APPLICATION CONFIGURATION DATA SERVICE

Author: Rijn Buve – Reviewers: Charles Davies, Kees van Boxel, Kees Schuerman, Andreas Wuest, Sven Baselau

Version: 2016-10-11

Introduction

This document describes the requirements for and API of a generic service to read key-value pairs from a server, given a number of criteria. The key-value are referred to as "parameters" and generally represent application configuration data for the client.[1]

Typical use cases for this service are:

- Application or system configuration of clients
- Service discovery for clients

Typical operational contexts for this services are:

- Cases where many clients shared the same configuration but you wish to have the ability to specify special cases (e.g. for testing)
- Clients may wish to apply controls to reduce data usage (such as for over-the-air data plans) by using configurable service parameters (such as request frequency or scope)
- You wish to have controls to minimize the amount of duplication between mostly shared but slightly differing configurations
- High demand, high availability environments
- The ability to apply robustness controls, such as safe fallback configurations

The service is essentially defined as a read-only key-value store. It returns string values for keys by default, but it is perfectly fine for a client to provide its configuration as a binary "blob" (represented in the value of 1 key-value pair).

The service is generic and can potentially be used in many environments, such as environments for automotive clients (for CS), mobile apps, PNDs or even back-end web services.

The service is not specifically designed to be multi-tenanted (although it can be used as such). As a micro-service, it makes sense to deploy an instance of the service per context. It has been designed to be easily and highly scalable and to be able to run at low resource cost.

Functional Requirements

We define a "configuration" as a set of key/value-pairs which are context specific (i.e. the client knows how to interpret the key/values, not the service).

We recognize the following functional requirements for the parameter service:

[1] This service may be considered a "micro-service": a narrowly focused, independently deployable service.

1. The service offers a REST API to fetch a configuration based on client-provided hierarchical criteria.
2. The service provides a way to reconfigure the system.
3. The service offers the ability to provide string values for key-value pairs. The strings may represent a string, integer, or any other (possibly encoded) type (such as base64 encoded binary blobs).
4. The service offers controls to minimize the number of configurations that need to be stored for clients.
5. The service offers controls to minimize data usage by clients.
6. The service supports JSON or XML (or both) for the specification and retrieval of configurations.

Non-Functional Requirements

We recognize the following non-functional, or quality requirements for the parameter service:

1. The service needs to support linear horizontal scalability, both in performance and availability.
2. The service needs to support a sufficient workload capacity; P98[2] >= 50 concurrent requests.
3. The service needs to support a sufficient throughput; P98 <= 50 msec.
4. The service needs to offer high-availability; 99.9% per instance.
5. The service needs to "run cheaply"; must run on an AWS "t2.medium" instance or less (given a reasonable set of configurations).
6. The service is generic in a way that there no references to device-related, automotive-related or context-specific concepts.
7. The service (as a whole) does not require downtime when reconfiguring. (Individual nodes may go down, if the service is still able to operate well.)
8. The service requires no authentication/authorization (to be dealt with outside this service).

Service Description

The service can be visualized as operating on a search tree, which consists of nodes and leafs. The nodes themselves represent hierarchical search terms; the leafs are the search results, see Figure 1.

Given a set of hierarchical search terms, the service will try to match the search terms in the search tree and returns the leaf of the deepest node still matching the terms.

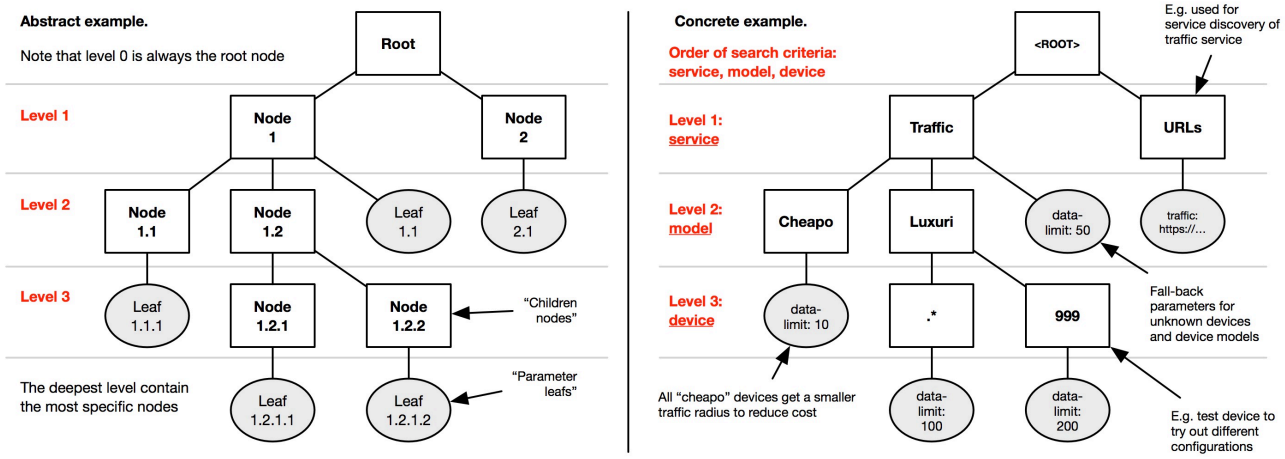

Figure 1. Search tree with nodes (search terms) and leafs (search results).

Example. Consider the example tree on the right in Figure 1.

This example represents a service configuration in which devices need to be able to get configuration parameters for a traffic service and to find the URL of the traffic service to connect to. The configuration, in this case, limits the amount of data a device is allowed to get and is called "**data-limit**". Every device knows its "model type", which in this case is "**cheapo**" or "**luxuri**" and has a unique device ID.

[2] P98 stands for the "98th percentile" of a population (only 2% of the population does not fall in this category).

The square boxes (nodes) in the tree represent the search terms that need to be matched and the circles (leafs) contain the search results.[3]

The service should produce the following search results, if a client provides its search criteria:

- service=**traffic,** model=**luxuri,** device=**123**, returns **radius:100**, returns leaf of the **.*** node.
- service=**traffic,** model=**luxuri,** device**l999**, returns **radius:200,** falls back to the leaf of the **999** node.
- service=**traffic,** model=**cheapo,** device=**789**, returns **radius:25**, falls back to the leaf of the **cheapo** node.
- service=**urls,** model=**lucury,** device=**123**, returns **traffic:https://...**, falls back to lead of **urls** node.

The search returns the leaf of the deepest node that can be matched using the search criteria. If no such node is found (and the root node does not contain a leaf), a NOT FOUND code is returned by the service (which in HTTP would be represented by a 404 status code).

Note that if the root node contains a leaf, this will always be the highest fallback node, which will be returned when no nodes match. This means that if the root node has a leaf, the search will never produce a NOT FOUND result.

The service allows the integrator to specify an initial search tree, which is read from a configuration file as well as an integration REST API to manage the search tree.

API Description

JSON or XML Configuration Files and Responses

The service accepts both JSON and XML as the input for configuration files and the output for responses. The default response format for the REST API is JSON (if the HTTP "**Accept:**" header is omitted or is "**Accept:application/json**").

To retrieve XML response, simply specify "**Accept:application/xml**". The element names in JSON and XML are the same for all the responses.

Note that this document uses JSON for most of its examples, but every configuration files or response is allowed to be XML or JSON.

Initializing the Search Tree

The service is configured by providing it a properties file which contains the URI of the configuration of the search tree. This URI is called the **startupConfigurationURI**.

Both the service properties file as well as the **startupConfigurationURI** must be available during start-up or the service will not boot (and produce a human understandable error entry in the log as to why it will not boot).

During start-up, the **startupConfigurationURI** is read, which may be either a URL (prefixed "http[s]:"), a file from a shared drive (prefixed "file:"), or a file on the classpath (prefixed "classpath:"). The format of the configuration file can be either JSON or XML. The service will figure out which one it is.

The contents reflect the entire search tree, which starts at the root node of the tree (examples are further down this document). If the URI was not specified, it could not be found, or its format was incorrect, an error entry will be put in the log and any subsequent call to retrieve information about the tree will return a HTTP FORBIDDEN (403) status code.

Monitoring hooks should represent this error state as well. (For example, if the service is to be deployed on Amazon Elastic Beanstalk with an HTTP "/status" monitoring call and initialization fails, the status call returns HTTP FORBIDDEN (403)).

[3] Note that although in this example only a single parameter is returned (radius=xyz), you would normally specify any number of parameters as a search result.

The service is expected to be run as a bunch of parallel nodes with the same configuration, connected to a load balancer. When a change to the search tree is required, you should edit the configuration file, or redirect the properties file to another configuration and restart all instances/nodes of the service, one-by-one (see also next paragraph).

Adding additional server nodes is as easy as attaching more nodes to the load balancer and booting them. For operation, nodes need no disk space, no database and no connection with other nodes.

To retrieve the configuration of a node, you can simply execute "GET /tree". The response matches the original configuration tree and can init fact be used as the input for another node.

Providing a New Configuration to Immutable Service Nodes

After a service node has read its configuration, it becomes immutable and cannot be altered in any way. The service provides no API to modify the configuration of nodes during runtime other than stopping nodes and restarting them with a different configuration which will be read during start-up.

This approach might seem inflexible, but it's extremely reliable and robust and removes the need for intra-node communication, shared databases and such. All you need to do to reconfigure a running system, is provide a new configuration file and restart each node, one by one, which will cause zero downtime for the end-user. (And a service like Amazon Elastic Beanstalk will do this automatically for you.)

Querying the Search Tree

This is the method clients would normally use to fetch a configuration. It is used to search the tree for the best matching search result. The returned result may either be a match for all input search terms, or be the best "fallback" configuration if one or more search terms cannot be matched.

Its format is:

```
GET /tree? {level1}={value1} & {level2}={value2} & …
```

(Spaces added for clarity only). The level names should match the levels as described in the "**levels**" attribute of the top node of the search tree. The search values are matched against the node names.

If a level name is omitted from the query (or forgotten), it is assumed to be empty. You can match empty names using the node name "**.***" (or another regular expression which matches the empty string).

The result of a search query contains the parameters of the leaf of the deepest node matching the search criteria as well as the path of the node to which the leaf is attached:

```
{"parameters": [{"key": "{key1}", "value": "{value1}"}, {…}, …],
"matched": "…"}
```

If no search result was found, HTTP status code 404 (NOT FOUND) is returned instead.

The "**matched**" attribute is particularly useful to understand whether fallback parameters were used and if so, from which node.

For example, if a query was executed for **"x=1&y=2&z=3",** but the configuration did not specify a value for a node at level **z** with value **3**, then the returned "**matched**" string could indicate the actual node it returned matched "**x=1&y=2**" or "**x=1"**, rather than "**x=1&y=2&z=3**". Note that if no match was found and the parameters (leaf) of the root node were returned, the value of "**matched**" is the empty string "".

This fallback mechanism is unambiguous and deterministic due to the hierarchical nature of the search tree configuration. This means fallbacks should never need to cause unexpected surprises.

Example. Consider the example tree on the right in Figure 1 again.

Using the search method, you would get these results:

```
GET /tree?service=urls
        matched=service=urls
        traffic:https://… (exact match)
```

```
GET /tree?service=urls&model=luxuri&deviceID=123
        matched=service=urls
        traffic:https://… (ignoring model/deviceID, fallback to urls)
GET /tree?service=traffic
        matched=service=traffic
        data-limit:50      (exact match)
GET /tree?service=traffic&model=luxuri&deviceID=123
        matched=service=traffic&model=luxuri&device=.*
        data-limit:100     (exact match with .* node)
GET /tree?service=traffic&model=luxuri&deviceID=456
        matched=service=traffic&model=luxuri
        radius:50          (456 not found, cannot match .* node, fallback to luxuri)
GET /tree?service=traffic&model=cheapo&deviceID=789
        matched=service=traffic&model=cheapo
        radius:10          (789 not found, fallback to cheapo)
GET /tree?service=other
        NOT FOUND          (other not found, no fallback found)
```

Getting Multiple Configurations at Once

It is possible to query the search tree for multiple results at once, reducing the number of calls a client needs to make to the service. To execute multiple searches in a single call, separate the search terms like this:

```
GET /tree? {level1}={valueX},{valueY} & {level2}={value2},… & …
```

Terms must be separated by comma's. If you wish to provide an empty string, you should only provide the comma and not search term. If you wish to use the same search term in subsequent searches, simply omit the comma (the last term will be re-used).

The following query will execute 2 search in a single call, for [traffic,luxury,123] and for [sys,luxury,123]:

```
GET /tree? service=traffic,sys & model=luxuri & device=123
```

As you can see the search terms for **model** and **device** are re-used in the second search. If you don't want that, or wish to use other search terms, you can provide other values as well. For example, to search for these terms [traffic,luxury,123] and for [sys, ,test], you specify:

```
GET /tree? service=traffic,sys & model=luxury, & device=123,test
```

The returned response of a multi-search call is a JSON array of results, rather than a single result. The search results in the array are always in the same order as the input search query elements.

Note that if **any** of the sub-queries are not found, the entire query responds with 404 (NOT FOUND), even if some of the searches did not fail.

Getting Individual Nodes from the Search Tree

The service exposes REST methods to query the configuration of the search tree. (Don't confuse the querying the configuration of the search tree with searching the search tree for results.)

```
GET /tree[/{level1}[/{level2[…]]]  -- get specific tree nodes
```

This produces a response like:

```
{"nodes": ["{node1}", "{node2}", …],
"parameters": [{"key": "{key1}"}, {"value": "{value1}"}, {…}, …]}
```

The "nodes" array is optional and lists the children nodes with search terms one level below the specified node. The "parameters" value is the optional leaf node of this node and lists the search result (an array of key-value pairs).

Important: Normally, you would not use this method to get a configuration for a device, as it requires you to *exactly* specify a node in a hierarchy and does not evaluate any regular expression. (You would

typically use this only in a program that wants to read the actual nodes, for example, for display purposes).

Normally, you would use the "**search**" method below, because it provides fallback configurations if certain elements of the search path cannot be matched. The method to fetch individual nodes is usually only useful for the root node, to fetch the entire configuration from a node.

Example. Consider Figure 1 again. Using the method to query the individual tree nodes, you would get these results (remember you normally use the "search" method instead!):

```
GET /                       => produces help text for service
GET /tree                   => { levels=service/model/deviceID, nodes={…} }
GET /tree/urls              => { parameters: [ traffic:https://…  ] }
GET /tree/urls/luxuri/123   => NOT FOUND
GET /tree/traffic           => { nodes: { …cheapo and luxury… }, parameters: [ data-limit:50 ] }
GET /tree/traffic/luxuri/.* => { parameters: [ radius:100 ] }
GET /tree/traffic/luxuri/123 => NOT FOUND
GET /tree/traffic/luxuri/999 => { parameters: [ radius:200 ] }
```

HTTP Status Codes

The HTTP status codes for the service are defined as:

- Successful GET: 200 – OK
- Node not found for GET: 404 – NOT FOUND
- No search result found for GET: 404 – NOT FOUND

Specifying the Configuration File

The configuration file is read from the "`ApplicationConfigurationData.startupConfigurationURI`" property in the properties file of the service. Below are some examples fo specifying configuration files for nodes to read during start-up. We also provide 2 examples of the configuration file to produce the configuration tree of Figure 1 (one as JSON and one as XML).

Examples:

Specify a URL location for an initial configuration (assuming a properties file):

```
ApplicationConfigurationData.startupConfigurationURI = http://datacenter/myconf.json
```

Specify a configuration file on a shared drive:

```
ApplicationConfigurationData.startupConfigurationURI = file://share-1/myconf.xml
```

Specify a configuration file on the classpath:

```
ApplicationConfigurationData.startupConfigurationURI = classpath://myconf.xml
```

Configuration file for the search tree in Figure 1 in JSON format.

```
{
  "levels": ["service", "model", "deviceID"],
  "nodes": [
    {
      "match": "traffic",
      "nodes": [
        {
          "match": "cheapo",
          "parameters": [{"key": "data-limit", "value": "10"}]
        }, {
          "match": "luxuri",
          "nodes": [
            {
              "match": ".*",
              "parameters": [{"key": "radius", "value": "100"}]
            }, {
              "match": "999",
```

```
              "parameters": [{"key": "radius", "value": "200"}]
            }
          ]
        }
      ],
      "parameters": [{"key": "data-limit", "value": "50"}]
    }, {
      "match": "urls",
      "parameters": [{"key": "traffic", "value": "https://…"}]
    }
  ],
  "modified": "2016-04-05T17:28:16Z"
}
```

Configuration file for the search tree in Figure 1 in XML format.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tree>
    <levels>
        <level>service</level>
        <level>model</level>
        <level>deviceID</level>
    </levels>
    <nodes>
        <node>
            <match>traffic</match>
            <nodes>
                <node>
                    <match>cheapo</match>
                    <parameters>
                        <parameter>
                            <key>data-limit</key>
                            <value>10</value>
                        </parameter>
                    </parameters>
                </node>
                <node>
                    <match>luxuri</match>
                    <nodes>
                        <node>
                            <match>.*</match>
                            <parameters>
                                <parameter>
                                    <key>data-limit</key>
                                    <value>100</value>
                                </parameter>
                            </parameters>
                        </node>
                        <node>
                            <match>999</match>
                            <parameters>
                                <parameter>
                                    <key>data-limit</key>
                                    <value>200</value>
                                </parameter>
                            </parameters>
                        </node>
                    </nodes>
                </node>
            </nodes>
            <parameters>
                <parameter>
                    <key>data-limit</key>
                    <value>50</value>
                </parameter>
            </parameters>
        </node>
        <node>
            <match>urls</match>
            <parameters>
                <parameter>
                    <key>traffic</key>
                    <value>https://…</value>
                </parameter>
            </parameters>
        </node>
    </nodes>
    <modified>2016-04-05T17:28:16Z</modified>
</tree>
```

Using Regex's to Match Ranges of Criteria

The service allows you to specify regex's in the configuration "**match**" values. In that case, specified names in a search query are matched against these regex's. This allows you to specify configuration for multiple search paths (read: devices) in a simple way.

For example, if the configuration contains a node:

```
"match": "Device100[0-9]+","nodes": [ … ], …
```

Then, all of the queries below will be served the same parameters:

```
GET /tree?device=Device1002
GET /tree?device=-Device10072
GET /tree?device=Device100846
```

This makes it much easier to specify exceptional configurations, for example, for test devices.

Note 1: Exact, literal string matches always prevail over regex matches. So, if a search string "**Joe**" is provided, a match which node name "**Joe**" always prevails over any node with a regular expression (including ".*").

Note 2: The regular expression ".*" matches amny string, including the "empty string" (which is used when a search term is empty or not provided in the query).

Using HTTP Header If-Modified-Since

To minimize data usage, specifically over-the-air, the service allows the use of the standard HTTP header "If-Modified-Since". The supplied value is matched to the "modified" date of the matched node, or if that node has no "modified" attribute, of its nearest parent root.

The service will return HTTP status code 304 (NOT MODIFIED) if the found modified date is not newer than the supplied date/time. (If no modified attribute was specified in the configuration tree, a full response is always returned.)

The modified date/time format in the configuration tree conforms to ISO-8601 :

```
"modified": "1994-11-06T07:49:37Z"
```

Note that the format of the "If-Modified-Since" HTTP header is defined by W3C, RFC 1123, as:

```
If-Modified-Since: Sun, 06 Nov 1994 08:49:37 GMT
```

Failure to provide the correct format for the HTTP header (or accidentally use the ISO format in the HTTP header) results in ignoring the header, which means a full result is always returned.

Every response from the server contains the "Last-Modified" HTTP header to indicate the actual last modification time. This header looks like this (also for 304 responses):

```
Last-Modified: Sun, 06 Nov 1994 08:49:37 GMT
```

Using HTTP Header If-None-match

Another way to minimize data usage is using ETags and the HTTP header "If-None-Match". Every response produced by the service contains an ETag in the HTTP header, which looks like this (also for 304 responses):

```
ETag: "51cba67887b54ccaefbba417dab6b9f64ba2d765"
```

The ETag is essentially a form of hashcode for the result, which means that if 2 response are the same, they have the same ETag (and if the ETag differs, they would be different).

This value may be passed in a next request in the HTTP header "If-None-Match", like this:

```
If-None-Match: "51cba67887b54ccaefbba417dab6b9f64ba2d765"
```

If the response to be returned would be the same as the response that generated the ETag supplied in the If-None-Match value, HTTP status code 304 (NOT MODIFIED) is returned, without a body. Otherwise a full response (with a new ETag) is returned. The caller should use the new ETag in subsequent calls.

Important: Note the quotes around the values. They are important. The ETag value *must* be enclosed in quotes according to the W3C standard. Failure to do so treats the ETag value as non-matching (always returning a full response body).

Also important: Which method you should choose, If-Modified-Since or If-None-Match, is up to you. Note that if you supply an ETag in If-None-Match, it prevails over the If-Modified-Since value (and there is little valie in using the "modified" value).

Sharing Configuration Subtrees with Includes

It is possible to share subtrees of the configuration tree for different configurations. For example, if a specific subtree resembles the settings for a service and these settings are shared for many device configuration, you may wish to specify these settings once in a separate configuration URI and simply include that URI anywhere in the configuration tree.

This looks like this:

```
{
  "nodes": [
    { "include" : "file://shared/sys_config.json" },
    { "include" : "file://shared/service1_config.json" },
    { "include" : "file://shared/service2_config.json" }
  ],
  "modified" : "2016-04-05T17:28:16Z"
}
```

The include file `"sys_config.json"` would live on the class path in this case and might look like this:

```
{
  "match": "SYS",
  "parameters": [{"key": "demo", "value": "false"}, {"key": "sound", "value": "off"}]
}
```

Include files can live on the classpath (classpath:), on a shared drive (file:) or web-service (http:/https:). Nodes names from peer nodes must be unique, also when include files are used, or the service won't start.

Include files can be nested to any level, but cannot be included recursively (as the recursion would not end). This allows you, for example, to create a:

config file per high-level service, which includes
config files per device configuration, which include
config files for individual devices

to create a hierarchy of config files, which may be easier to maintain than 1 enormous config file.

Note that only the "modified" time stamp of the root node is considered for the HTTP If-Modified-Since optimization discussed earlier.

Matching the Requirements

The service as defined in this document has been implemented at

https://bitbucket.tomtomgroup.com/projects/ST/repos/application-configuration-data-service/browse

It matches the requirements listed as follows:

- Reconfiguring the system is done by providing a new configuration setup and rebooting the nodes one-by-one. This will cause zero downtime of the system as whole and is extremely reliable as a mechanism.

- The number of configurations can be kept low by using the "include" mechanism wisely and providing smart fallback configurations, or using regular expressions in the configuration tree to have one configuration apply to many clients.
- The service is stateless, which makes it trivial to up- or downscale the nodes horizontally.
- The service does not require a database or other middleware, which makes it extremely simple to deploy, monitor and operate. This also reduces operational costs to a minimum (requiring the smallest of virtual machines).
- Everything is served from memory, which produces extremely fast responses.

Change Log

2015-08-05 Initial version, extracted from document "Perseus Integration for CS". Corrected key-value type to array type.
2015-08-07 Removed reference to JSON, improved HTTP status code results.
2015-08-10 Added the ability to read an initial configuration setup from a (properties specified) URI. Also added management interface for changing parameters of a node only (POST).
2015-08-12 Checked final comments.
2015-09-17 Removed remaining comments.
2016-04-05 Removed all write/update methods. The service is now defined to be read-only. Added If-Modified-Since.
2016-04-11 Added include functionality.
2016-07-25 Updated document to include full XML support and corrected examples.
2016-09-22 Major change to API: GET no longer hierarchically structured, only the configuration tree is.
2016-09-26 Renamed parameter 'order' to 'levels' for consistency.
2016-10-10 Provided simplified syntax for single result calls and improved examples.
2016-10-11 Dropped levels/search-syntax and renamed node 'name' to 'match'. Updated examples. Clarified text.