About RESTful API

Overview

This page reviews the RESTful API for GroundWork Monitor.

CONTENTS

RELATED RESOURCES

- Developer Reference

WAS THIS PAGE HELPFUL?

- Leave Feedback

1.0 Introduction

REST (**RE**presentional **S**tate **T**ransfer) is a distributed computing style based on the architectural successes of the web. The RESTful Services for GroundWork Monitor are a set of distributed web service APIs adhering to the design principles of REST. In REST terminology, a web service API is known as a resource.

The foundation web services are based on the following REST principals:

- Each resource (web service) is addressable by a URI
- Resources are manipulated via their representations (e.g. XML, JSON)
- A uniform interface is exposed for all resources based on HTTP methods (GET, PUT, POST, DELETE)
- Messages are self-descriptive - i.e. each message contains everything necessary for a stateless server to process it

By applying web and HTTP principles to web services, web services will gain some of the desirable properties of the web such as horizontal scalability, decoupled clients and servers, reduced latency through the use of proxies and real interoperability.

Another very real feature of RESTful APIs are their simplicity and familiarity of use. With any RESTful API, it should be relatively trivial to point a simple client (e.g. curl in a shell script) at a URI for the API and quickly build something useful. This makes a refreshing change from heavyweight middleware frameworks such as CORBA or WS-*.

The following convention applies to all REST services for the foundation unless otherwise noted on the separate service.

- Follow REST principles described above
- Provide easy to use and consume APIs based on the Foundation object model. For example, Host related services all start with `/api/hosts`, or Device services start with `/api/devices`
- Input and Output supports both XML and JSON for all APIs
- Updates and inserts are performed with POST or PUT methods, and will support batch processing. Batch processing is not all or none. Some objects may succeed, while others may fail. The service will always return the status of each individual operation.
- REST APIs will be optionally (configurable) secure
- A simple query language will be supported using a HTTP query parameter named ‘query’

1.1 Rest Service API Conventions

- PUT vs. POST - POST methods are used to create new objects or invoke non-idempotent methods on an object. PUT methods are used to update objects in place.
- Content-Type HTTP request header - Specifies `application/json` or `application/xml` MIME types for POST and PUT request content.
- Accept HTTP request header - Specifies `application/json` or `application/xml` MIME types for response content. Defaults to `application/xml` if not supplied.
- JSON strings map notation - Response content example: { "KEY": "VALUE", ... }
- JSON strings collection notation - Response content example: ["VALUE", ...]
- XML strings map notation - Response content example: `<map><entry><key>KEY</key><value>VALUE</value></entry>...</map>`
- XML strings collection notation - Response content example: `<list><value>VALUE</value>...</list>`

1.2 HTTP Methods

Only standard HTTP methods are supported:

- GET - for querying and retrieving individual resources or lists of resources

- POST - for creating resources (idempotent)
- PUT for updating an existing resource
- DELETE for deleting resources

1.3 Conditional GET

Not supported in the initial release. Support can easily be added, but may require changes in the Collage database to support last modified date on all resources. If supported the REST API would return a 304 code for resource requests that have not changed with an empty body.

1.4 Supported Formats

- XML
- JSON

1.5 Creating Objects

Objects are created using the HTTP Post Method. Parameters should be supplied in the request body as a JSON or XML. Successfully creating or updating a resource will result in a 200 HTTP status code.

All XML post data will follow the general pattern of a plural collection name, and repeated individual singular elements, for example:

```
<hosts>
  <host hostName="host-100" description="First of my servers"
```

And for JSON:

```
{ "hosts":
 [
  {
  "hostName"
```

Creating Objects Asynchronously

Some APIs support the creation of objects asynchronously with the HTTP Post Method. The default is to create them synchronously. To override this behavior and create the objects asynchronously, pass in the *async* request parameter:

async=true

APIs that Support Asynchronous Mode

- Hosts API
- Services API

A synchronous (default) request will not return back to your client code until the completion of processing all objects provided in the post data. Whereas an asynchronous request will submit the work to a queue, and return immediately.

Successfully submitting the request will result in a 200 HTTP status code. Failure to submit to the request will return a 500 HTTP Status Code.

1.6 Deleting Objects

Objects are deleted using the HTTP Delete Method. Parameters should be supplied in the request body as a JSON or XML. Successfully deleting a resource will result in a 200 HTTP status code.

All XML post data for deletions will follow the same pattern as for creates, with the exception that only the objects primary key(s) are required: all other fields are ignored.

```
<hosts>
  <host hostName="host-100" description="First of my servers"
...
```

And for JSON:

```
{ "hosts":
 [
   {
    "hostName": "host-100”,
    "description”: "First of my servers"
    }
...
```

Objects can also be deleted by passing comma-separated (and encoded) primary keys on the path:

```
curl -X DELETE -H "Accept: application/xml" --header "Content-Type: application/xml"
http://localhost/api/hosts/host-1,host-2,host-3
```

1.7 Error Handling

Any request may result in one of the following error codes:

- 400 Bad Request - This will be returned if the URI is invalid. For example /api/hosts/123 is valid while /api/hosts/123/badly_formed_url will return 400.
- 401 Unauthorized - returned by any request if authentication is required and missing or wrong.
- 404 Not Found - Only used when GET request is valid, but resource is not found (e.g. request for Host id that does not exist). See the section HTTP Responses below for special handling POSTs and DELETEs, when a resource is not found.
- 415 Unsupported Media Type - can be returned for some POST and PUT endpoints that expect UTF-8 content.
- 422 Unprocessable Entity - Only used when creating or updating a resource and the content of the request is malformed or invalid in some way that prevents the resource from being created or updated. For example, if performing an update on a resource, but not providing the JSON required.
- 500 Internal Server Error - This should only be used when handling exceptions on the server.

When error codes are sent, XML, JSON, or HTML content is also sent to describe the error. The specific representation sent is based on the content type accepted by the request, (i.e. the Accept header). Here are examples of XML and JSON content:

JSON Bad Authorization Token, (401 Unauthorized)

```
{
  "error" : "Unauthorized",
  "status" : 401
}
```

XML Host Not Found, (404 Not Found)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error error="Host name [not-a-host] was not found" status="404"/>
```

XML Invalid API Endpoint, (404 Not Found)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error error="Could not find resource for relative : /not-an-api of full path: http:
//localhost:8080/foundation-webapp/api/not-an-api" status="404"/>
```

JSON Invalid Hosts Query String, (500 Internal Server Error)

```
{
  "error" : "An error occurred processing request for hosts.",
  "status" : 500
}
```

HTML content is sent when the URL does not resolve to any REST API service or the endpoint is invoked from the browser where any content

type is accepted.

1.8 Date Formats

All dates returned by the Rest API will be in ISO-8601 Extended Combined Date and Time format:

YYYY-MM-DD'T'hh:mm:ss.SSSZZZZZZ

Where:

YYYY = Year
MM = Month
DD = Day
HH = hour
mm = minute
ss = second
SSS = millisecond
ZZZZZZ = Timezone in format of 6 required characters: ±HH:mm
± either plus (+) or minus (-) to represent the UTC offset from UTC (Coordinated Universal Time (GMT))
HH (hour) mm(minute) format

GMT time would be +00:00 (zero hour, zero minute offset from UTC; the 'Zulu' form will not be returned)

Examples:
2014-04-01T22:09:35.001-07:00
2014-04-01T22:09:35.001+06:30
2014-04-02T05:09:35.001+00:00

See http://en.wikipedia.org/wiki/ISO_8601#UTC

1.9 POST Date Formats

Dates provided in POST data must also be formatted in one of the following formats:

- ISO-8601 UTC full format
  - Example: 2014-04-03T16:25:25.012-06:00
- ISO-8601 UTC + RFC-822 full format
  - Example: 2014-04-03T16:25:25.012-0600
- ISO-8601 'Zulu' UTC short format
  - Example: 2014-04-03T22:25:25.021Z
- The number of milliseconds that have elapsed since midnight, January 1, 1970, UTC
  - Example: 1396563925021
    For all ISO-8601 formats, milliseconds are optional but recommended for accurate monitoring data

1.10 Mapping Depth

Some APIs allow for shallow, deep, or other representations of entities. Shallow representation retrieves only the primitive data attributes (of an entity (i.e. Host, Device), and the properties of that entity. Deep representations will take longer to retrieve, since objects need to joined from the database to bring together all associated collections and 1:1 associations.

shallow	primitive data attributes, 1:1 association names, properties (default)
deep	shallow, plus: associated collections (shallow), 1:1 associations shallow
simple	Same as shallow except there are no properties (only supported for Hosts)
full	recursive form of deep where collections and associations are also retrieved deep, (only supported for Categories)
sync	special depth used when synchronizing entities with external systems, (only supported for Hosts and Services)

Mapping depth is provided as a query parameter named "depth". Valid values are "shallow", "simple", "deep", "full", and "sync". Not all entities support the depth parameter. The following entities do support depth:

1. Category
2. Device
3. Host
4. HostGroup

Not Supported

1. Event
2. Graph

3. Service
4. Statistics

If a depth query parameter is not provided, the default is Shallow.

1.11 HTTP Responses

The HTTP response depends on the type of operation. There are three general types of responses:

1. GET single entity - returns a single entity in JSON or XML
2. GET collection - used on queries, returns a collection of entities in JSON or XML
3. POST, PUT, DELETE - returns a ResultSet, containing a list of the status of all operations

Get Operations

When looking up by primary key, GET operations return a single entity. All other GET operations like queries, will return a collection of one or more entities, wrapped by a container element. GET operation responses are dependent on the HTTP code. Your code should always check the HTTP status code prior to looking at the HTTP response:

- 200 - a successful request. The HTTP response body contains a single entity or collection of entities
- 404 - not found. There will be no response body in this case
- 401 - unauthorized. There will be no response body in this case
- 500 - server error. Usually a simple text message. But in severe error cases, the App server will return HTML
- other - bad parameters. Usually a simple text message. But in severe error cases, the App server will return HTML

Examples of GETs

A primary key GET will return a single `<host>` element:

```
<host id="1" hostName="localhost" description="localhost" monitorStatus="UP" />
```

Queries return collections of entities for example:

```
<hosts>
  <host id="1" hostName="localhost" description="localhost" monitorStatus="UP" />
  <host id="2" hostName="localhost2" description="l2" monitorStatus="UP" />
</hosts>
```

POST, PUT, GET Operations

The HTTP response for POST, PUT and DELETE operations return a `<results>` collection of one or more `<result>` elements. For POST, PUT and DELETE operations, the server tries to return 200, with detailed response codes for each individual operation. Thus these kind of CRUD operations are not transactional. For example, if your application posts 3 host entities, 2 might succeed, and one update might fail.

Your code should always check the HTTP status code prior to looking at the HTTP response:

- 200 - a successful request. The HTTP response body contains a result set of multiple operations
- 401 - unauthorized. There will be no response body in this case
- 500 - server error. Usually a simple text message. But in severe error cases, the App server will return HTML
- other - bad parameters. Usually a simple text message. But in severe error cases, the App server will return HTML

Example Response

```
<results count='3' success='2' failure='1' entityType='Host' operation='Update'>
  <result status='failure' message='Bad hostName provided' entity='host""33' />
  <result status='success' entity='host34'
       location='http://localhost/monitor/api/hosts/host34' />
  <result status='success' entity='host36'
       location='http://localhost/monitor/api/hosts/host36' />
</results>
```

The **results** element has the following attributes:

1. **count** - the number of entities processed
2. **success** - the count of successful operations
3. **failure** - the count of unsuccessful operations
4. **entityType** - the entity type being processed

5. **operation** - for POST its "Update", DELETE its "Delete"

The **result** element has several attributes:

1. **status** - the result of the operation on an individual entity. Either ‘success’ or ‘failure’
2. **message** - a message describing details of operation on this entity.
    - Normally, messages are only provided on failure
3. **entity** - the primary key or primary name of an entity
4. **location** - a usable URL pointing to the Restful API entry point for retrieving a distinct entity.
    - Normally only provided upon successful operations

2.0 Searching and Queries

Queries are specified as the WHERE clause of a query only. The column names in queries must map to Collage entity model names, not database columns. The actual names valid in queries are included in this document. Valid names are specific to which entity you are querying i.e. Host, Event, Service... .

A simple query
hostName = ‘localhost’

Queries can be anded or or-ed:
(device = ‘localhost’ and severity = ‘FATAL’)

Parenthesis are supported to group expressions:
((device = ‘localhost’ and severity = ‘FATAL’) and (msgCount > 3))

Standards operators supported =, <>, >, <, like, between, in
property.ExecutionTime between 500 and 1000
(appType = 'NAGIOS' and device like '172.28.113%')";

Properties are prefixed by the prefix "property.":
(property.ExecutionTime < 10 and property.Latency between 800 and 900)

Queries can be sorted using the order by clause. Queries can be sorted on properties:
(property.ExecutionTime < 10 and property.Latency between 800 and 900) order by property. Latency

Date Range queries are supported:
property.LastStateChange > '2013-05-17 09:33:00'
(property.LastStateChange between '2013-05-20' and '2013-05-22') and monitorStatus = 'UP'

Functions are supported in queries:
day(property.LastStateChange) = 18

Not null checks syntax:
(appType is not null)

In Queries are supported:
host in ('qa-load-xp-1','qa-sles-11-64','do-win7-1')

2.1 Limitations to Queries

1. Only WHERE and SORT BY clauses supported
2. Expression literals and identifiers must be space delimited

```
VALID     : (hostName = 'localhost')
  NOT VALID : (hostName='localhost')
```

   However, IN clauses must not have spaces

```
VALID     : host in ('1','2','3')
  NOT VALID : host in ('1', '2', '3')
```

3. Joins are limited to only modeled associations
4. Queries are implemented as a single HTTP query parameter and thus must be HTTP encoded.

Supported Functions

Date Functions	day, month, year, hour, minute, current_date

Functions Not Supported

Aggregate Functions	sum, count, avg, min, max

3.0 Paging

By default, all queries will return the entire result set. There are two request parameters supported by all queryable REST APIs:

first	Query	Paging. First record to start from
count	Query	Paging. Number of records to include when paging

These two parameters are optional. If they are not specified, no paging will occur and the entire result set will be return.

Using the paging parameters requires the client application to track the page counts. If your application wanted to present pages the results of a query in pageable format, the API call should specify a "first" parameter and a “count” parameter. These two parameters define the window of paging over a query.

A first request should set first=1, and count should be set to the number of records returned. For example, here is a first request, asking for 20 items:
GET /api/events?query=service='http_alive'&first=1&count=20

A second request would retrieve items 21-40
GET /api/events?query=service='http_alive'&first=21&count=20

And so on...
GET /api/events?query=service='http_alive'&first=41&count=20

4.0 Dynamic Properties

The GroundWork REST API has a set of APIs that closely mirror the entities in the GroundWork database. An entity is simply a class of objects in GroundWork's data model. In the database, they are represented as tables. All entities in the GroundWork database have a base set of standard attributes. These attributes can be retrieved, queried, and stored onto entities at run time. When querying the REST API, you will always get back an object's base attributes.

Additionally, some of the entity types have dynamic properties. These dynamic properties can be user-defined. By default, an entity will not have any dynamic properties. However applications, including internal GroundWork applications, can add dynamic properties to entities at run time. Through the Rest API, dynamic properties can be easily created with the PropertyType API, and then attached to entities that support dynamic properties. When retrieving entities that support dynamic properties, these properties are automatically returned in a properties map.

The following APIs support Dynamic Properties:

1. Host
2. ApplicationType
3. Event
4. Service

Out of the box, there are a set of Dynamic Property Types:

Property Type	Description	Type
30DayMovingAvg		Boolean
AcknowledgeComment		String
AcknowledgedBy		String
Alias	Host Alias information	String
ApplicationCode		String
ApplicationName		String
CactiRRDCommand	Cacti RRD Command	String
Category	Category of snmp device	String
Comments	Host or Service Comments in XML format	String

ContactNumber	Last output received	String
ContactPerson	Last output received	String
CurrentAttempt	Current attempt running check	Long
CurrentNotificationNumber	The count of notifications	Integer
DeactivationTime	The time when the host was deactivated	String
ErrorType		String
Event_Name	Event_Name	String
Event_OID_numeric	Event_OID_numeric	String
Event_OID_symbolic	Event_OID_symbolic of snmp device	String
ExecutionTime		Double
ExtendedInfo		String
LastNotificationTime	The time of the last notification	Date
LastPluginOutput	Last output received	String
LastStateChange	The time of the last change of state	Date
Latency		Double
Location	Last output received	String
LoggerName		String
MaxAttempts	Max attempts configured	Long
Notes	Configuration Notes field	String
Parent	List of parent hosts separated by commas	String
PercentStateChange		Double
PerformanceData	The last Nagios performance data	String
RRDCommand	Custom RRD command	String
RRDLabel	Label for Graph	String
RRDPath	fully qualified path to RRD image	String
RemoteRRDCommand	Remote RRD Command	String
RetryNumber	The number of times an attempt has been made to contact the service	Integer
ScheduledDowntimeDepth		Integer
ServiceDependencies		String
SubComponent		String
TimeCritical	The amount of time that the entity has had a status of CRITICAL	Long
TimeDown	The amount of time that the host has been DOWN	Long
TimeOK	The amount of time that the entity has had a status of OK	Long
TimeUnknown	The amount of time that the entity has had a status of UNKNOWN	Long
TimeUnreachable	The amount of time that the host has been UNREACHABLE	Long
TimeUp	The amount of time that the host has been UP	Long
TimeWarning	The amount of time that the entity has had a status of WARNING	Long
UpdatedBy	UpdatedBy	String
Variable_Bindings		String

ipaddress	ip Address of snmp device	String
isAcceptPassiveChecks		Boolean
isAcknowledged	Has the current state been acknowledged?	Boolean
isChecksEnabled		Boolean
isEventHandlersEnabled		Boolean
isFailurePredictionEnabled		Boolean
isFlapDetectionEnabled		Boolean
isHostFlapping		Boolean
isNotificationsEnabled		Boolean
isObsessOverHost		Integer
isObsessOverService		Boolean
isPassiveChecksEnabled	Nagios 2.0	Boolean
isProblemAcknowledged		Boolean
isProcessPerformanceData		Boolean
isServiceFlapping		Boolean

Queries on dynamic properties are prefixed by "property." For example:
GET /api/services?query=property.LastPluginOutput like 'OK%'

In responses, properties are included in a property map. For example:
XML:

```
<properties>
  <property name="MaxAttempts" value="10"/>
  <property name="ExecutionTime" value="6"/>
</properties>
```

JSON:

```
"properties": {
     "Comments": "Additional comments",
     "Latency": "125.0"
   }
```

5.0 Security

In versions 7.0.0 through 7.0.1 of the GroundWork Rest API, all APIs were protected by HTTP Basic Authentication. Every request you made to this web service must provide HTTP Authentication headers. Starting with version 7.0.2, the Rest API has discontinued support of HTTP Basic Authentication.

5.1 Token Based Authentication

With version 7.0.2 onwards, Token-based Authentication is required. Token-based Authentication introduces a new, optimized authentication algorithm that will greatly optimize and improve the performance of the Rest APIs. Instead of requiring authentication with each and every request, authentication is only required once, to retrieve a token. This trusted token can then be used on all subsequent API calls to authorize the requesting application access to protected resources. This token must be passed with each request as an HTTP header.

5.2 Token Session Validity Duration

A token is on valid for a configured lifetime. The duration of this lifetime defaults to 8 hours. When making an API call, it is possible that the token can expire. In this case, your application will need to handle re-authentication with credentials. Note that our Java and Perl Client APIs will handle this for you automatically.

5.3 Authentication Process

Before making any API requests, you must first make an authentication request. Authentication is achieved by call the `/auth/login` API with encoded and/or encrypted credentials:
`curl -i -X POST -d "user=d3N1c2Vy&password=d3N1c2Vy&gwos-app-name=cloudhub"` http://localhost/api/auth/login
The user must always be Base64 encoded. If security encryption is enabled, the encrypted password should be passed without additional encoding; otherwise, unencrypted passwords must be Base64 encoded. If authentication succeeds, a 200 response code is returned along with a single token in the response. This token must be passed on to all subsequent calls in the GWOS-API-TOKEN HTTP header:
`curl -H "Accept: application/xml" -H "GWOS-API-TOKEN:24528E5E970AFEC8D1E887DD9CA2C825" -H "GWOS-APP-NAME:cloudhub"` http://localhost:8080/foundation-webapp/api/categories
An additional APP-NAME HTTP header is required. To summarize, all API calls (except `auth/login`) will always require two HTTP headers:
GWOS-API-TOKEN - the token return by /api/auth/login
GWOS-APP-NAME - identify your application with this header

5.4 Re-Authentication Process

When a token expires, applications will need to handle this exception. A request with an expired token will return an *HTTP Status Code 401 Unauthorized*.

> ⚠ *Note that our Java and Perl Client APIs will handle re-authentication for you automatically.*

When this happens, your application will need to re-authenticate with the `/api/auth/login` API retrieving a new token. With this new token in the HTTP header GWOS-API-TOKEN, you can then resubmit your request.

See the Auth APIs in the RESTful Services section for more details.