply the server that “knows” (i.e. has Nodes defined for) all the Regional servers. This means the Master system must contain all the parts needed by the Configuration to be deployed (and all the sub-parts needed by the sub-Configurations, etc.)

Each Configuration also contains a property called the “provisioning constraint” which is equivalent to a PROCESSED BY clause. It tells the deployment process **which** servers the Configuration should be applied to. In this case, that means it must be able to select the “Region”, “Factory”, “Assembler”, “Packager” or “Labeler” systems.

To summarize, here are the steps you might go through to create the set of Configurations to accomplish the deployment:

1. Start by logging in to the “Master” system; this is probably the system where you developed all the resources and which has Nodes for all the Regions to be targeted.

- 2025/12/22 10:35
- Deployment Reference Guide -
2. For each Configuration, identify the “parts list”, i.e. the set of Resources (Rules, Types, Sources, etc.) that are required.

3. Create Configurations for each category of systems in the hierarchy, adding references to the required parts. (It is generally easiest to start at the “bottom” of the tree and work your way up, going from Assembler, Packager and Labeler to Factory and then Region. This is because the Factory Configuration will **contain** the Configurations for each of its “children”.) For each Configuration specify a provisioning constraint that will target it at the proper servers. (e.g. the “Factory” Configuration should have a provisioning constraint that targets it at only the Factory servers.) Work your way up the tree until all Configurations are complete.

For example, this screen shows how you might create the “Assembler” Configuration:

Configuration: New Configuration

Name:

Assembler

Description

Description

Provisioning Constraint:

1

`{"type": "Assembler"}`

Provision as Tree?

☒

+

Rule

+

Source

+

Type

+

Procedure

+

Topic

+

Visual Rule

+

Configuration

+

Client

+

RCS Request

+

RCS Payload

+

Collaboration Type

Resource

Resource Type	Name
Rule	AssemblerRule
Type	Instructions

After all the Configurations are complete the “list” page might look like this:

Configurations

Filter by Name or #tag

Refresh

New

Delete All

Delete

Edit Tags

<input type="checkbox"/>	Name ↑↓	Tags	Actions	Description
<input type="checkbox"/>	Assembler		<div><div>▶▶▶▶</div><div><div>⌂</div><div>🗑</div></div></div>	
<input type="checkbox"/>	Factory		<div><div>▶▶▶▶</div><div><div>⌂</div><div>🗑</div></div></div>	
<input type="checkbox"/>	Labeler		<div><div>▶▶▶▶</div><div><div>⌂</div><div>🗑</div></div></div>	
<input type="checkbox"/>	Packager		<div><div>▶▶▶▶</div><div><div>⌂</div><div>🗑</div></div></div>	
<input type="checkbox"/>	Region		<div><div>▶▶▶▶</div><div><div>⌂</div><div>🗑</div></div></div>	

Page 1 of 1

< Back

1

Next >

Once the need Configurations have been created you are ready to deploy. You start the process by clicking one of the “deployment” Actions on the topmost Configuration in the hierarchy (“Region”, in this example.)

Note that there are two “deployment” Action buttons:

- The single right-arrow does the deployment with minimal, “terse” reporting.

- The double right-arrow does the deployment with maximum, “verbose” reporting.

Unless your deployment is huge (and you want to minimize network traffic), you should generally choose the “verbose” deployment, which gives you maximum feedback about what happened during the process.

Here is an overview of the deployment process:

1. From the “Configuration List” page in the Vantiq IDE click the “Deploy” action for the “topmost” configuration (Region, in this case)
2. The deployment process begins, running on the local “Master” server; first it will propagate all the resources required by the “Region” level and below to all the “Region” servers. (This includes all the “sub-Configurations” and their requirements as well.) When done the deployment process will be restarted on all the “Region” servers, causing the next level of Configurations (Factory) to be propagated to **their** servers.
3. The process continues, shedding the outer layer of Configurations at each step as it moves down the tree. Deployment will “bottom out” on the “leaves” of the tree when there are no further sub-Configurations to propagate.

Some notes about this process-

Each “layer” of the tree only receives the “parts” it needs, as well as any parts which must be passed on to the next level. At the “leaves” (such as “Packager”), the systems only receive the resources they need, and nothing else. The “payload” gets smaller as you move down towards the leaves of the tree.

The deployment process is implemented by using the PROCESSED BY clause against a Configuration’s “provisioning constraint”. The constraint may apply to zero or more servers. If there are more than one servers targeted this process will happen **simultaneously** on all the targets (since each is running on its own server). For example if a Region has 10 Factories all 10 will actually be deployed at the same time.

As the process works it way down the tree, it will gather status information about each step and report back. At the end of the deployment process the entire “log” will be shown; if there were any failures (due to non-responsive systems, for example) they will be noted in the log.

Note that because of the way Configurations are nested, the “Region” Configuration really knows nothing about the details of the “Factory” Configurations it contains.

Note that the Regional servers do **not** have a direct connection to the Packagers (for example) and they don’t need one. (This is in contrast to the PROCESSED BY clause, which only applies to the Nodes a server has direct connection to.) The deployment process itself handles the “relay” down the tree, so each level only needs to know about the children it owns directly.

Deploying a Mesh Configuration

For a “mesh” environment you would generally build one “container” Configuration which contained nothing but a sub-Configuration for each kind of “thing” it might find in the network. That means 3 different kinds of Configurations, where each Configuration contains the Configurations for the parts inside it:

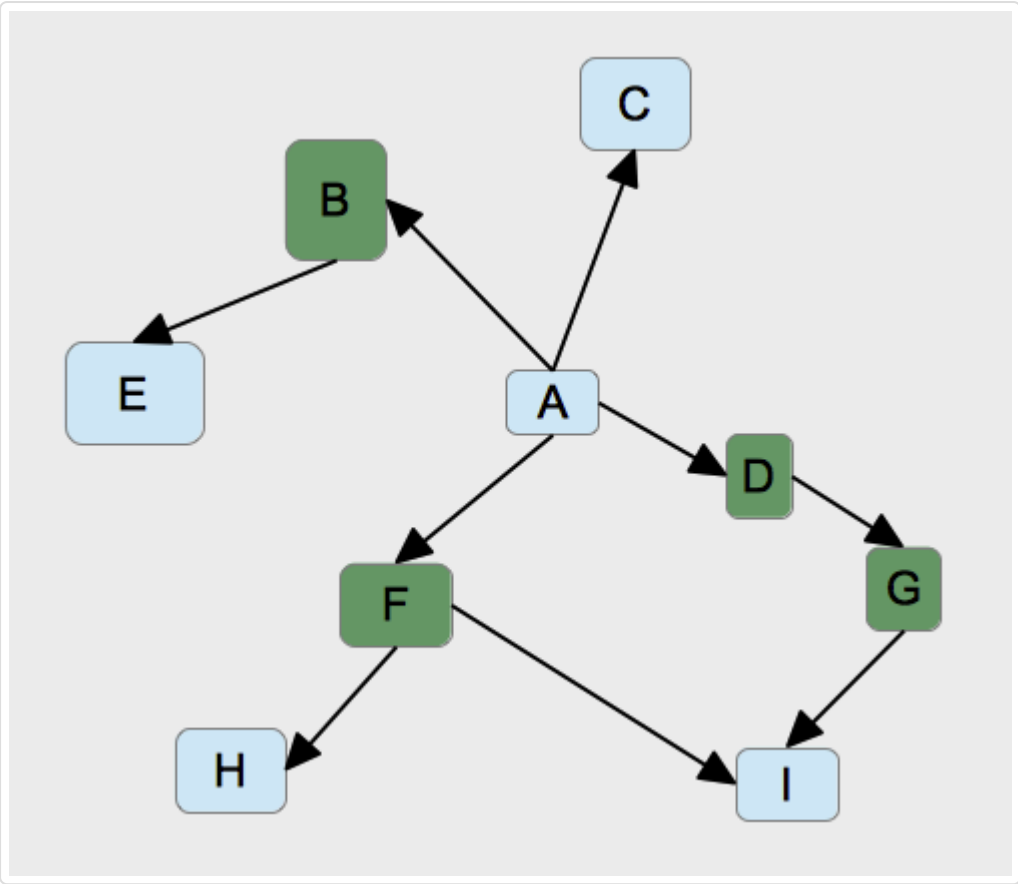
Container Configuration

Blue Kiosk Configuration

Green Kiosk Configuration

The deployment process needs to know that this is a “mesh” rather than a “tree”, so you must turn **off** the “Provision as Tree” checkbox when creating the Container configuration. You must also set the Container’s provisioning constraint so it will visit all the Blue and Green Kiosk nodes, perhaps like this:

```
{
  "ars_properties.kiosk": {"$in":["blue","green"]}
}
```



The Blue and Green Configurations must have provisioning constraints that target them at the appropriate servers such as this for Blue:

```
{
  "ars_properties.kiosk": "blue"
}
```

and this for Green:

```
{
  "ars_properties.kiosk": "green"
}
```

The deployment process for the Container Configuration in the “mesh” case is similar to what we saw above but the details are a little different.

1. From the “Configuration List” page in the Vantiq IDE, click the “Deploy” action for the “Container” configuration.
2. The deployment process begins, running on the local “Master” server; it will propagate all the resources required by the Container’s sub-Configurations to any server marked “blue” or “green” (because that’s what the Container’s provisioning constraint asks for.) If these configuration’s contained Rules only the ones appropriate to the particular server type will be activated. When done, the entire process will be restarted on all the known “blue” and “green” servers by handing off the entire Container to each one.
3. The process continues, propagating the same set of resources throughout the mesh. The process will end when the entire mesh has been visited and updated. (Once a node has been configured it won’t be visited again, which avoids a problem if the mesh has “loops”.)

Some notes about this process-

Each node of the mesh receives **all** the parts required by any potential server, since we never know what the next server might require. Thus the “payload” stays the same size all the way through the process.

As the process works it way through the mesh it will gather status information about each step and report back. At the end of the deployment process the entire “log” will be shown; if there were any failures (due to non-responsive systems, for example) they will be noted in the log.

It is assumed that there must be a natural starting point for the mesh, such that by following all the links all the nodes would be visited. (In the example above that would system “A”.) The “Master” server (where the deployment process is initiated) must have access to this starting node. If the mesh is actually broken up into discrete “islands” then the Master server must have Nodes defined for all the starting points that will allow the entire mesh (or in this case *meshes*) to be visited.

Undeployment

You may also “undeploy” a configuration; this makes an attempt to remove all the resources from all the systems that were created during the “deployment” process. (Leaving the systems in their original state before the deployment was done.)

You “undeploy” a Configuration by clicking the single or double **left**-arrow from the Configuration list. (These Action button are found immediately to the right of the “deployment” Actions.)

Activation Constraints

“Rules” and “Sources” are different from other resources in that they can either be “active” or “inactive”. Rules and Sources deployed in a Configuration will always be marked “active” by default.

There may be situations in which you want these resources to be deployed to various nodes in your network but only marked active on certain target systems. In that case you can use an advanced feature called an “activation constraint”.

Activation constraints are analogous to the “provisioning constraints” that were discussed above, but rather than controlling which nodes the resources will be deployed to, they control which nodes upon which the resources will be made “active.” (Remember that these “constraints” can be thought of as “where clauses” which will be applied against the candidate Node objects.)

For example, suppose you have a rule in your configuration called “MyExperimentalRule” which has an “activation constraint” that looks like this:

```
{
  "ars_properties.status": "experimental"
}
```

This means the rule would be deployed to all systems whose “ars_properties” match the configuration’s provisioning constraint, but would only be active on those systems whose Nodes had a property “status” with the value “experimental”. In this way you could cause the rule to be inactive on all systems **except** the “experimental” ones. Once you were satisfied the rule was working you could remove its activation constraint and re-deploy, which would cause it to be activated everywhere it was provisioned. (A Source could be managed the same way.)

Setting Source Activation Constraints

To set the activation constraint on a source, you simply edit the Source’s definition; you will find an “Activation Constraint” field where it can be entered. (Remember that activation constraints must be well-formed JSON objects. Among other things this means the property names should be enclosed in double-quotes as shown above.)

Setting Rule Activation Constraints

Setting Rule activation constraints is a little different, since Rule definitions are completely defined by the VAIL code that you enter. You add an activation constraint to a Rule by adding a special clause to the first line, like this:

```
RULE MyExperimentalRule ACTIVATIONCONSTRAINT {"ars_properties.status": "experimental"}
// The rest of the Rule body...
...
...
```

Bootstrapping

In order to streamline the deployment of distributed networks of Vantiq nodes, there is a bootstrapping process for automatically configuring new nodes as they come online. There are two additional resources needed to support bootstrapping: **Node Templates** and **Node Configurations**. Node templates are node definitions that outline the configuration of a Vantiq Server that may be deployed many times, in similar but not quite identical ways. Node configurations are an exportable resource that can be placed in the configuration directory of a new Vantiq node (e.g., an edge server), to automatically connect the new node to another node and deploy any configurations specified on the parent node onto the new node. This is easiest to visualize in a parent-child relationship, where a node template defined on the parent installation would outline the definition of a child node.

To deploy a preconfigured child node that will automatically connect to its parent and provision itself with resources defined on the parent node, include a node configuration in the config directory (specified by the *io.vantiq.configDir* option, which defaults to the folder named *config*, parallel to *bin*) on the child node at initial startup time. Each individual child node would get a unique node configuration from the parent node, based on a node template defined on the parent node. This is necessary for security reasons, because the credentials used to make the initial communication between the child and parent nodes are one time use only. To get a Node Configuration for bootstrapping a child node, you would “fill out” a template on the parent node which creates a Node Configuration (a json file), download the configuration and drop it in the config directory on the child node prior to startup.

Example Scenario

There are 100 convenience stores spread out across the state, and the goal is to manage inventory across all of these locations through a centralized Vantiq system. In this scenario, there is a single parent node running in the cloud (i.e: api.vantiq.com). Each convenience store independently monitors its own inventory and the local conditions of the store, and reports status changes and aggregate metrics to the parent node. Each convenience store needs an edge device running the Vantiq server installed on premise to perform this local management, and these are the child nodes. The servers installed in the convenience stores all have a similar set of responsibilities that will be implemented by a standard collection of Rules, Types, Sources and other system resources that can be initially defined on the parent node, and then automatically created on each child node via the bootstrapping process. Here’s an outline of steps to perform this bootstrapping process.

1. Begin by identifying a collection of resources on the centralized parent node that should be provisioned onto a set of child nodes. These might include inventory types, rules that are triggered on every purchase, and sources for collecting data from sensors throughout the convenience store. Define these on the parent node, either in a Configuration object or as a Deployment Partition.
2. Create a Node Template on the parent node. The Node Template contains a “templateProperties” object with keys for each property that a child node must fill out in order to be uniquely identified, such as the owner of the particular convenience store and a location id. The Node Template “templateConfig” field will contain the list of “configurations” or “partitions” (defined on the parent node) that will be applied to each child node, the “originatingName” by which each child node will identify its parent node, and the type of each child node. In this case child is specified as the type, however peer can be specified to imply no hierarchical relationship between nodes.
3. Once the template is defined, “fill out” the template on the parent node to create a Node Configuration. Node configurations are unique to individual child nodes, so template would be filled out with different property values for each convenience store.
4. “Filling out” the template generates a Node Configuration record on the parent node which can be downloaded as a JSON file. By default the file should be named *bootstrap.json*.

5. The *bootstrap.json* file must then be added to the config directory (specified by the *io.vantiq.configDir* option at start time) on the child node. When the child node starts up, if a *bootstrap.json* file is found in the config directory the bootstrapping process will automatically commence, and the parent and child nodes will become connected and all resources defined in the Configurations and Partitions defined in the original template will be created within the child node. Once bootstrap is complete, the *bootstrap.json* file can be removed from the config directory.

To illustrate some of the configurable options that can be used when bootstrapping a child node, consider a machine where the *vantiq-server* launch script can be found in */opt/vantiq/bin*, and there is a config directory in */etc/vantiq/config*. The server will run the edge version of the Vantiq server as specified in the *startup.json* config file. The bootstrap file was renamed *mybootstrap.json* and placed in */etc/vantiq/config*. To start the server and have it bootstrap from the *mybootstrap.json* file, run the following command from */opt/vantiq/bin*:

```
./vantiq-server run io.vantiq.vertx.BootstrapVerticle \
  -conf startup.json \
  -Dio.vantiq.bootstrap.config=mybootstrap.json \
  -Dio.vantiq.configDir=/etc/vantiq/config
```

Once a node is successfully bootstrapped, remove the “*-Dio.vantiq.bootstrap.config=mybootstrap.json*” command the next time you restart the node.

Node Templates

A Node Template is an instance of the Node system resource, with a few special properties and values. Here’s an example of a template:

```
{
  "name": "convenienceStoreTemplate",
  "deliveryMethod": "bestEffort",
  "uri": "https://api.vantiq.com",
  "type": "template",
  "credentialType": "newtoken",
  "templateConfig": {
    "configurations": [ "RuleConfiguration1", "RuleConfiguration2"],
    "direction": "BOTH",
    "namespaceName": "convenienceStore",
    "originatingName": "ParentNode",
    "partitions": []
    "profiles": [
      "test1.admin__system",
      "test1.admin"
    ],
    "autoUpdate": "daily",
    "dailyAtHour": 0,
    "type": "peer",
    "organization": {
      "name": "ConvenienceStoreOrg",
      "quotaSettings": "unlimited"
    }
  }
  "ars_properties": {
  },
  "templateProperties":{
    "owner": null,
    "region": null,
    "phone": null,
    "locationId": null
  }
}
```

A few important things to note:

- The *credentialType* is set to “newtoken”, meaning each Node Configuration created from this template is assigned a unique long lived access token which is used to uniquely identify messages from a single child node. Alternatively, “token” can be specified and a token can be pre-assigned in the template for use by all child nodes based on the template.
- The *uri* property defines the address of the parent node.
- The *templateProperties* list is used to specify the list of properties that should be “filled out” to finish a node configuration that is unique to a single child node. In this case there may be many convenienceStore nodes, and so each node will need to specify an owner, region, phone number, and locationId to uniquely identify that node (for purposes of running commands with PROCESSED BY clauses).
- The *templateConfig* contains a set of properties to be used during bootstrapping:
 - “*namespaceName*”, which is the name of the namespace created to host node definition to its parent and resources from “configurations” and “partitions”.
 - “*organization*”, which is the organization hosting the namespace.
If the organization does not exist, it is created and the “*quotaSettings*” value is applied to the new organization. Values for “quotaSettings” can be “unlimited”, “default” or “custom”. If set to “custom”, another property “quota” must be specified with a JSON document defining the quota to be applied. If no organization is specified, a default organization with a “default” quota settings is created. The name of this default organization is derived from the name of the namespace. Specified quota settings are not applied to an existing organization.
 - “*configurations*”, which are a list of Configuration names that should be provisioned onto any child node constructed from this template. This is how rules, types, procedures, and other resources are automatically provisioned from a parent to a child node.
 - “*partitions*”, which is similar to “configurations”. It contains a list of partitions created by [Deployment Tool](#).

- “*autoUpdate*”, with possible values of “disabled” (default value), “hourly” or “daily”. It defines how frequently the child node will try to pull partitions from the source node. This feature applies to partitions only, not configurations.
- “*dailyAtHour*”, which is hour of the day to perform auto update. Only used when “*autoUpdate*” is set to “daily”.
- “*originatingName*”, which is the name the child will use to identify the node definition it uses to connect to its parent.
- “*direction*” with possible values of “BOTH” or “ONE”. “BOTH” means bootstrapping will create Nodes in both the parent namespace and bootstrapping namespace so they can communicate both ways. “ONE” means only bootstrapping namespace creates a Node definition to communicate with the parent. “ONE” is typically used in nodes running behind a firewall.
- “*profiles*” which are the profiles associated with the new generated access token to authorize all communication from the child node to the parent. It is used only if “*credentialType*” is set to “newtoken”.
- “*type*”, which is the type of each child node. This value will be child or peer, depending on if there is an implied hierarchy between the nodes.

The node template viewed in Vantiq IDE:

Type:

Template

Name:

convenienceStoreTemplate

Delivery Mode:

Best Effort

Name of 'parent' Node:

ParentNode

URI of 'parent' Node:

https://api.vantiq.com

Credential Type:

New Long-Lived Access Token

Profiles:

test1.admin__system, test1.admin

Reconnect:

Configurations to be deployed:

+ Add a Configuration

Configuration	Actions
<div>ruleConfiguration1</div>	<div>+</div> <div></div>
<div>ruleConfiguration2</div>	<div>+</div> <div></div>

Partitions to be Deployed:

+ Add a Deployment Partition

Partition	Actions
-----------	---------

Deploy Options:

Automatic Update:

Daily

at

0:00

Template Properties:

+ Add a Template Property

Name	Value	Actions
<div>owner</div>	<div>value</div>	<div>+</div> <div></div>
<div>region</div>	<div>value</div>	<div>+</div> <div></div>
<div>phone</div>	<div>value</div>	<div>+</div> <div></div>
<div>locationId</div>	<div>value</div>	<div>+</div> <div></div>

Properties (ars_properties):

+ Add a Property

Name	Value	Actions
------	-------	---------

Client Options:

Edit Client Options

Namespace Name:

convenienceStore

Organization Name:

ConvenienceStoreOrg

Organization Quota:

Unlimited

Quota is applied to new organization only.

Direction:

BOTH

Direction:

BOTH

Node Configurations

Node Configurations are a system resource which can be exported as a JSON file and loaded along with other configurations to bootstrap a new Vantiq node. The Node Configuration contains the basic information needed to connect the child node to its parent and create a unique identity by which the parent can identify the child. Here is an example of the contents of a Node Configuration:

```
{
  "name" : "newNode",
  "nodeConfiguration" : {
    "originatingUri" : "https://api.vantiq.com",
    "templateName" : " convenienceStoreTemplate ",
    "nodeProperties" : {
      "owner" : "John",
      "region" : "CA",
      "phone" : "555-555-5555"
    },
    "remoteUri" : "https://127.0.0.1:8080",
    "credentialType" : "token",
    "bootstrapToken" : "__one-time_token_generated_when_node_configuration_is_created__"
  }
}
```

A few important things to note:

- The *name* property is the name assigned to the bootstrapped node, and is the name the parent will use to identify the child node bootstrapped with this node configuration.
- The *originatingUri* in the nodeConfiguration object is the address of the parent node.
- The *templateName* is the name of Node Template
- The *nodeProperties* is a list of *templateProperties* defined in the node template (which are to be filled out for each node).
- The *remoteUri* is the address of the child node. This must be provided prior to creating the node configuration if the template specified direction “BOTH”. If a template with directory “ONE” is used, remoteUri can be empty.
- The *bootstrapToken* is generated when the node configuration is created. It will be deleted from the parent after the bootstrapping is successfully completed on the child node.

The Node configuration can be saved as a JSON file and placed in the config directory on the child node prior to the initial startup, and then on initial startup the child node will request all of the additional information necessary for bootstrapping from the parent node. The Node Configuration file should be named bootstrap.json by default, or specify an alternative name with the io.vantiq.bootstrap.config option at start time.

A Node Configuration is created by posting the template name and node configuration object (without credential information) to the Node Configuration resource like this:

```
POST https://dev.vantiq.com/api/v1/resources/nodeconfigs
{
  "templateName": "convenienceStoreTemplate",
  "node": {
    "name": "newNode",
    "uri": " http://127.0.0.1:8080",
    "ars_properties": {
      "owner": "John",
      "region": "CA",
      "phone": "555-555-5555"
    }
  }
}
```

Note that not all of the information that appears in the node configuration is required when making the request to create the node configuration. Other information is pulled from the template definition on the parent node, the parent node’s self definition, and the credential information is generated at the time the request is made.

The result of this post would be the creation of a Node Configuration resource on the parent node. When supplying that configuration at startup time to a child node, the bootstrap process would result in the following:

- A Node record is created on the Parent node named “newNode” which corresponds to the newly created child.
- An organization “ConvenienceStoreOrg” is created on the child node, if it does not already exist.
- A namespace named “convenienceStore” is created on the child node, within the organization “ConvenienceStoreOrg”.
- A Node record is created on the child node named “ParentNode” which corresponds to the parent node.
- All resources defined on the parent node in Configuration “RuleConfiguration1” and “RuleConfiguration2” would be created on the child node.
- The bootstrapToken that appears in the bootstrap.json will be deleted from parent node once bootstrapping is completed successfully.
- A new long lived access token would be generated on the Parent node and given to the child node to uniquely identify all communication from the child node to the parent.

The node configuration can also be created in Vantiq IDE.

Create Node Configuration 'newNode'

Name:

Template Name:

URI:

Properties:

Name	Value
owner	<input type="text" value="John"/>
region	<input type="text" value="CA"/>
phone	<input type="text" value="555-555-5555"/>
locationId	<input type="text" value="Property Value"/>

The content of bootstrap.json for the node configuration can be seen by clicking the action button.

The screenshot shows the 'Node Configurations' window. At the top, there's a search bar labeled 'Filter by Name or #tag'. Below it are buttons for 'Refresh', 'New', 'Delete All', and 'Delete'. A table lists configurations with columns: Name, Template, Parent URI, Child URI, and Actions. One configuration is listed: 'newNode' with template 'convenienceStoreTemplate', parent URI 'https://test.vantiq.com', and child URI 'http://127.0.0.1:8080'. An action menu icon is next to it.

A modal window displays the JSON for 'newNode':

```
{
  "name": "newNode",
  "nodeConfiguration": {
    "originatingUri": "https://test.vantiq.com",
    "templateName": "convenienceStoreTemplate",
    "nodeProperties": {
      "owner": "John",
      "region": "CA",
      "phone": "555-555-5555",
      "locationId": "12345"
    },
    "remoteUri": "http://127.0.0.1:8080",
    "credentialType": "token",
    "bootstrapToken": "[REDACTED]"
  }
}
```

Note that if a child node initial startup fails for any reason, all resources created during the bootstrap process are automatically cleaned up. For instance, if a new Organization and a Namespace are created during bootstrap, they will be deleted if the child node fails to start up. To disable this automatic cleanup, set the `cleanupOnFailure` property to `false` in the bootstrap.json file.

Self Node

If the template specifies a direction “BOTH” and the Node configuration specifies a *remoteURI* value with an `http` or `https` scheme, the client Node namespace self Node URI value is set to *remoteURI*, otherwise it is set to the system [self Node value](#).

Named WebSocket connection

To allow the parent Node to communicate with the child Node using a [named WebSocket connection](#), you must specify “BOTH” as the Node Template direction and define the parent Node address Uri with a WebSocket scheme (e.g., `"uri": "wss://api.vantiq.com"`). The named WebSocket is specific to each child Node and is specified with the Node Configuration `remoteUri` property using the `vqs` scheme. For example:

```
{
  "name" : "newNode",
  "nodeConfiguration" : {
    "originatingUri" : "wss://api.vantiq.com",
    "templateName" : " convenienceStoreTemplate ",
    "nodeProperties" : {
      "owner" : "John",
      "region" : "CA",
      "phone" : "555-555-5555"
    },
    "remoteUri" : "vqs://convenienceStore1",
    "credentialType" : "token",
    "bootstrapToken" : "__one-time_token_generated_when_node_configuration_is_created__"
  }
}
```

The Node “newNode” created on the parent will have a `vqs://convenienceStore1` Uri allowing the parent to communicate to the child Node. Note that, as mentioned in the [named WebSocket connection](#) section, the WebSocket connection between child to parent must be established first to allow the parent Node to communicate back to the child Node.

Deployment Tool

Deploying a distributed application to a set of nodes by hand-picking resources into different Configurations is not an easy task.

In a typical software development life cycle, developers would like to test the application in a test environment before deploying to a real production environment. Building Configurations to be shared by different environments is even more complicated.

The *Deployment Tool* is designed to simplify deployment tasks. The Deployment Tool creates Configurations automatically by analyzing Rules and Procedures, and presents a graphical view of how resources are deployed to nodes. It also allows developers to customize deployments so the same application can be easily adjusted for different environments.

The User’s Guide for the Deployment Tool may be found [here](#).