# GenieACS Documentation

***Release 1.2.16***

**GenieACS Inc.**

**Mar 29, 2026**

# TABLE OF CONTENTS


I Installation 1

1 Installation Guide 3
1.1 Prerequisites . . . 3
1.2 Install GenieACS . . . 3
1.3 Configure systemd . . . 4

2 Environment Variables 9

II Administration 13

3 Provisions 15
3.1 Built-in functions . . . 15
3.2 Path format . . . 17
3.3 Creating/deleting object instances . . . 17
3.4 Special GenieACS parameters . . . 18

4 Virtual Parameters 21
4.1 Examples . . . 21

5 Administration FAQ 25
5.1 Duplicate log entries when using `log()` function . . . 25
5.2 Configurations not pushed to device after factory reset . . . 25
5.3 Most device parameters are missing . . . 26

III Integration 27

6 Extensions 29

7 API Reference 31
7.1 Endpoints . . . 31
7.2 Tasks . . . 37
7.3 Presets . . . 39
7.4 Provisions . . . 40

IV Security 41

8 HTTPS 43

**9 CPE Authentication** **45**
9.1 CPE to ACS . . . . . 45
9.2 ACS to CPE . . . . . 46

**10 Roles and Permissions** **47**

# Part I

# Installation

CHAPTER

ONE

# INSTALLATION GUIDE

This guide is for installing GenieACS on a single server on any Linux distro that uses *systemd* as its init system.

The various GenieACS services are independent of each other and may be installed on different servers. You may also run multiple instances of each in a load-balancing/failover setup.

**Attention**

For production deployments make sure to configure TLS and change `UI_JWT_SECRET` to a unique and secure string. Refer to *HTTPS* section for how to enable TLS to encrypt traffic.

## 1.1 Prerequisites

**Node.js**

GenieACS requires Node.js 12.13 and up. Refer to https://nodejs.org/ for instructions.

**MongoDB**

GenieACS requires MongoDB 3.6 and up. Refer to https://www.mongodb.com/ for instructions.

## 1.2 Install GenieACS

**Installing from NPM:**

```
sudo npm install -g genieacs@1.2.16
```

**Installing from source**

If you prefer installing from source, such as when running a GenieACS copy with custom patches, refer to README.md file in the source package. Adjust the next steps below accordingly.

## 1.3 Configure systemd

**Create a system user to run GenieACS daemons**

```
sudo useradd --system --no-create-home --user-group genieacs
```

**Create directory to save extensions and environment file**

We'll use `/opt/genieacs/ext/` directory to store extension scripts (if any).

```
mkdir /opt/genieacs
mkdir /opt/genieacs/ext
chown genieacs:genieacs /opt/genieacs/ext
```

Create the file `/opt/genieacs/genieacs.env` to hold our configuration options which we pass to GenieACS as environment variables. See *Environment Variables* section for a list of all available configuration options.

```
GENIEACS_CWMP_ACCESS_LOG_FILE=/var/log/genieacs/genieacs-cwmp-access.log
GENIEACS_NBI_ACCESS_LOG_FILE=/var/log/genieacs/genieacs-nbi-access.log
GENIEACS_FS_ACCESS_LOG_FILE=/var/log/genieacs/genieacs-fs-access.log
GENIEACS_UI_ACCESS_LOG_FILE=/var/log/genieacs/genieacs-ui-access.log
GENIEACS_DEBUG_FILE=/var/log/genieacs/genieacs-debug.yaml
NODE_OPTIONS=--enable-source-maps
GENIEACS_EXT_DIR=/opt/genieacs/ext
```

Generate a secure JWT secret and append to `/opt/genieacs/genieacs.env`:

```
node -e "console.log(\"GENIEACS_UI_JWT_SECRET=\" + require('crypto').randomBytes(128).
↪toString('hex'))" >> /opt/genieacs/genieacs.env
```

Set file ownership and permissions:

```
sudo chown genieacs:genieacs /opt/genieacs/genieacs.env
sudo chmod 600 /opt/genieacs/genieacs.env
```

**Create logs directory**

```
mkdir /var/log/genieacs
chown genieacs:genieacs /var/log/genieacs
```

**Create systemd unit files**

Create a systemd unit file for each of the four GenieACS services. Note that we're using EnvironmentFile directive to read the environment variables from the file we created earlier.

Each service has two streams of logs: access log and process log. Access logs are configured here to be dumped in a log file under `/var/log/genieacs/` while process logs go to *journald*. Use `journalctl` command to view process logs.

> **Attention**
>
> If the command **`systemctl edit --force --full`** fails, you can create the unit file manually.

1. Run the following command to create `genieacs-cwmp` service:

   ```
   sudo systemctl edit --force --full genieacs-cwmp
   ```

   Then paste the following in the editor and save:

   ```
   [Unit]
   Description=GenieACS CWMP
   After=network.target

   [Service]
   User=genieacs
   EnvironmentFile=/opt/genieacs/genieacs.env
   ExecStart=/usr/bin/genieacs-cwmp

   [Install]
   WantedBy=default.target
   ```

2. Run the following command to create `genieacs-nbi` service:

   ```
   sudo systemctl edit --force --full genieacs-nbi
   ```

   Then paste the following in the editor and save:

   ```
   [Unit]
   Description=GenieACS NBI
   After=network.target

   [Service]
   User=genieacs
   EnvironmentFile=/opt/genieacs/genieacs.env
   ExecStart=/usr/bin/genieacs-nbi

   [Install]
   WantedBy=default.target
   ```

3. Run the following command to create `genieacs-fs` service:

   ```
   sudo systemctl edit --force --full genieacs-fs
   ```

   Then paste the following in the editor and save:

```
[Unit]
Description=GenieACS FS
After=network.target

[Service]
User=genieacs
EnvironmentFile=/opt/genieacs/genieacs.env
ExecStart=/usr/bin/genieacs-fs

[Install]
WantedBy=default.target
```

4. Run the following command to create `genieacs-ui` service:

```
sudo systemctl edit --force --full genieacs-ui
```

Then paste the following in the editor and save:

```
[Unit]
Description=GenieACS UI
After=network.target

[Service]
User=genieacs
EnvironmentFile=/opt/genieacs/genieacs.env
ExecStart=/usr/bin/genieacs-ui

[Install]
WantedBy=default.target
```

**Configure log file rotation using logrotate**

Save the following as `/etc/logrotate.d/genieacs`

```
/var/log/genieacs/*.log /var/log/genieacs/*.yaml {
    daily
    rotate 30
    compress
    delaycompress
    dateext
}
```

**Enable and start services**

```
sudo systemctl enable genieacs-cwmp
sudo systemctl start genieacs-cwmp
sudo systemctl status genieacs-cwmp

sudo systemctl enable genieacs-nbi
```

```
sudo systemctl start genieacs-nbi
sudo systemctl status genieacs-nbi

sudo systemctl enable genieacs-fs
sudo systemctl start genieacs-fs
sudo systemctl status genieacs-fs

sudo systemctl enable genieacs-ui
sudo systemctl start genieacs-ui
sudo systemctl status genieacs-ui
```

Review the status message for each to verify that the services are running successfully.

CHAPTER

TWO

# ENVIRONMENT VARIABLES

Configuring GenieACS services can be done through the following environment variables:

> **Attention**
>
> All GenieACS environment variables must be prefixed with `GENIEACS_`.

**MONGODB_CONNECTION_URL**
MongoDB connection string.

Default: `mongodb://127.0.0.1/genieacs`

**EXT_DIR**
The directory from which to look up extension scripts.

Default: `<installation dir>/config/ext`

**EXT_TIMEOUT**
Timeout (in milliseconds) to allow for calls to extensions to return a response.

Default: `3000`

**DEBUG_FILE**
File to dump CPE debug log.

Default: unset

**DEBUG_FORMAT**
Debug log format. Valid values are 'yaml' and 'json'.

Default: `yaml`

**LOG_FORMAT**
The format used for the log entries in `CWMP_LOG_FILE`, `NBI_LOG_FILE`, `FS_LOG_FILE`, and `UI_LOG_FILE`. Possible values are `simple` and `json`.

Default: `simple`

**ACCESS_LOG_FORMAT**
The format used for the log entries in `CWMP_ACCESS_LOG_FILE`, `NBI_ACCESS_LOG_FILE`, `FS_ACCESS_LOG_FILE`, and `UI_ACCESS_LOG_FILE`. Possible values are `simple` and `json`.

Default: `simple`

**CWMP_WORKER_PROCESSES**
The number of worker processes to spawn for genieacs-cwmp. A value of 0 means as many as there are CPU cores available.

Default: `0`

**CWMP_PORT**
The TCP port that genieacs-cwmp listens on.

Default: `7547`

**CWMP_INTERFACE**
The network interface that genieacs-cwmp binds to.

Default: `::`

**CWMP_SSL_CERT**
Path to certificate file. If omitted, non-secure HTTP will be used.

Default: unset

**CWMP_SSL_KEY**
Path to certificate key file. If omitted, non-secure HTTP will be used.

Default: unset

**CWMP_LOG_FILE**
File to log process related events for genieacs-cwmp. If omitted, logs will go to stderr.

Default: unset

**CWMP_ACCESS_LOG_FILE**
File to log incoming requests for genieacs-cwmp. If omitted, logs will go to stdout.

Default: unset

**NBI_WORKER_PROCESSES**
The number of worker processes to spawn for genieacs-nbi. A value of 0 means as many as there are CPU cores available.

Default: `0`

**NBI_PORT**
The TCP port that genieacs-nbi listens on.

Default: `7557`

**NBI_INTERFACE**
The network interface that genieacs-nbi binds to.

Default: `::`

**NBI_SSL_CERT**
Path to certificate file. If omitted, non-secure HTTP will be used.

Default: unset

**NBI_SSL_KEY**
Path to certificate key file. If omitted, non-secure HTTP will be used.

Default: unset

**NBI_LOG_FILE**
File to log process related events for genieacs-nbi. If omitted, logs will go to stderr.

Default: unset

**NBI_ACCESS_LOG_FILE**
File to log incoming requests for genieacs-nbi. If omitted, logs will go to stdout.

Default: unset

**FS_WORKER_PROCESSES**
The number of worker processes to spawn for genieacs-fs. A value of 0 means as many as there are CPU cores available.

Default: `0`

**FS_PORT**
The TCP port that genieacs-fs listens on.

Default: `7567`

**FS_INTERFACE**
The network interface that genieacs-fs binds to.

Default: `::`

**FS_SSL_CERT**
Path to certificate file. If omitted, non-secure HTTP will be used.

Default: unset

**FS_SSL_KEY**
Path to certificate key file. If omitted, non-secure HTTP will be used.

Default: unset

**FS_LOG_FILE**
File to log process related events for genieacs-fs. If omitted, logs will go to stderr.

Default: unset

**FS_ACCESS_LOG_FILE**
File to log incoming requests for genieacs-fs. If omitted, logs will go to stdout.

Default: unset

**FS_URL_PREFIX**
The URL prefix (e.g. 'https://example.com:7567/') to use when generating the file URL for TR-069 Download requests. Set this if genieacs-fs and genieacs-cwmp are behind a proxy or running on different servers.

Default: auto generated based on the hostname from the ACS URL, FS_PORT config, and whether or not SSL is enabled for genieacs-fs.

**UI_WORKER_PROCESSES**
The number of worker processes to spawn for genieacs-ui. A value of 0 means as many as there are CPU cores available.

Default: `0`

**UI_PORT**
The TCP port that genieacs-ui listens on.

Default: `3000`

**UI_INTERFACE**
The network interface that genieacs-ui binds to.

Default: `::`

**UI_SSL_CERT**
Path to certificate file. If omitted, non-secure HTTP will be used.

Default: unset

**UI_SSL_KEY**
Path to certificate key file. If omitted, non-secure HTTP will be used.

Default: unset

**UI_LOG_FILE**
File to log process related events for genieacs-ui. If omitted, logs will go to stderr.

Default: unset

**UI_ACCESS_LOG_FILE**
File to log incoming requests for genieacs-ui. If omitted, logs will go to stdout.

Default: unset

**UI_JWT_SECRET**
The key used for signing JWT tokens that are stored in browser cookies. The string can be up to 64 characters in length.

Default: unset

# Part II

# Administration

CHAPTER

# THREE

# PROVISIONS

A Provision is a piece of JavaScript code that is executed on the server on a per-device basis. It enables implementing complex provisioning scenarios and other operations such as automated firmware upgrade rollout. Apart from a few special functions, the script is essentially a standard ES6 code executed in strict mode.

Provisions are mapped to devices using presets. Note that the added performance overhead when using Provisions as opposed to simple preset configuration entries is relatively small. Anything that can be done via preset configurations can be done using a Provision script. In fact, the now deprecated configuration format is still supported primarily for backward compatibility and it is recommended to use Provision scripts for all configuration.

When assigning a Provision script to a preset, you may pass arguments to the script. The arguments can be accessed from the script through the global `args` variable.

**Note**

Provision scripts may get executed multiple times in a given session. Although all data model-mutating operations are idempotent, a script as a whole may not be. It is, therefore, necessary to repeatedly run the script until there are no more side effects and a stable state is reached.

## 3.1 Built-in functions

### 3.1.1 `declare(path, timestamps, values)`

This function is for declaring parameter values to be set, as well as specify constraints on how recent you'd like the parameter value (or other attributes) to have been refreshed from the device. If the given timestamp is lower than the timestamp of the last refresh from the device, then this function will return the last known value. Otherwise, the value will be fetched from the device before being returned to the caller.

The timestamp argument is an object where the key is the attribute name (e.g. `value`, `object`, `writable`, `path`) and the value is an integer representing a Unix timestamp.

The values argument is an object similar to the timestamp argument but its property values being the parameter values to be set.

The possible attributes in 'timestamps' and 'values' arguments are:

- `value`: a [<value>, <type>] pair

This attribute is not available for objects or object instances. If the value is not a [<value>, <type>] array then it'll assumed to be a value without a type and therefore the type will be inferred from the parameter's type.

- `writable`: boolean

The meaning of this attribute can vary depending on the type of the parameter. In the case of regular parameters, it indicates if its value is writable. In the case of objects, it's whether or not it's possible to add new object instances. In the case of object instances, it indicates whether or not this instance can be deleted.

- `object`: boolean

True if this is an object or object instance, false otherwise.

- `path`: string

This attribute is special in that it's not a parameter attribute per se, but it refers to the presence of parameters matching the given path. For example, given the following wildcard path:

`InternetGatewayDevice.LANDevice.1.Hosts.Host.*.MACAddress`

Using a recent timestamp for path in `declare()` will result in a sync with the device to rediscover all Host instances (`Host.*`). The path attribute can also be used to create or delete object instances as described in *Path format* section.

The return value of `declare()` is an iterator to access parameters that match the given path. Each item in the iterator has the attribute 'path' in addition to any other attribute given in the `declare()` call. The iterator object itself has convenience attribute accessors which come in handy when you're expecting a single parameter (e.g. when path does not contain wildcards or aliases).

```
// Example: Setting the SSID as the last 6 characters of the serial number
let serial = declare("Device.DeviceInfo.SerialNumber", {value: 1});
declare("Device.LANDevice.1.WLANConfiguration.1.SSID", null, {value: serial.value[0]});
```

### 3.1.2 clear(path, timestamp)

This function invalidates the database copy of parameters (and their child parameters) that match the given path and have a last refresh timestamp that is less than the given timestamp. The most obvious use for this function is to invalidate the database copy of the entire data model after the device has been factory reset:

```
// Example: Clear cached device data model Note
// Make sure to apply only on "0 BOOTSTRAP" event
clear("Device", Date.now());
clear("InternetGatewayDevice", Date.now());
```

### 3.1.3 commit()

This function commits the pending declarations and performs any necessary sync with the device. It's usually not required to call this function as it called implicitly at the end of the script and when accessing any property of the promise-like object returned by the `declare()` function. Calling this explicitly is only necessary if you want to control the order in which parameters are configured.

### 3.1.4 ext(file, function, arg1, arg2, ...)

Execute an extension script and return the result. The first argument is the script filename while second argument is the function name within that script. Any remaining arguments will be passed to that function. See *Extensions* for more details.

### 3.1.5 log(message)

Prints out a string in genieacs-cwmp's access log. It's meant to be used for debugging. Note that you may see multiple log entries as the script can be executed multiple times in a session. See *this FAQ*.

## 3.2 Path format

A parameter path may contain a wildcard (*) or an alias filter (`[name:value]`). A wildcard segment in a parameter path will apply the declared configuration to zero or more parameters that match the given path where the wildcard segment can be anything.

An alias filter is like a wildcard, but additionally performs filtering on the child parameters based on the key-value pairs provided. For example, the following path:

`Device.WANDevice.1.WANConnectionDevice.1.WANIPConnection.[AddressingType:DHCP].ExternalIPAddress`

will return a list of ExternalIPAddress parameters (0 or more) where the sibling parameter AddressingType is assigned the value "DHCP".

This can be useful when the exact instance numbers may be different from one device to another. It is possible to use more than one key-value pair in the alias filter. It's also possible to use multiple filters or use a combination of filters and wildcards.

## 3.3 Creating/deleting object instances

Given the declarative nature of provisions, we cannot explicitly tell the device to create or delete an instance under a given object. Instead, we specify the number of instances we want there to be, and based on that GenieACS will determine whether or not it needs to create or delete instances. For example, the following declaration will ensure we have one and only one WANIPConnection object:

```
// Example: Ensure we have one and only one WANIPConnection object
declare("InternetGatewayDevice.WANDevice.1.WANConnectionDevice.1.WANIPConnection.*",␣
↪null, {path: 1});
```

Note the wildcard at the end of the parameter path.

It is also possible to use alias filters as the last path segment which will ensure that the declared number of instances is satisfied given the alias filter:

```
// Ensure that *all* other instances are deleted
declare("InternetGatewayDevice.X_BROADCOM_COM_IPAddrAccCtrl.X_BROADCOM_COM_
↪IPAddrAccCtrlListCfg.[]", null, {path: 0});

// Add the two entries we care about
declare("InternetGatewayDevice.X_BROADCOM_COM_IPAddrAccCtrl.X_BROADCOM_COM_
```

(continues on next page)

(continued from previous page)

```
↪IPAddrAccCtrlListCfg.[SourceIPAddress:192.168.1.0,SourceNetMask:255.255.255.0]",
↪{path: now}, {path: 1});
declare("InternetGatewayDevice.X_BROADCOM_COM_IPAddrAccCtrl.X_BROADCOM_COM_
↪IPAddrAccCtrlListCfg.[SourceIPAddress:172.16.12.0,SourceNetMask:255.255.0.0]", {path:␣
↪now}, {path: 1});
```

## 3.4 Special GenieACS parameters

In addition to the parameters exposed in the device's data model through TR-069, GenieACS has its own set of special parameters:

### 3.4.1 DeviceID

This parameter sub-tree includes the following read-only parameters:

- `DeviceID.ID`
- `DeviceID.SerialNumber`
- `DeviceID.ProductClass`
- `DeviceID.OUI`
- `DeviceID.Manufacturer`

### 3.4.2 Tags

The `Tags` root parameter is used to expose device tags in the data model. Tags appear as child parameters that are writable and have boolean value. Setting a tag to `false` will delete that tag, and setting the value of a non-existing tag parameter to `true` will create it.

```
// Example: Remove "tag1", add "tag2", and read "tag3"
declare("Tags.tag1", null, {value: false});
declare("Tags.tag2", null, {value: true});
let tag3 = declare("Tags.tag3", {value: 1});
```

### 3.4.3 Reboot

The `Reboot` root parameter hold the timestamp of the last reboot command. The parameter value is writable and declaring a timestamp value that is larger than the current value will trigger a reboot.

```
// Example: Reboot the device only if it hasn't been rebooted in the past 300 seconds
declare("Reboot", null, {value: Date.now() - (300 * 1000)});
```

### 3.4.4 FactoryReset

Works like `Reboot` parameter but for factory reset.

```
// Example: Default the device to factory settings
declare("FactoryReset", null, {value: Date.now()});
```

### 3.4.5 Downloads

The `Downloads` sub-tree holds information about the last download command(s). A download command is represented as an instance (e.g. `Downloads.1`) containing parameters such as `Download` (timestamp), `LastFileType`, `LastFileName`. The parameters `FileType`, `FileName`, `TargetFileName` and `Download` are writable and can be used to trigger a new download.

```
declare("Downloads.[FileType:1 Firmware Upgrade Image]", {path: 1}, {path: 1});
declare("Downloads.[FileType:1 Firmware Upgrade Image].FileName", {value: 1}, {value:
↪"firmware-2017.01.tar"});
declare("Downloads.[FileType:1 Firmware Upgrade Image].Download", {value: 1}, {value:␣
↪Date.now()});
```

Common file types are:

- `1 Firmware Upgrade Image`
- `2 Web Content`
- `3 Vendor Configuration File`
- `4 Tone File`
- `5 Ringer File`

> **Warning**
>
> Pushing a file to the device is often a service-interrupting operation. It's recommended to only trigger it on certain events such as `1 BOOT` or during a predetermined maintenance window).

After the CPE had finished downloading and applying the config file, it will send a `7 TRANSFER COMPLETE` event. You may use that to trigger a reboot after the firmware image or configuration file had been applied.

CHAPTER

FOUR

# VIRTUAL PARAMETERS

Virtual parameters are user-defined parameters whose values are generated using a custom Javascript code. Virtual parameters behave just like regular parameters and appear in the data model under `VirtualParameters.` path. Virtual parameter names cannot contain a period (`.`).

The execution environment for virtual parameters is almost identical to that of provisions. See *Provisions* for more details and examples. The only differences between the scripts of provisions and virtual parameters are:

- You can't pass custom arguments to virtual parameter scripts. Instead, the variable `args` will hold the current vparam timestamps and values as well as the declared timestamps and values. Like this:

```
// [<declared attr timestamps, declared attr values>, <current attr timestamps>,
↪<current attr values>]
[{path: 1559849387191, value: 1559849387191}, {value: ["new val", "xsd:string"]}, {path:␣
↪1559840000000, value: 1559840000000}, {value: ["cur val", "xsd:string"]}]
```

- Virtual parameter scripts must return an object containing the attributes of this parameter.

**Note**

Just like a regular parameter, creating a virtual parameter does not automatically add it to the parameter list for a device. It needs to fetched (manually or via a preset) before you can see it in the data model.

## 4.1 Examples

### 4.1.1 Unified MAC parameter across different device models

```
// Example: Unified MAC parameter across different device models
let m = "00:00:00:00:00:00";
let d = declare("Device.WANDevice.*.WANConnectionDevice.*.WANIPConnection.*.MACAddress",
↪{value: Date.now()});
let igd = declare("InternetGatewayDevice.WANDevice.*.WANConnectionDevice.*.
↪WANPPPConnection.*.MACAddress", {value: Date.now()});

if (d.size) {
  for (let p of d) {
    if (p.value[0]) {
      m = p.value[0];
      break;
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
else if (igd.size) {
  for (let p of igd) {
    if (p.value[0]) {
      m = p.value[0];
      break;
    }
  }
}

return {writable: false, value: [m, "xsd:string"]};
```

### 4.1.2 Expose an external value as a virtual parameter

```
// Example: Expose an external value as a virtual parameter
let serial = declare("DeviceID.SerialNumber", {value: 1});
if (args[1].value) {
  ext("example-ext", "set", serial.value[0], args[1].value[0]);
  return {writable: true, value: [args[1].value[0], "xsd:string"]};
}
else {
  let v = ext("example-ext", "get", serial.value[0]);
  return {writable: true, value: [v, "xsd:string"]};
}
```

### 4.1.3 Create an editable virtual parameter for WPA passphrase

```
// Example: Create an editable virtual parameter for WPA passphrase
let m = "";
if (args[1].value) {
  m = args[1].value[0];
  declare("Device.WiFi.AccessPoint.1.Security.KeyPassphrase", null, {value: m});
  declare("InternetGatewayDevice.LANDevice.1.WLANConfiguration.1.KeyPassphrase", null,
↪{value: m});
}
else {
  let d = declare("Device.WiFi.AccessPoint.1.Security.KeyPassphrase", {value: Date.now()}
↪);
  let igd = declare("InternetGatewayDevice.LANDevice.1.WLANConfiguration.1.KeyPassphrase
↪", {value: Date.now()});

  if (d.size) {
    m = d.value[0];
  }
  else if (igd.size) {
    m = igd.value[0];
  }
```

(continues on next page)

(continued from previous page)

```
}

return {writable: true, value: [m, "xsd:string"]};
```

CHAPTER

FIVE

# ADMINISTRATION FAQ

## 5.1 Duplicate log entries when using `log()` function

Because GenieACS uses a full fledged scripting language for device configuration, the only way to guarantee that it has satisfied the 'desired state' is by repeatedly executing the script until there's no more discrepancies with the current device state. Though it may seem like this will cause duplicate requests going to the device, this isn't actually the case because device configuration are stated declaratively and that the scripts themselves are pure functions in the context of a session (e.g. Date.now() always returns the same value within the session).

To illustrate with an example, consider the following script:

```
log("Executing script");
declare("Device.param", null, {value: 1});
commit();
declare("Device.param", null, {value: 2});
```

This will set the value of the 'Device.param' to 1, then to 2. Then as the script is run again the value is set back to 1 and so on. A stable state will never be reached so GenieACS will execute the script a few times until it gives up and throws a fault. This is an edge case that should be avoided. A more typical case is where the script is run once or twice. Essentially if an execution doesn't result in any request to the CPE or a change in the data model then a stable state is deemed to have been reached.

## 5.2 Configurations not pushed to device after factory reset

After a device is reset to its factory default state, the cached data model in GenieACS's database needs to be invalidated to force rediscovery. Ensure the following lines are called on `0 BOOTSTRAP` event:

```
const now = Date.now();

// Clear cached data model to force a refresh
clear("Device", now);
clear("InternetGatewayDevice", now);
```

## 5.3 Most device parameters are missing

For performance reasons (server, client, and network), GenieACS by default only fetches parts of the data model that are necessary to satisfy the declarations in your provision scripts. Create declarations for any parameters you need fetched by default.

If you're unsure and want to explore the available parameters exposed by the device, refresh the root parameter (e.g. `InternetGatewayDevice`) from GenieACS's UI. You typically only need to do that one time for a given CPE model.

# Part III

# Integration

CHAPTER

SIX

# EXTENSIONS

Given that *Provisions* and *Virtual Parameters* are executed in a sandbox environment, it is not possible to interact with external sources or execute any action that requires OS, file system, or network access. Extensions exist to bridge that gap.

Extensions are fully-privileged Node.js modules and as such have access to standard Node libraries and 3rd party packages. Functions exposed by the extension can be called from Provision scripts using the `ext()` function. A typical use case for extensions is fetching credentials from a database to have that pushed to the device during provisioning.

By default, the extension JS code must be placed under `config/ext` directory. You may need to create that directory if it doesn't already exist.

The example extension below fetches data from an external REST API and returns that to the caller:

```
// This is an example GenieACS extension to get the current latitude/longitude
// of the International Space Station. Why, you ask? Because why not.
// To install, copy this file to config/ext/iss.js.

"use strict";

const http = require("http");

let cache = null;
let cacheExpire = 0;

function latlong(args, callback) {
  if (Date.now() < cacheExpire) return callback(null, cache);

  http
    .get("http://api.open-notify.org/iss-now.json", (res) => {
      if (res.statusCode !== 200)
        return callback(
          new Error(`Request failed (status code: ${res.statusCode})`),
        );

      let rawData = "";
      res.on("data", (chunk) => (rawData += chunk));

      res.on("end", () => {
        let pos = JSON.parse(rawData)["iss_position"];
        cache = [+pos["latitude"], +pos["longitude"]];
        cacheExpire = Date.now() + 10000;
        callback(null, cache);
```

(continues on next page)

(continued from previous page)

```
      });
    })
    .on("error", (err) => {
      callback(err);
    });
}

exports.latlong = latlong;
```

To call this extension from a Provision or a Virtual Parameter script:

```
// The arguments "arg1" and "arg2" are passed to the latlong. Though they are
// unused in this particular example.
const res = ext("ext-sample", "latlong", "arg1", "arg2");
log(JSON.stringify(res));
```

CHAPTER

SEVEN

# API REFERENCE

GenieACS exposes a rich RESTful API through its NBI component. This document serves as a reference for the available APIs.

This API makes use of MongoDB's query language in some of its endpoints. Refer to MongoDB's documentation for details.

**Note**

The examples below use `curl` command for simplicity and ease of testing. Query parameters are URL-encoded, but the original pre-encoding values are shown for reference. These examples assume genieacs-nbi is running locally and listening on the default NBI port (7557).

**Warning**

A common pitfall is not properly percent-encoding special characters in the device ID or query in the URL.

## 7.1 Endpoints

### 7.1.1 GET /<collection>/?query=<query>

Search for records in the database (e.g. devices, tasks, presets, files). Returns a JSON representation of all items in the given collection that match the search criteria.

*collection*: The data collection to search. Could be one of: tasks, devices, presets, objects.

*query*: Search query. Refer to MongoDB queries for reference.

#### Examples

- Find a device by its ID:

```
query = {"_id": "202BC1-BM632w-000000"}
```

```
curl -i 'http://localhost:7557/devices/?query=%7B%22_id%22%3A%22202BC1-BM632w-000000%22
↪%7D'
```

- Find a device by its MAC address:

```
query = {
  "InternetGatewayDevice.WANDevice.1.WANConnectionDevice.1.WANIPConnection.1.MACAddress
↪": "20:2B:C1:E0:06:65"
}
```

```
curl -i 'http://localhost:7557/devices/?query=%7B%22InternetGatewayDevice.WANDevice.1.
↪WANConnectionDevice.1.WANIPConnection.1.MACAddress%22%3A%2220:2B:C1:E0:06:65%22%7D'
```

- Search for devices that have not initiated an inform in the last 7 days.

```
query = {
  "_lastInform": {
    "$lt" : "2017-12-11 13:16:23 +0000"
  }
}
```

```
curl -i 'http://localhost:7557/devices/?query=%7B%22_lastInform%22%3A%7B%22%24lt%22%3A
↪%222017-12-11%2013%3A16%3A23%20%2B0000%22%7D%7D'
```

- Show pending tasks for a given device:

```
query = {"device": "202BC1-BM632w-000000"}
```

```
curl -i 'http://localhost:7557/tasks/?query=%7B%22device%22%3A%22202BC1-BM632w-000000%22
↪%7D'
```

- Return specific parameters for a given device:

```
query = {"_id": "202BC1-BM632w-000000"}
```

```
curl -i 'http://localhost:7557/devices?query=%7B%22_id%22%3A%22202BC1-BM632w-000000%22
↪%7D&projection=InternetGatewayDevice.DeviceInfo.ModelName,InternetGatewayDevice.
↪DeviceInfo.Manufacturer'
```

The `projection` URL param is a comma-separated list of the parameters to receive.

### 7.1.2 POST /devices/<device_id>/tasks?[connection_request]

Enqueue task(s) and optionally trigger a connection request to the device. Refer to *Tasks* section for information about the task object format. Returns status code 200 if the tasks have been successfully executed, and 202 if the tasks have been queued to be executed at the next inform.

*device_id*: The ID of the device.

*connection_request*: Indicates that a connection request will be triggered to execute the tasks immediately. Otherwise, the tasks will be queued and be processed at the next inform.

The response body is the task object as it is inserted in the database. The object will include `_id` property which you can use to look up the task later.

#### Examples

- Refresh all device parameters now:

```
curl -i 'http://localhost:7557/devices/202BC1-BM632w-000000/tasks?connection_request' \
-X POST \
--data '{"name": "refreshObject", "objectName": ""}'
```

- Change WiFi SSID and password:

```
{
  "name": "setParameterValues",
  "parameterValues": [
    ["InternetGatewayDevice.LANDevice.1.WLANConfiguration.1.SSID", "GenieACS",
↪"xsd:string"],
    ["InternetGatewayDevice.LANDevice.1.WLANConfiguration.1.PreSharedKey.1.PreSharedKey",
↪ "hello world", "xsd:string"]
  ]
}
```

```
curl -i 'http://localhost:7557/devices/202BC1-BM632w-000000/tasks?connection_request' \
-X POST \
--data '{"name":"setParameterValues", "parameterValues": [["InternetGatewayDevice.
↪LANDevice.1.WLANConfiguration.1.SSID", "GenieACS", "xsd:string"],[
↪"InternetGatewayDevice.LANDevice.1.WLANConfiguration.1.PreSharedKey.1.PreSharedKey",
↪"hello world", "xsd:string"]]}'
```

### 7.1.3 POST /tasks/<task_id>/retry

Retry a faulty task at the next inform.

*task_id*: The ID of the task as returned by 'GET /tasks' request.

#### Example

```
curl -i 'http://localhost:7557/tasks/5403908ef28ea3a25c138adc/retry' -X POST
```

### 7.1.4 DELETE /tasks/<task_id>

Delete the given task.

*task_id*: The ID of the task as returned by 'GET /tasks' request.

#### Example

```
curl -i 'http://localhost:7557/tasks/5403908ef28ea3a25c138adc' -X DELETE
```

### 7.1.5 DELETE /faults/<fault_id>

Delete the given fault.

*fault_id*: The ID of the fault as returned by 'GET /faults' request. The ID format is "<device_id>:<channel>".

#### Example

```
curl -i 'http://localhost:7557/faults/202BC1-BM632w-000000:default' -X DELETE
```

### 7.1.6 DELETE /devices/<device_id>

Delete the given device from the database.

#### Example

```
curl -X DELETE -i 'http://localhost:7557/devices/202BC1-BM632w-000001'
```

> **Note**
>
> Note that the device will be registered again when/if it contacts the ACS again (e.g. on the next periodic inform).

### 7.1.7 POST /devices/<device_id>/tags/<tag>

Assign a tag to a device. Has no effect if such tag already exists.

*device_id*: The ID of the device.

*tag*: The tag to be assigned.

#### Example

Assign the tag "testing" to a device:

```
curl -i 'http://localhost:7557/devices/202BC1-BM632w-000000/tags/testing' -X POST
```

### 7.1.8 DELETE /devices/<device_id>/tags/<tag>

Remove a tag from a device.

*device_id*: The ID of the device.

*tag*: The tag to be removed.

#### Example

Remove the tag "testing" from a device:

```
curl -i 'http://localhost:7557/devices/202BC1-BM632w-000000/tags/testing' -X DELETE
```

### 7.1.9 PUT /presets/<preset_name>

Create or update a preset. Returns status code 200 if the preset has been added/updated successfully. The body of the request is a JSON representation of the preset. Refer to *Presets* section below for details about its format.

*preset_name*: The name of the preset.

#### Example

Create a preset to set 5 minutes inform interval for all devices tagged with "test":

```
query = {
  "weight": 0,
  "precondition": "{\"_tags\": \"test\"}"
  "configurations": [
    {
      "type": "value",
      "name": "InternetGatewayDevice.ManagementServer.PeriodicInformEnable",
      "value": "true"
    },
    {
      "type": "value",
      "name": "InternetGatewayDevice.ManagementServer.PeriodicInformInterval",
      "value": "300"
    }
  ]
}
```

```
curl -i 'http://localhost:7557/presets/inform' \
-X PUT \
--data '{"weight": 0, "precondition": "{\"_tags\": \"test\"}", "configurations": [{"type
↪": "value", "name": "InternetGatewayDevice.ManagementServer.PeriodicInformEnable",
↪"value": "true"}, {"type": "value", "name": "InternetGatewayDevice.ManagementServer.
↪PeriodicInformInterval", "value": "300"}]}'
```

### 7.1.10 DELETE /presets/<preset_name>

```
curl -i 'http://localhost:7557/presets/inform' -X DELETE
```

### 7.1.11 PUT /files/<file_name>

Upload a new file or overwrite an existing one. Returns status code 200 if the file has been added/updated successfully. The file content should be sent as the request body.

*file_name*: The name of the uploaded file.

The following file metadata may be sent as request headers:

- `fileType`: For firmware images it should be "1 Firmware Upgrade Image". Other common types are "2 Web Content" and "3 Vendor Configuration File".
- `oui`: The OUI of the device model that this file belongs to.
- `productClass`: The product class of the device.
- `version`: In case of firmware images, this refer to the firmware version.

#### Example

Upload a firmware image file:

```
curl -i 'http://localhost:7557/files/new_firmware_v1.0.bin' \
-X PUT \
--data-binary @"./new_firmware_v1.0.bin" \
--header "fileType: 1 Firmware Upgrade Image" \
--header "oui: 123456" \
--header "productClass: ABC" \
--header "version: 1.0"
```

### 7.1.12 DELETE /files/<file_name>

Delete a previously uploaded file:

```
curl -i 'http://localhost:7557/files/new_firmware_v1.0.bin' -X DELETE
```

### 7.1.13 GET /files/

Gets all previously uploaded files.

### 7.1.14 GET /files/?query={"filename":"<filename>"}

Find files using a query.

## 7.2 Tasks

Find the different available tasks and their object structure.

### 7.2.1 getParameterValues

```
query = {
  "name": "getParameterValues",
  "parameterNames": [
    "InternetGatewayDevice.WANDevice.1.WANConnectionDevice.1.
↪WANIPConnectionNumberOfEntries",
    "InternetGatewayDevice.Time.NTPServer1", "InternetGatewayDevice.Time.Status"
  ]
}
```

```
curl -i 'http://localhost:7557/devices/00236a-96318REF-SR360NA0A4%252D0003196/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name": "getParameterValues", "parameterNames": ["InternetGatewayDevice.
↪WANDevice.1.WANConnectionDevice.1.WANIPConnectionNumberOfEntries",
↪"InternetGatewayDevice.Time.NTPServer1", "InternetGatewayDevice.Time.Status"] }'
```

You may request a single or multiple parameters at once.

After the task has been executed successfully you can then fetch the CPE object and read the parameters from the JSON object.

```
query = {"_id": "00236a-96318REF-SR360NA0A4%2D0003196"}
```

```
curl -i 'http://localhost:7557/devices/?query=%7B%22_id%22%3A%2200236a-96318REF-
↪SR360NA0A4%252D0003196%22%7D'
```

### 7.2.2 refreshObject

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name": "refreshObject", "objectName": "InternetGatewayDevice.WANDevice.1.
↪WANConnectionDevice"}'
```

### 7.2.3 setParameterValues

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name": "setParameterValues", "parameterValues": [["InternetGatewayDevice.
↪ManagementServer.UpgradesManaged",false]]}'
```

Multiple values can be set at once by adding multiple arrays to the parameterValues key. For example:

```
{
  name: "setParameterValues",
  parameterValues: [["InternetGatewayDevice.ManagementServer.UpgradesManaged", false], [
↪"InternetGatewayDevice.Time.Enable", true], ["InternetGatewayDevice.Time.NTPServer1",
↪"pool.ntp.org"]]
}
```

### 7.2.4 addObject

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name":"addObject","objectName":"InternetGatewayDevice.WANDevice.1.
↪WANConnectionDevice.1.WANPPPConnection"}'
```

### 7.2.5 deleteObject

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name":"deleteObject","objectName":"InternetGatewayDevice.WANDevice.1.
↪WANConnectionDevice.1.WANPPPConnection.1"}'
```

### 7.2.6 reboot

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name": "reboot"}'
```

### 7.2.7 factoryReset

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name": "factoryReset"}'
```

### 7.2.8 download

```
curl -i 'http://localhost:7557/devices/00236a-SR552n-SR552NA084%252D0003269/tasks?
↪timeout=3000&connection_request' \
-X POST \
--data '{"name": "download", "file": "mipsbe-6-42-lite.xml"}'
```

## 7.3 Presets

Presets assign a set of configuration or a Provision script to devices based on a precondition (search filter), schedule (cron expression), and events.

### 7.3.1 Precondition

The `precondition` property is a JSON string representation of the search filter to test if the preset applies to a given device. Examples preconditions are:

- `{"param": "value"}`
- `{"param": value", "param2": {"$ne": "value2"}}`

Other operators that can be used are `$gt`, `$lt`, `$gte` and `$lte`.

### 7.3.2 Configuration

The configuration property is an array containing the different configurations to be applied to a device, as shown below:

```
[
  {
    "type": "value",
    "name": "InternetGatewayDevice.ManagementServer.PeriodicInformEnable",
    "value": "true"
  },
  {
    "type": "value",
    "name": "InternetGatewayDevice.ManagementServer.PeriodicInformInterval",
    "value": "300"
  },
  {
    "type": "delete_object",
    "name": "object_parent",
    "object": "object_name"
```

(continues on next page)

(continued from previous page)

```
  },
  {
    "type": "add_object",
    "name": "object_parent",
    "object": "object_name"
  },
  {
    "type": "provision",
    "name": "YourProvisionName"
  },
]
```

The configuration type `provision` triggers a Provision script. In the example above, the provision named "YourProvisionName" will be executed.

## 7.4 Provisions

### 7.4.1 Create a provision

The Provision's JavaScript code is the body of the HTTP PUT request.

```
curl -X PUT -i 'http://localhost:7557/provisions/mynewprovision' --data 'log("Provision␣
↪started at " + now);'
```

### 7.4.2 Delete a provision

```
curl -X DELETE -i 'http://localhost:7557/provisions/mynewprovision'
```

### 7.4.3 Get provisions

Get all provisions:

```
curl -X GET -i 'http://localhost:7557/provisions/'
```

# Part IV

# Security

CHAPTER

EIGHT

# HTTPS

TODO

CHAPTER

NINE

# CPE AUTHENTICATION

## 9.1 CPE to ACS

> **Note**
>
> By default GenieACS will accept any incoming connection via HTTP/HTTPS and respond to it.

The following parameters are used to set and get (password is redacted but can be set) the username/password used to authenticate against the ACS:

Username: `Device.ManagementServer.Username` or `InternetGatewayDevice.ManagementServer.Username`

Password: `Device.ManagementServer.Password` or `InternetGatewayDevice.ManagementServer.Password`

### 9.1.1 Enable CPE to ACS Authentication

CPE to ACS authentication can be configured in the web interface by using the *Config* option in the *Admin* tab.

Go to the *Admin* -> *Config* page and click on *New config* button at the bottom of the page. This will open pop-up which requires you to fill in a key and value. The key should be `cwmp.auth`. The value accepts a boolean. Setting the value to `true` makes it so that GenieACS accepts any incoming connection, setting it to `false` makes GenieACS deny all incoming connections. This can be further configured using the `AUTH()` and `EXT()` functions.

### 9.1.2 The `AUTH()` function

The `AUTH()` function accepts two parameters, username and password. It checks the given username and password with the incoming request to determine whether to return true or false.

Basic usage of the `AUTH()` function could be as follows:

```
AUTH("fixed-username", "fixed-password")
```

This will only accept incoming request who authenticate with “fixed-username” and “fixed-password”.

The various device parameters can be referenced from within the `cwmp.auth` expression. For example:

```
AUTH(Device.ManagementServer.Username, Device.ManagementServer.Password)
```

### 9.1.3 The `EXT()` function

The `EXT()` function makes it possible to call an *extension* script from the auth expression. This can be used to fetch the credentials from an external source:

```
AUTH(DeviceID.SerialNumber, EXT("authenticate", "getPassword", DeviceID.SerialNumber))
```

## 9.2 ACS to CPE

TODO

CHAPTER

TEN

# ROLES AND PERMISSIONS

TODO