61 1100 080

ASSA ABLOY HOSPITALITY WEB API

TABLE OF CONTENTS

1 SUMMARY

The *ASSA ABLOY Hospitality Web API* is an interface to DC-One and VisiOnline systems through a secure channel. Through this interface, it is possible to issue guest and staff cards, modify existing guest cards, create users and modify which group the user belongs to.

The interface is implemented as a web service, using the principles of REST and formatting requests and responses as JSON. The use of these standardized technologies makes it easy to understand and implement for third-party developers, and simplifies integration with existing network equipment.

The transport protocol is HTTPS (HTTP over SSL), which is a well-tested, secure solution and allows exchanging security credentials and cryptographic keys without additional security measures.

The functionality of the Web API is currently a subset of the *PMS Plus Protocol* [1], combined with new features not present in the old protocol. For example, it is not possible to issue staff cards, rent safes, or set rooms in Alarms Disabled Mode with the current implementation. Features of the Web API that are not present in the *PMS Plus Protocol*, apart from increased security, include monitoring of network nodes and obtaining the logical structure of the network.

2 TERMINOLOGY

Client - In the context of this API, a computer controlled by an entity that is external to the DC-One/VisiOnline system but has a network connection to the DC-One/VisiOnline server.

HTTP - Hypertext Transfer Protocol, an application protocol for distributed media.

HTTPS - Hypertext Transfer Protocol Secure, HTTP encrypted with SSL.

JSON - JavaScript Object Notation, a text-based standard for data representation.

REST - Representational State Transfer, an architectural style for distributed systems.

Serial number - In the context of smart cards, a unique number that was written to the card during manufacturing and that cannot be modified.

Server - In the context of this API, the computer that hosts the DC-One/VisiOnline production server software.

SSL - Secure Socket Layer, a cryptographic protocol.

3 SECURITY AND AUTHENTICATION

3.1 Encrypted communication channel

The communication between the client and the server relies on SSL in order to be protected against eavesdropping and manipulation by a third party. This allows the exchange of sensitive information, such as passwords and access keys, between the client and the server without additional considerations.

3.2 Initial authentication

Before a client may access most parts of this API on the server, it must be authenticated. The credentials consists of a username, a password, and in some cases a key card. With the proper credentials, the server agrees to create a *session* and returns a *session ID* and a *session access key*. Section 5.44 contains details about the authentication request and response.

A *session* has a limited life-time and the server may instruct the client to re-authenticate in order to establish a new session (see section 4.4). An instruction to re-authenticate happens as a response to a request, and the last request will not have been carried out in this case. The client will need to create a new session and then re-send the last request.

3.3 Authentication of requests

The *session access key* that was returned by the server is used by the client to create a signature for every request that it sends to the server. In this way, the client proves to the server that it has the correct key, without having to send the key with every request. By not sending the key repeatedly, it makes cryptographic attacks a lot harder. By including the current time in the signature, and by imposing a time-limit on the session, the server will also prevent old sessions from being replayed by an attacker.

The request signature is similar to the mechanism employed by Amazon S3, described in the AWS documentation [3]. HMAC-SHA1 is used for signing a combination of the request method, URL, and some specific request headers. The signature procedure is described in section 4.5.

4 PROTOCOL

The Web API has been designed according to the principles of *Representational State Transfer* (REST) and is implemented as a Web Service. The requests must be formatted according to HTTP/1.1 [2]. The HTTP methods GET, POST, PUT and DELETE are used to retrieve and modify the resources that are defined by the API, and the body of these HTTP requests are encoded in JSON format.

4.1 Operations

Resources can be divided into two types:

Basic resources - Entities that can be read or modified using a single request.

Collection resources - Entities that represents a collection of basic resources.

In principle, all resources could be accessed using the following actions:

- Read all properties of a resource using a GET request.
- Read all resources of a certain type by using a GET request on a collection resource.
- Create a new resource with a PUT request, given that the client supplies the ID of the new resource.
- Create a new resource with a POST request to a collection resource, allowing the server to assign the ID of the new resource.
- Replace all properties of an existing resource by using a PUT request.
- Modify one or more properties of an existing resource by using a POST request.
- Remove a resource using a DELETE request.

However, the API imposes various restrictions of certain resources. These restrictions may vary depending on the permissions associated with the authenticated user.

- Some properties of a resource are left out in a GET request.
- The user is not allowed to obtain the full list of resources within a collection.
- Some resources can not be created through the Web API.
- Some resources can not be created using a PUT request, since the system must assign the resource name.
- Some resources can not be replaced using a PUT request, since some properties are not allowed to be modified by the user.
- Some resources can not be removed by the user.

The complete set of valid operations for each type of resource is described in section 5.

4.2 URL structure

The resources are identified by URLs which are formed by combining transport protocol, server name, base URL of the web service, a version identifier, and the relative path of the resource. The transport protocol is HTTPS and the version identifier of this API is `v1`. If the server has the name `dc.example.com` and the web service has been configured to serve requests below the path `/api`, a card with *id* 135792 can be reached using the URL `https://dc.example.com/api/v1/cards/135792`.

A URL may also have a set of parameters, which then are supplied as standard query string, i.e. a list of `key=value` pairs separated by `&` and appended to the URL with a `?`, e.g. `https://dc.example.com/api/v1/cards?validTime=20130110T0900&cardHolder=jdoe`.

Throughout this document, these values will be used in examples, and it should be replaced by values derived from configuration of each particular facility.

The URL must be percent-encoded according to RFC 3986 [4]. The characters `/` and `?` must not be encoded when they separate different path segments. However, they must be encoded when they are used within a path segment, e.g. when they are part of the ID of a resource.

4.3 Request body

The body of all requests to the server and responses from the server will be in JavaScript Object Notation (JSON) and the HTTP header `Content-Type` must be set to `application/json charset=utf-8`.

When accessing a resource (both reading and modifying), the whole or part of the resource will be included in the response. The response will be either a JSON object, containing at least a member named *id*, or a list of objects of the same type, where all objects contains at least a member named *id*. E.g. when creating a new card resource, using a POST request to the `/api/v1/cards` resource, the system will assign an *id* to the card and return this in the JSON object representing the newly created resource.

4.3.1 Basic data types

The JSON objects may contain strings, integers and Boolean values (*true* or *false*). Values that hold the *id*:s of other resources are represented as strings. Date and time values are represented as strings.

4.3.2 Date and time representation

When dates and times are contained in a request, they must be formatted according to ISO 8601 and in most cases in local time, i.e. without time zone designators. Some exceptions exist, where the time instead must be represented in UTC. When the server returns dates and times, they will also be formatted according to ISO 8601 local time. The server will adjust some timestamps, e.g. by applying margins for expire times or rounding off seconds and minutes, and the time actually used by the server is then returned in the response.

4.3.3 Character encoding

The format of the request and response bodies are expected to be encoded in UTF-8. The JSON strings may contain any valid backslash escaping.

4.4 Status codes and error objects

Every request is acknowledged with a standard HTTP status code, indicating the overall result of the operation. The codes are used according to the following list.

200 OK - The request was successful.

201 Created - The request has been fulfilled and resulted in a new resource being created.

400 Bad Request - The request cannot be fulfilled due to bad syntax.

401 Unauthorized - The client has either not logged in, has logged in with bad credentials, or the session has expired.

403 Forbidden - The client has logged in correctly but has not permission to carry out this type of request.

404 Not Found - The requested resource could not be found.

405 Method Not Allowed - The request was made using a method not supported by that resource.

415 Unsupported Media Type - The request did not have content type of *application/json*.

In addition to this, an error code (4xx) will be coupled with a JSON object giving details about what went wrong. A typical client may in some cases present the **message** property to the operator. A client developer will find the property **developerMessage** of use in finding out what happened and what can be done to avoid this particular situation in the future. The rest of the properties may only be useful for logging and sending to technical support. This object may contain the following properties, although some properties can be left out.

status - Integer which is the same as the HTTP status code.

code - Integer identifying the situation in more detail.

resource - URL to the resource involved in the error.

properties - List of properties involved in the error.

message - Short description in the English language suitable to be presented to an operator.

developerMessage - Long technical description suitable to be logged and viewed by a client developer.

4.4.1 Detailed error codes

Below follows a list of error codes that can be returned in the **code** property of the error object. The first 3 digits of the error code are the same as the HTTP status code of the response.

40101 - The time skew between the client and the server is too big. Adjust the HTTP header `Date` or `X-Aah-Date` to match the server time.
40102 - A new session could not be created since the client has supplied bad credentials.
40103 - The supplied *session ID* does not exist. It may have expired.
40104 - The signature of the request did not match calculated signature.
40302 - Refused to overwrite a valid card. You must use `overwriteValidCard=true`.

4.5 Request signatures

This section may be easier to grasp by studying and running the reference implementation in section 6 and by reproducing the use cases 7.1.1, 7.1.2 and 7.1.3

Each request made to a resource other than those required for creating a session must be signed with the *session access key*. The signature is passed in the HTTP header field `Authorization` and has the form `AWS sessionId:signature`. Failure to provide a correct signature in the `Authorization` header field will result in **401 Unauthorized**, as described in section 4.4.

The first step when signing a request is compiling the string that should be signed. This string is built up from the following six elements:

1. HTTP verb
2. Content-MD5 header
3. Content-Type header
4. Date
5. Canonicalized headers
6. Canonicalized resource

The resulting string, encoded as UTF-8, is signed using HMAC-SHA1 with the *session access key* [5, 6]. The signature, encoded as Base64, is sent as the HTTP header `Authorization` on the form `AWS sessionId: signature` (note the space between `AWS` and session ID, but no spaces around the colon).

When the server receives the request, it will pick up its local copy of the access key associated with the session ID supplied by the client and generate the signature using the method described here. If this signature matches the one in the `Authorization` header, the server will process the request. If the server has no key associated with the session ID, it will return error code 40103. If the signatures don't match, it will return error code 40104.

4.5.1 HTTP verb

This is the HTTP method name, `GET`, `POST`, `PUT`, etc., followed by a line feed character.

4.5.2 Content-MD5 header

This is the value of the `Content-MD5` HTTP header field, which is the MD5 digest of the request body encoded as Base64. It is terminated by a line feed character. If the request has no `Content-MD5` header, this part will contain only the line feed character.

4.5.3 Content-Type header

This is the value of the `Content-Type` HTTP header field, which will be `application/json charset= utf-8` in a POST or PUT request. It is terminated by a line feed character. If the request has no `Content-Type` header (in e.g. a GET request), this part should contain only the line feed character.

4.5.4 Date

This is the value of the `Date` HTTP header field, which is the date and time when the request was sent, according to the client. It is terminated by a line feed character. The client must synchronize its time to the server, as described in section 4.5.7. If the header `X-Aah-Date` is present, this part should be empty except for the line feed character.

4.5.5 Canonicalized headers

If the HTTP header `X-Aah-Date` is present, it should be included after it has been converted using the following method:

1. Convert the HTTP header name to lower-case.
2. Trim any white-space around the colon following the header name.
3. Append a line feed character after the header value.

If there is no HTTP header `X-Aah-Date`, this part should be empty.

4.5.6 Canonicalized resource

This part starts with the path part of the URL, without decoding percent-encoded characters, up until but not including the query string. If the URL has a query string, the `?` is preserved at the beginning. The rest is divided at the `&` signs. These parts are sorted lexicographically and put back together again to form a canonicalized version of the same query string. This string is *not* terminated by a line feed character.

4.5.7 Client and server time in requests

The timestamp of the request need to be within 15 minutes of the server time when the request is received, or the server returns error 40101. Since this timestamp is included in the string that the client signs with the access key, it prevents requests that are older than 15 minutes from being replayed to the server.

When the client receives the error 40101, it can read the server's time in the HTTP header `Date` of the response. The client then needs to adjust the time it sends in its own `Date` header in order to convince the server that the request is valid. If the client software cannot control the value of the `Date` header, it can instead use the header `X-Aah-Date` with the same format. The `X-Aah-Date` always takes precedence over `Date`.

4.6 Future expansions

The protocol may receive updates in the form of new properties in either requests or responses of a resource. A client must not fail if it encounters unexpected properties in a response. Any new properties and query parameters introduced for a resource will not be mandatory, but will instead include reasonable default values, so the client can ignore the new features if desired.

If a property changes meaning or format in a way that would break the previously documented functionality, the base URL of the protocol will change from `/api/v1` to `/api/v2`. This allows unmodified clients to continue using the old functionality.

4.7 Performance

It is recommended to keep the HTTPS connection alive between requests in order to speed up the processing of subsequent requests.

5 RESOURCES

This section explains all the methods, properties and query parameters for all the resources exposed through this API. The resources are listed in alphabetical order.

All access to resources usually starts with the creation of a *session* resource (5.44). However, if an operator card is required to create a session, a list of encoders can be retrieved without using an `Authorization` header, by reading the *encoders* resource (5.27.1).

Once a session has been established, start by reading about the resource that is the most integral part of the required function. From there, other resources will be referred to in the text. The list below may serve as a starting point.

/alarms - Monitor alarms (5.3)

/allDoors - Get the information of all the doors in details (5.5)

/blockingSchedules - Get information for the blocking schedules (5.8)

/calendar - Get information for the calendar (5.12)

/cancelCards - Get information for all the cancelled cards (5.13)

/cardNames - Get information card names used in the system (5.15)

/cards - Issue and validate cards (5.16)

/commonDoorRelays - Get information for common door's relay (5.19)

/doors - Send commands to doors and read door information (5.22)

/doorsPassage - Get information for door passage (5.24)

/doorsSupplement - Get door suplplement information (5.26)

/eventFilters - Get information for event filters (5.30)

/eventNames - Get information for event names (5.32)

/lockFirmwares - Get information and download the lock firmwares (5.34)

/moduleFirmwares - Get information and download the firmwares other then lock (5.37)

/nodes - Monitor network nodes (5.38)

/notes - Get information about the notes (5.41)

/relayUnits - Get information for relay units (5.43)

/systemParameters - Read some system parameters (5.46)

/timeScheduleLines - Get information for time schedules (5.49)

/validGuestCards - Get information for valid guest cards (5.53)

5.1 /alarmTypes

5.1.1 Methods

GET - Read a list of all alarm types. Returns a list of objects containing **id** and **name**.

5.2 /alarmTypes/{id}

5.2.1 Methods

GET - Read information about an alarm type.

5.2.2 Properties

id - Unique ID of this alarm type.

name - Short description of the alarm type.

5.3 /alarms

Requires the option *Web service for checking and confirming alarms*.

5.3.1 Methods

GET - Search for alarms that have been triggered or completed by the system, or receive notification when a new alarm of a certain type occurs. Returns a list of objects where the **id** property refers to an alarm resource and the **updateTime** indicates the exact instance when the alarm was updated (see section 5.4).

5.3.2 Query parameters

alarmType - The type of alarms that should be returned in the list, formatted as a comma-separated list of alarm types. For information about alarm types, see section 5.1.

updatedSince - Only return alarms that were triggered or completed after this date and time. For continuous watching of alarms, in order to prevent receiving the same alarm twice, use the highest value of the **updateTime** property from the last search response. The time must be represented in UTC, in order to avoid glitches when the server changes to or from DST.

timeout - If no alarms match the search query, wait this number of seconds for new alarms to be updated. If omitted or zero, the result is returned immediately, even if the list is empty.

5.4 /alarms/{id}

5.4.1 Methods

GET - Read information about an alarm.

POST - Mark an alarm as completed, by setting the **completed** property to *true*.

5.4.2 Properties

id - Unique ID of this alarm.

alarmType - The ID of an *alarmTypes* resource indicating the type of alarm (see section 5.1).

description - Textual description of the alarm.

alarmTime - The date and time of when the alarm occurred.

card - The card that caused the alarm, or *null* if the alarm wasn't caused by a card.

cardHolder - The holder of the card that caused the alarm.

door - The door that caused the alarm, or *null* if the alarm wasn't caused by a door.

completed - Boolean that indicates if the alarm has been completed.

completionTime - The date and time of when the alarm was completed, or *null* if the alarm is hasn't been completed.

completionUser - The user that completed the alarm, or *null* if the alarm is hasn't been completed.

updateTime - The date and time of when the alarm was updated by the server, in UTC.

5.5 /allDoors

Note: this resource is restricted and can only be used from a service device with granted access.

5.5.1 Methods

GET - Get the list of all the doors with details.

5.5.2 Properties

See section (5.22.2) for details.

5.6 /blockingGroups

5.6.1 Methods

GET - Read a list of all blocking groups. Returns a list of objects where the **id** contains the name of the blocking group.

5.7 /blockingGroups/{id}

5.7.1 Methods

GET - Read all properties of a blocking group.

PUT - Create a new blocking group. This requires that the *Blocking groups* option has been enabled.

POST - Change the **id** of the blocking group. This requires that the *Blocking groups* option has been enabled.

5.7.2 Properties

id - Name of the resource used when referring to the blocking group from other resources.

5.8 /blockingSchedules

Note: this resource is restricted and can only be used from a service device with granted access.

5.8.1 Methods

GET - Get the list of all the blocking scedules.

5.8.2 Properties

doorID - The number of the door.

groupsToBlock - Binary value of groups to block.

timeScheduleID - The number of the time schedule.

5.9 /callback

Requires the option *Web service for callback*.

It is possible to set-up callbacks for specific resources and receive callbacks when those resources are changed in the system.

5.9.1 Methods

POST - Sets the properties for a new callback.

5.9.2 Properties

resources - an object that contains the resources that should be included in the callback.

events - resource the handles events in the system.

emiEvents - resource the handles EMI events in the system.

cards - resource that handles cards in the system.

fields - an array of field names that should be included in the callback.

eventCodes - an array of event codes that should be included in the callback (events only).

5.10 /callback{id}

Requires the option *Web service for callback*.

5.10.1 Methods

GET - Returns the queue of changes for the specified callback.

DELETE - Removes the specified callback.

5.10.2 Properties

id - Unique identifier of a previously set-up callback.

resources - an object that contains the resources queued up for the callback.

events - object that contains the queued up events.

emiEvents - object that contains the queued up events.

cards - object that contains the queued up cards.

5.11 /calendar

Note: this resource is restricted and can only be used from a service device with granted access.

5.11.1 Methods

GET - Get the list of all the items relating to calendar.

5.11.2 Methods

calendarDate - The date or start date.

everyYear - Boolean value that tells if this should occur every year.

holidayNumber - The number of the day connected to this record.

toDate - The end date.

5.12 /calendar/{calendar date}

5.12.1 Methods

GET - Get the information of calendar for given calendar date.

5.13 /cancelCards

Note: this resource is restricted and can only be used from a service device with granted access.

5.13.1 Methods

GET - Get the list of all the cancelled card in the system.

5.13.2 Properties

cancelled - Boolean value that tells if the card is cancelled.

expireTime - The expiration time of the card.

registrationNumber - The registration number of the card.

5.14 /cardNames

Note: this resource is restricted and can only be used from a service device with granted access.

5.14.1 Methods

GET - Get the list of all the card names.

5.14.2 Properties

cardName - The name of the card.

cardNameID - The number of the card name.

5.15 /cardNames/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.15.1 Methods

GET - Get the name of card from given card name ID.

5.16 /cards

Requires the option *Web service for guest cards*.

5.16.1 Methods

GET - Search for cards by specifying a search criterion using the query parameters listed in section 5.16.3.

The response has the form of a list of objects where the **id** property refers to a card resource (see section 5.17.2). If no cards match the search criterion, an empty list is returned.

POST - Create a new card or read a card from an encoder.

The minimal set of properties that need to be specified are **format**, **expireTime**, and **doorOperations**. If successful, the server creates a new card resource and assigns an **id** to it. The contents of the new resource is included in the response.

It is also possible to create a card by only specifying **format** and **cardHolder**. This requires that the *card holder* belongs to a *user group* which has *uniform access* enabled (see section 5.50.3) and that the *card holder* has **extendedExpireTime** set (see section 5.51.3).

Encode card on encoder - Set query parameter `action=encode`.

Encode loyalty card - Include the **serialNumbers** property in the body. The card with a matching serial number will be encoded at an auto-update encoder or at an online door that have received the card image.

Read card on encoder - Set query parameter `action=readCard`.

5.16.2 Properties

See section 5.17.2

5.16.3 Query parameters for GET

validTime - Matches cards where this time occurs between **startTime** and **expireTime**.

startTime - Matches cards that expire after this time. Must be used in conjunction with **expireTime**.

expireTime - Matches cards that become valid before this time. Must be used in conjunction with **startTime**.

cardHolder - Matches cards where **cardHolder** is equal to this value.

guestDoor - Matches cards that has this door listed in its **doorOperations** list, under **operation** `guest`.

Parameters for exactly one time match (either **validTime** or both of **startTime** and **expireTime**), and at least one of the other parameters must be specified. The conditions of all the parameters must be fulfilled for every card that is returned in the response.

5.16.4 Query parameters for POST

action - Set to `encode` to encode a card, or `readCard` to read the card on the encoder.

autoJoin - If set to `true`, this card will join with any overlapping cards. If set to `false` or omitted, the server will return an error if there exists any overlapping cards and the **joiners** property does not specify at least one of them. Cannot be used in combination with `override`.

autoUpdate - If set to `true` in combination with `action` being `readCard`, the card on the encoder will be updated, if it has any pending modifications stored on the server.

encoder - The encoder that shall be used when query parameter `action` is `encode` or `readCard`. Since the parameters are sent in the URL, any special characters such as whitespace or slash must be percent-encoded.

override - If set to `true`, this card may override any overlapping cards that are not specified as joiners. If set to `false` or omitted, the server will return an error if overlapping cards exist. Cannot be used in combination with `autoJoin`.

overwriteValidCard - If set to `true`, the encoder may write to a card even if it already contains valid access data. If set to `false` or omitted, the server will return error code 40302 if an already valid card is found on the encoder.

sendJoinersOnline - If set to `false`, any modifications that are carried out by the server as a result of `updateJoiners` being set to `true` will not be sent to the locks. Instead, they will only reach physical cards through an encoder performing auto-update. If set to `true` or omitted, the modifications will be sent to any guest room doors related to the modifications.

sendOnline - If set to `false`, the server will not send this modification to any locks. Instead, it will only reach a physical card through an encoder performing auto-update. If set to `true` or omitted, the modification will be sent to any guest room doors related to the modification, unless it was written directly to a card through an encoder in the same request.

updateJoiners - If set to `true`, the server will allow adding doors to a card with joiners. The card images of the joiners will be automatically updated so they are not overridden by the modified card. If set to `false` or omitted, the server will return an error if the client tries to add doors to a card with joiners.

5.17 /cards/{id}

A *card* resource represents the data that has been encoded or will be encoded to a card in the near future. For every new card that should be issued, a new card resource need to be created.

Each card resource contains a list of serial numbers of physical cards, which is empty at first. When the client requests that a specific *encoder* writes the contents of the card resource to a card, the serial number of the card at the encoder is stored in this list. The client may also supply serial numbers when creating or modifying the card resource. In this case, a card with the proper serial number will be encoded automatically when presented to an *auto-update station* or at one of its associated guest room doors. However, while the auto-update station will always have access to the latest card modifications, a door is required to be online in order to receive information about the cards it shall encode.

5.17.1 Methods

GET - Read all properties of a card.

POST - Modify some of the properties of a card.

5.17.2 Properties

id - Registration number of this card, assigned by the system.

cardHolder - The user that this card has been or will be issued to. This must be a valid resource in the *users* collection (see section 5.51). It defaults to the *guest* user if none is specified when creating the resource. The user may cause **expireTime** and **doorOperations** to be implicitly set (see sections 5.51.3 and 5.50.3). The **cardHolder** can only be set at creation and can never be changed afterwards.

created - The date and time of when this resource was created, or when a property last was modified that affected the information that needed to be encoded on the card.

startTime - The date and time of when the card becomes valid. Defaults to the current time.

expireTime - The date and time of when the card becomes invalid. If **cardHolder** refers to a user with **extendedExpireTime**, it cannot be set longer than this, and it will be automatically increased by the system (see section 5.51.3).

cancelled - Boolean value telling if the card has been cancelled. Defaults to false and can only be changed from false to true, never from true to false.

format - Format of the card. It must be one of the strings listed in section 5.17.3 and is mandatory when creating a card.

joiners - List of cards that this card shall share door access with instead of overriding it, provided that the cards share at least one door. This property is only included in the response to a request if the `fields` parameter is set to `joiners`.

doorOperations - List of objects specifying which doors the card has access. Each object in the list has the following properties:

- **operation** - Door operation for the doors in this object. It must be set to `guest` for guest rooms and `normal` for doors to common rooms.
- **doors** - List of doors that should be opened with the specified operation. The first door in the list of the first object with **operation** `guest` has the special role of being the main door of the card.
- **doorGroups** - List of door groups that should be opened with the specified operation. Use the *doors* resource (see section 5.22) to find out which doors belong to a specific door group.

numberOfIssuedCards - Integer specifying how many cards that has been issued based on this resource. Starts at zero and is automatically incremented by the system.

serialNumbers - List of serial numbers of all issued cards based on this resource. Serial numbers are automatically appended to the list by the system when cards are written to by an encoder. If the cards are of a type without serial number, a value "0" is appended for each card. If serial numbers are supplied by the client, a card can be encoded at the guest doors of this card or at an auto-update station.

sectors - List of object describing the raw sector data of a card when query parameter `action` is `readCard`. Each object in the request must contain the properties **index**, **key** and **keyType**, and the response will contain **index** and **contents**. The following properties are defined for each item of the list.

- **index** - Integer identifying the sector of the card. The first sector has index 0.
- **key** - Hexadecimal 12-character string specifying the 6-byte key that is required in order to read the contents of the sector.
- **keyType** - Type of the key, either `A` or `B`.

contents - Hexadecimal string holding the full contents of the sector. The size of the sector is given by the length of of the string.

pendingAutoUpdate - Boolean value telling if the card has any pending modifications that will be written to the card either at a lock that has received the modification or at an encoder performing auto-update.

timeSchedule - Assign a Time schedule to the card so that the user can be granted access at certain days and between certain hours. The time schedule has to be exists in the Database otherwise web service will throw an exception.

5.17.3 Card formats

The valid values of the **format** property are `rfid48` and `TLCode` only.

5.17.4 Encoding and reading card on encoder

Instead of supplying a **serialNumber**, the client can encode the card directly by specifying an encoder. A card on an encoder can also be read directly. When reading a card, the request body will be empty, and the response will contain a full card resource. If the card on the encoder has not been encoded by this system, all properties of the card resources are left out, except the **serialNumbers** property. Reading a serial number from a card does not guarantee that the card can be used by the system, only that it is a card with serial number.

5.18 /commonDoorRelays

Note: this resource is restricted and can only be used from a service device with granted access.

5.18.1 Methods

GET - Get the list of common door relays.

5.18.2 Properties

commonID - The number of the common door.

doorID - The number of the relay.

5.19 /commonDoorRelays/{door id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.19.1 Methods

GET - Get the common door relay information for given door id.

5.20 /counters

5.20.1 Methods

GET - Reads a list of all the stored counters in the database.

5.20.2 Properties

counterName - Name of the counter.

defaultCount - The default counter value.

description - A textual description of the counter.

doors - The doors connected to the counter.

maxCount - The maximum counter value.

timeSchedule - The name of the connected time schedule (if applicable).

resetTime - The time of day when the counter resets.

5.21 /doors

5.21.1 Methods

GET - Return a list of doors in the database.

or

GET - Search for cards by specifying a search criterion using the query parameters listed in section 5.21.2.

5.21.2 Query parameters

doorGroup - Return only doors that belong to this door group.

5.22 /doors/{id}

5.22.1 Methods

GET - Read all normal properties of a door (see section 5.22.2)

POST - Send a command to a door, by specifying exactly one of the command properties (see section 5.22.4) and query parameters (see section 5.22.5).

5.22.2 Normal properties

id - Name of the door. This value is of *string* type, even if guest room doors often are numerical by nature.

allowOnOff - Boolean value for allowing on/off opening of the room.

autoLock - Boolean value that tells if the lock should enter locked state when the door frame switch is closed.

doorCategory - The conceptual type of door (guest, entrance, elevator reader, etc...).

doorGroup - Name of the door group that this door belongs to. Each door belongs to exactly one door group.

doorGroupID - The number of the door group.

doorID - The real number of the door.

doorName - The name of the door (same as id).

doorType - The type of lock hardware.

escapeReturnMode - The manual locking mode.

exitButton - Exit button mode.

exitButtonOpenTime - Exit button open time.

externalRelayMode - Boolean value that tells if internal or external relays should be used.

localName - The name of the door adjusted for string-sorting.

online - Boolean value that tells if the door is online.

openTime - The open time (in seconds) of the door.

roomIntervalStart - The start value of the entrance interval.

roomIntervalStop - The end value of the entrance interval.

5.22.3 Door states

A door can be placed in one of the 4 states below, using remote commands, function cards or service tools.

Normal - The door can be opened with a valid keycard or with a remote open command, but is otherwise locked.

Passage - The door is unlocked some periods of the day (or all day) according to a passage time schedule.

Stand open - The door is unlocked for a specified time period, ignoring any passage time schedules during this time.

Emergency open - The door is unlocked unconditionally, ignoring any passage time schedules or if the stand open time expires.

5.22.4 Command properties

Requires the option *Web service for sending open/close commands to doors*.

A *door* resource can also be used to send commands to locks that can be reached by the server, either through ZigBee or Ethernet. For this purpose, a door has a set of *command properties*, used for sending direct commands to the lock. These properties are write-only and only one can be present in each request.

Writing to a command property is done by sending a POST request. The server will immediately try to reach the specified device and it will not send a response until the command has reached its target, or when the command has timed out and the server has stopped trying. This means that it may take up to 20 seconds to complete a successful command to ZigBee lock.

The **checkedOut** property is exceptional, in that it is immediately acknowledged or rejected. The server will then carry out the command in the background and the client will not know when it has been completed.

The command properties are listed below.

open - Set to *true* to send an opening command to the door. The door is locked again automatically after a certain time. Setting this property to *false* is not allowed.

revokePassageTime - A door with a passage time schedule will be deactivate passage mode until this time occurs. The string `forever` means it will remain inactive until it is modified. *null* means that the passage time schedule is reactivated immediately.

standOpenTime - The door will remain open until this time occurs. The string `forever` means that the door will be kept open until this property is set to another value. *null* means that the door is not kept open by this method.

emergencyOpen - Set to *true* to put the door in *Emergency Open* mode, causing it to remain unlocked regardless of other activities. Set to *false* to return the door to normal mode.

checkedOut - Set to *true* to send a check out command to the door. Setting this property to *false* is not allowed.

checkoutTime - It is an optional property to send the checkout time together with checkout command(please refer parameter **checkedOut** for checkout command). If this property is set, then checkout command is not sent to the door until specified time has already passed. In case of mobile keys, the key is not removed from the phone until checkout time has passed.

doors - This property provides list of doors to be checked out. Checkout command for a single door can also be sent by mentioning the door in the request header, so this property is not required in that case, List of doors can be used only with **checkedOut** command. Refer user case section 'Sending checkout commands' for different use cases.

5.22.5 Query parameters

timeout - The number of seconds that the server should try to reach the door when writing to a command property. If omitted or zero, the server will not be able to send the command to ZigBee doors.

5.23 /doorsPassage

Note: this resource is restricted and can only be used from a service device with granted access.

5.23.1 Methods

GET - Get the list of all the door passages.

5.23.2 Properties

doorID - The number of the door.

timeScheduleID - The number of the time schedule.

5.24 /doorsPassage/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.24.1 Methods

GET - Get the information of door passage for given door ID.

5.25 /doorsSupplement

Note: this resource is restricted and can only be used from a service device with granted access.

5.25.1 Methods

GET - Get the list of all the doors that have supplement data.

5.25.2 Properties

doorID - The number of the door.

5.26 /doorsSupplement/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.26.1 Methods

GET - Get the supplement data for given door ID.

5.26.2 Properties

`IA_Control` - Controls how the alarm should be handled.

`IA_Output` - Controls the relay in ”Intruder alarm” mode.

Prealarm - The pre-alarm mode.

`T_Alarmbypass` - Time in seconds the intruder alarm relay should be active.

`T_DA_Pulse` - Time in 1/2 seconds the duress alarm is active.

`T_IA_Pulse` - Time in 1/2 seconds the intruder alarm relay should be active when alarm is turned on or off.

`T_Indication_IA_Set` - Time in seconds the indicator for alarm set shall be active.

`T_Indication_IA_Unset` - Time in seconds the indicator for alarm clear shall be active.

`T_Indication_Locked` - Time in seconds the indicator for locked state shall be active.

`T_Indication_Unlocked` - Time in seconds the indicator for unlocked state shall be active.

`T_Prealarm` - Grace time in seconds before pre-alarm is triggered.

doorID - The number of the door.

5.27 /encoders

5.27.1 Methods

GET - Read the names of all encoders. Returns a list of objects where the **id** property contains the name of the encoder. These value can be used as the **encoder** property when creating a *session* (see section 5.44.2) or with *cards* resources as the value of the `encoder` query parameter (see section 5.16.4). This method can be performed without an `Authorization` header.

5.28 /encoders/{id}

5.28.1 Methods

POST - Abort an encoder operation.

5.28.2 Query parameters

action - Set to `abort` in order to abort an ongoing encoding operation.

5.29 /eventFilters

Note: this resource is restricted and can only be used from a service device with granted access.

5.29.1 Methods

GET - Read the list of all the event filters.

5.29.2 Properties

enabled - Boolean value if the event shall be enabled or not.

eventSubGroupId - The number of the event sub group.

5.30 /eventFilters/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.30.1 Methods

GET - Check if given sub group is enabled or not.

Note - Value of event sub group can be only from 0 to 127.

5.31 /eventNames

Note: this resource is restricted and can only be used from a service device with granted access.

5.31.1 Methods

GET - Get the list of all the event names in the Database.

5.31.2 Properties

eventCode - The number of the event.

eventName - The textual description of the event.

5.32 /eventNames/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.32.1 Methods

GET - Get the description for the given event code.

5.33 /lockFirmwares

Note: this resource is restricted and can only be used from a service device with granted access.

5.33.1 Methods

GET - Get the list of all the lock firmwares in the database.

5.33.2 Properties

NGDUHeadID - The unique number associated with a specific firmware version.

NGDUVersion - The version string of the firmware.

buildTime - The time and date the firmware was built.

fwBCC - The checksum of the firmware data.

hwType - The hardware the firmware was meant for.

noteID - The id of the description.

numPackets - The total number of parts the firmware is split into.

5.34 /lockFirmwares/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.34.1 Methods

GET - Get the specified lock firmware for given NGDUHeadID.

5.34.2 Properties

NGDUCode - Data blob in asciihex.

NGDUHeadID - The unique number associated with a specific firmware version.

NGDUPackageNo - The sequence number of the part. The complete firmware should be concatenated in sequence.

5.35 /mobileaccess

5.35.1 Methods

GET - Reads a list of all credential services accounts stored in the database.

5.35.2 Properties

name - Name of the account.

description - A textual description of the account.

5.36 /moduleFirmwares

Note: this resource is restricted and can only be used from a service device with granted access.

5.36.1 Methods

GET - Get the list of all the module firmwares in the database.

5.36.2 Properties

TMFHeadID - The unique number associated with a specific firmware version.

buildTime - The time and date the firmware was built.

fwBCC - The checksum of the firmware data.

fwVersion - The version string of the firmware.

hwType - The hardware the firmware was meant for.

noteID - The id of the description.

numPackets - The total number of parts the firmware is split into.

5.37 /moduleFirmwares/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.37.1 Methods

GET - Get the specified module firmware for given TMFHeadID.

5.37.2 Properties

TMFCode - Data blob in asciihex.

TMFHeadID - The unique number associated with a specific firmware version.

TMFPackageNo - The sequence number of the part. The complete firmware should be concatenated in sequence.

5.38 /nodes

Requires the option *Web service for reading network nodes*.

5.38.1 Methods

GET - Read information about all nodes in the network with a single request. Returns a list of objects, where each object contains the properties listed in section 5.39.2. The nodes are returned in a depth-first pre-order traversal, starting with the `ROOT` node, followed by one of its children, then by one of its grandchildren (if any), etc.

5.39 /nodes/{id}

A *node* resource represents the network part of a device that is connected to the server, either through ZigBee or Ethernet. Some nodes, such as the *ZigBee End node* and the *ZigBee Router*, may be associated with a *door* or a *thermostat*. In these cases, the associated functionality is identified by the property **function**. Other nodes, such as the *ZigBee Gateway* and the *TCP/IP Concentrator*, are fully defined by this resource, since their function is entirely related to the network. These kinds of nodes will have **function** set to *null*.

Nodes without Ethernet connectivity will have the properties **macAddress** and **ipAddress** set to *null*. Nodes without ZigBee connectivity will have **ieeeAddress** set to *null* and **panId**, **networkAddress** and **channel** set to -1.

5.39.1 Methods

GET - Read all properties of a node.

5.39.2 Properties

id - Name of the resource used when referring to the node from other resources. The server has the **id** ROOT and all other nodes can be found by inspecting the **children** property of the ROOT node or any of its descendants.

children - List of child nodes of this node.

role - String that informs about the logical location of this node within the network. The valid roles are listed in section 5.39.3.

name - A name that is either automatically assigned by the system, or an arbitrary name assigned by an operator.

macAddress - MAC address in hexadecimal, separated by colons.

ipAddress - IP address as dotted decimal.

ieeeAddress - IEEE address in hexadecimal, separated by colons.

panId - Integer identifying which PAN the node is on.

networkAddress - Integer identifying the node within its PAN.

channel - Integer representing which ZigBee channel the node is using.

firmwareVersion - String containing firmware version(s) of this node.

numberOfBufferedCommands - Integer telling the number of buffered commands that are waiting to be sent to this device.

latestReceiveTime - Time of when the latest message was received from the node.

latestLinkQuality - Integer in the range [0,100] representing the link quality reported in the latest message received from the node.

latestLinkQualityTime - Time of when the latest link quality value was received.

averageLinkQuality - Integer in the range [0,100] representing a running average of the latest link quality values reported by the node.

function - Object describing the function of this node, e.g. if it is connected to a door or a thermostat. The format of the function object is described in section 5.39.4. If this node has no function other than forwarding network traffic, this property will be *null*.

5.39.3 Roles

These are the valid values for the **role** property of a node.

ROOT - The production server

ZGW - ZigBee Gateway

RT - ZigBee Router

EN - ZigBee End node

CGW - TCP/IP Concentrator

ELI - Ethernet Lock Interface

RS-485 GW - RS-485 Gateway

5.39.4 Function object

The **function** property of a node is either an object with the following properties, or the value *null*.

type - `door`, `thermostat`, `thermostatController`, `doorSwitch`, `motionDetector`, `minibar` or `safe`.

id - Unique name or ID of the functional entity, e.g. the room name or motion sensor address.

subtype - If **type** is `doorSwitch`, this is one of `window`, `internal`, `external`. Otherwise, it is *null*.

5.40 /notes

Note: this resource is restricted and can only be used from a service device with granted access.

5.40.1 Methods

GET - Get the list of all the notes (description) in the database.

5.40.2 Properties

noteID - The number of the note.

5.41 /notes/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.41.1 Methods

GET - Get the note text for given note id.

5.41.2 Properties

noteText - The description of the note.

5.42 /relayUnits

Note: this resource is restricted and can only be used from a service device with granted access.

5.42.1 Methods

GET - Get the list of all the relay units in the database.

5.42.2 Properties

doorID - The number of the relay.

relayNum - The index of the relay.

doorID - The number of the elevator reader.

5.43 /relayUnits/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.43.1 Methods

GET - Get the relay unit information for given relay unit id.

5.44 /sessions

5.44.1 Methods

POST - Creates a session for a client, enabling it to access other resources. The client must supply a **username** and a **password**. Some systems are configured to require reading of an operator card as part of the authentication. In these cases, an **encoder** must also be supplied. The response will contain an **id** and an **accessKey**, which the client uses for signing each request and supplying it in an HTTP header, according to section 4.5.

Since the *sessions* resource is needed for obtaining an *access key*, this is the only resource that can be accessed without signing the request.

If the password or the card read by the specified encoder does not match the user credentials stored in the system, the server will return error code 40102.

A service device can request extended access by passing **email** and **uuid** in the request. The result will then contain a valid **serviceDeviceID**.

5.44.2 Properties

id - The ID that should be used together with the signature in the `Authorization` header.

username - The *id* of a *user* that is allowed to log in through the Web API.

password - The password of that user. This is never sent in a response object.

encoder - The encoder where the operator will put the operator card that belongs to the specified user. How to retrieve the list of encoders is explained in section 5.27.1.

email - If an e-mail is passed then the e-mail will be checked against the stored e-mails in the database.

uuid - This is the unique ID of the service device and should be passed along with the **email** property.

accessKey - If the session was created successfully, this key can be used to sign requests being sent to other resources.

serviceDeviceID - If the passed **email** and **uuid** is OK the returned serviceDeviceID should be used as registration number in the communication layer with the lock.

5.45 /sessions/{id}

5.45.1 Methods

DELETE - Terminate the current session. A session will also be terminated automatically after a period of inactivity or when another session is created by the same operator.

5.46 /systemParameters{parameter name}

Note: this resource is restricted and can only be used from a service device with granted access.

5.46.1 Methods

GET - Get the value of the specified parameter.

The server will read the specified parameter from the database and return the value in the response. Following is the list of parameters which can be accessed through this resource.

- ONLINE_AJARMAX_INTERNAL

- ONLINE_AJARMAX_GUEST
- ONLINE_AJARMAX_STAFF
- ONLINE_ENABLE
- NFC_BLE_UUIDCUSTOMERSPECIFIC
- ESCAPERETURN
- OPENMODE
- SYSTEM_DIVERSIFIEDKEYA
- SYSTEM_DONOTSAVEACCESSEVENTSWITHOUTPIN
- SYSTEM_ENABLEBLOCKINGSCHEDULES
- SYSTEM_ICLASSPAGES
- SYSTEM_ID
- SYSTEM_ID_OLD

5.47 /timeSchedules

5.47.1 Methods

GET - Reads a list of all time schedules in the database.

5.47.2 Properties

timeSchedule - The name of the time schedule.

description - The description of the time schedule.

5.48 /timeScheduleLines

Note: this resource is restricted and can only be used from a service device with granted access.

5.48.1 Methods

GET - Get the list of all the time schedules in the database.

5.48.2 Properties

timeScheduleID - The number of the time schedule.

5.49 /timeScheduleLines/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.49.1 Methods

GET - Get the detailed information of the specified time schedule ID.

5.49.2 Properties

blockExitButton - Boolean value if exit button should be enabled or not.

blockGroups - Boolean value if groups should be blocked or not.

dayBits - Bit field describing which days this should affect.

passageType - The type of passage.

standOpen - Boolean value if stand open should be active or not.

startTime - The start time in TLM-format.

stopTime - The end time in TLM-format.

timeScheduleID - The number of the time schedule.

5.50 /userGroups/{id}

All users must belong to a *user group*. The user group can contain a set of doors that can be of use when creating cards. Groups that have the property **uniformAccess** set to *true* will force its set of doors upon cards linking to that group.

5.50.1 Methods

GET - Read all properties of a user group.

PUT - Create a new user group. If the *Blocking groups* option is disabled, a blocking group with the same **id** is automatically created.

POST - Modify some of the properties of a user group.

5.50.2 Properties

id - Name of the resource used when referring to the user group from other resources.

doorOperations - List of objects specifying doors that can be useful when creating cards. The format is the same as the **doorOperations** property listed in section 7.6.2.

uniformAccess - Boolean that specifies that the doors listed in **doorOperations** are the *only* doors that a card holder belonging to this group can have access to (see section 5.50.3 for details).

expireTime - Replaces the **expireTime** of a card when **uniformAccess** is *true*.

blockingGroup - Blocking group that is used to temporarily block access to a group of cards for certain doors. If the *Blocking groups* option is disabled, this property must be either omitted or set to the **id** of the user group. If the *Blocking groups* option is enabled, this property is mandatory and must identify an existing blocking group (see section 5.6).

autoBlockInRentedRoom - With the Auto-blocking option, it is possible to automatically block chosen user groups (or if applicable, blocking groups) from all rented online rooms. An example is to autoblock the group **Carpenters** to avoid renovation in rooms where guests are staying.

defaultReportTime - This is the default time that will be used when setting up reports in User Notifications.

ignoreDeadbolt - This option allows the user to enter rooms that are deadbolted.

ignorePrivacy - If this property is true for a user group (e.g. managers), users of that user group can have access to the door even when the lock is set in privacy.

maxStaffCardsPerUser - Maximum number of cards that can be issued to a user of this group.

syncWithActiveDirectory - Synchronize the user groups with Active Directory.

timeSchedule - user group can be granted access at certain days and between certain hours provided in the time schedule.

validation - Validation of cards means that to validate cards is that the card holder enters a PIN code e.g. once a day at a certain validation lock, such as the main entrance. When the card has been validated, it will be able to access all concerned doors according to the access pattern that is set up for the user. When the validation time has passed, the card must be validated again in any auto-update encoder.

validationTime - When the validation time has passed, the card must be validated again.

5.50.3 Doors and user groups

If a user group has the property **uniformAccess** set to *false*, then the the **doorOperations** property is just a passive list of doors. A client can read the **doorOperations** from any user groups and add them, whole or in part, to the **doorOperations** property when modifying either a card resource or another user group resource.

However, if a user group has **uniformAccess** set to *true*, **doorOperations** has a more direct meaning. When creating a card with a **cardHolder** that has its **userGroup** pointing to a group with *uniform access*, the client may not supply **doorOperations**. Instead it is taken from the user group.

If **doorOperations** of a user group resource with *uniform access* is modified, these changes will propagate to all valid cards belonging to **cardHolders** within this group. The guest rooms of these cards will receive auto-update images so that the cards eventually will receive the modified access data.

5.51 /users/{id}

A *user* supplies information about a person or type of person that can hold a card. Each user must have a unique **id** which is assigned by the client upon creation using a PUT request. A user may also have an **extendedExpireTime**, which will be used by the system to extend the **expireTime** of cards belonging to this user. The complete set of properties is shown below.

5.51.1 Methods

GET - Read all properties of a user.

PUT - Create a new user.

POST - Modify some of the properties of a user.

If a user is modified to have a **userGroup** that has *uniform access*, this will trigger auto-update of valid cards associated with the user. Changing the **userGroup** to one without *uniform access* will not trigger any card updates.

5.51.2 Properties

id - Name of the resource used when referring to the user from other resources.

surname - Surname of the user.

givenName - Given name of the user.

userGroup - User group that this user belongs to. It must refer to an existing resource in the *user groups* collection (see section 5.50). It is mandatory when creating the resource.

extendedExpireTime - The maximum **expireTime** for cards that have this user listed as **cardHolder**. The system will automatically increase the **expireTime** of the card until it can finally be set to **extendedExpireTime**. It defaults to *null* when creating the resource (see section 5.51.3).

autoDelete - Boolean value telling the system that this user can be deleted by housekeeping after the last card referring to it has become invalid. It defaults to *true* when creating the resource.

userDetails1 - Arbitrary string of information about the user.

userDetails2 - Arbitrary string of information about the user.

userDetails3 - Arbitrary string of information about the user.

userDetails4 - Arbitrary string of information about the user.

cellPhone - cell phone of the user.

defaultReportTime - This is the default time that will be used when setting up reports in User Notifications.

email - E-mail of the user.

pin - If the user should have access to any keypad units, enter a PIN code for accessing the keypad units.

confirmPin - Duplicate entry same as PIN for confirmation to check that entered pin is correct.

5.51.3 Extended expire time

Requires the option *Auto-extension of guest cards*.

The **expireTime** of a card is typically limited to one year into the future. If a user has the **extendedExpireTime** property set, the system will periodically increase **expireTime** for cards where this user is set as **cardHolder**, until it finally reaches **extendedExpireTime**. The new **expireTime** will reach the card through auto-update at the guest room door.

5.52 /validGuestCards

Note: this resource is restricted and can only be used from a service device with granted access.

5.52.1 Methods

GET - Get the list of all the valid guest cards.

5.52.2 Properties

registrationNumber - The registration number of the valid guest card.

5.53 /validGuestCards/{id}

Note: this resource is restricted and can only be used from a service device with granted access.

5.53.1 Methods

GET - Get the detailed information of the valid guest cards for given registration number.

5.53.2 Properties

doorID - The number of the door.

expireTime - The expiration time of the card.

lockStatus - The last known status of the card.

registrationNumber - The registration number of the valid guest card.

6 REFERENCE IMPLEMENTATION

This section contains the source code of a reference implementation in Python [7] of a client using this Web API. It has been tested in Python 2.7.3, but should work in any version from 2.5 up to 2.7.x, but not in Python 3.x.

When the `AAHWebService` object is created, the caller specifies the URL of web service and the credentials of the operator. The constructor, `__init__()`, stores these in the object for use by the other methods.

The only public method of the class is `send_request()`, and it is used to send a request of any type. The parameter `body` is typically included when `method` is `'POST'` or `'PUT'` and omitted for `'GET'` and `'DELETE'`. The method will call `_create_session()` when the client needs to log on, and it will call `_send_single_request()` to carry out the actual request. If the server reports that the time skew between client and server is too large, `send_request()` will apply an offset to the date in the subsequent requests and try again.

The method `_create_session()` creates a new session on the server by sending the credentials that were earlier supplied to the constructor. If successful, it will store the ID and the access key for use when signing other requests.

The method `_send_single_request()` starts by encoding the `body` parameter as JSON and computing its MD5 digest, given that the parameter was specified. The signature is then constructed using the method described in section 4.5 and stored in the `Authorization` header. The request is sent to the server and the response is read. The response is assumed to be in JSON format, and if this is not the case, an exception is raised.

The final part of the implementation contains an example of how to create an instance of the `AAHWebService` class and perform some simple requests.

6.1 AAHWebService.py

```
import base64, hashlib, hmac, json, httplib, socket, re, email
from time import strptime,mktime,timezone,time,gmtime,strftime,daylight


class AAHWebService(object):

    def __init__(self, baseUrl, username, password, debug_level=0):
        """Constructor of AAHWebService object"""

        # Use SSL if baseUrl starts with 'https://'.
        # Split the rest of baseURL at the first '/' into host and basePath.
        m=re.match('(https?)://(.+?)(/.*)',baseUrl)
        self._use_ssl=(m.group(1)=='https')
        self._host=m.group(2)
        self._basepath=m.group(3)


        # Store credentials in an object ready to be sent to the server.
        self._credentials={'username':username,'password':password}

        self._debug_level=debug_level # Get text output from HTTPConnection
        self._session={} # Placeholder for session ID and access key
        self._time_skew=0 # Assume that server and client times match


    def send_request(self, method, url, body=None):
        """Send a request to the web service.
        Automatically handle session creation and time skew.
        """

        # Send the request at most 3 times: The first request may need
        # a new time skew, the second request may hit an expired session,
        # the third request is successful or has a permanent error.
        for _ in range(3):
            try:
                # If we are already logged in, send the request. Otherwise,
```

```
                # create the session and then send the request in the next
                # loop iteration.
                if 'accessKey' in self._session:
                    (status,resource,server_time)=\
                        self._send_single_request(method,url,body)
                    request_was_sent=True
                else:
                    (status,resource,server_time)=\
                        self._create_session()
                    request_was_sent=False

                # Handle time skew (40101) and expired session (40103).
                # If the request was sent and everything went fine, or if any
                # other type of error occurred, we leave that to the caller.
                if status==401 and resource['code']==40101:
                    _=strptime(server_time,'%a, %d %b %Y %H:%M:%S GMT')
                    self._time_skew=mktime(_)-timezone-time()+daylight*3600
                elif status==401 and resource['code']==40103:
                    del self._session['accessKey']
                elif request_was_sent:
                    return (status,resource)

            except (IOError,ValueError) as e:
                # Handle connection refused, nameserver lookup failure,
                # sslv3 alert handshake failure or server response that was
                # not in JSON format.
                # Set a temporary error code and try again.
                (status,resource)=(500,{'message':str(e).decode('utf-8')})
                if self._debug_level:
                    print 'error: %s'%resource['message']

        # If the server still returns an error after 3 attempts, nothing
        # else can be done except for returning the error to the caller.
        return (status,resource)

    def _create_session(self):
        """Send user credentials and record the ID and the access key."""
        (status,resource,server_time)=\
            self._send_single_request('POST','/sessions',\
                                      self._credentials, False)
        if status==201:
            self._session=resource # Success! Store ID and key.
        return (status,resource,server_time)


    def _send_single_request(self, method, url, body, sign=True):
        """Send a properly signed request to the web service."""

        url=self._basepath+url
        headers={}

        if body:
            # Encode request body as JSON
```

```
        body=json.dumps(body)
        headers['Content-Type']='application/json;charset=utf-8'

        # Compute MD5 digest
        h=hashlib.new('md5')
        h.update(body)
        headers['Content-MD5']=base64.b64encode(h.digest())

    # Format the date correctly, after applying the client/server skew
    headers['Date']=email.Utils.formatdate(time()+self._time_skew)

    if sign and 'id' in self._session:
        # Canonicalize the URL
        (canonicalized_resource,q,query_string)=url.partition('?')
        canonicalized_resource+=q+'&'.join(sorted(query_string.split('&')))

        # Build the string to be signed
        string_to_sign=method+"\n"
        if 'Content-MD5' in headers:
            string_to_sign+=headers['Content-MD5']
        string_to_sign+="\n"
        if 'Content-Type' in headers:
            string_to_sign+=headers['Content-Type']
        string_to_sign+="\n"+headers['Date']+"\n"+canonicalized_resource

        # Create the signature
        h=hmac.new(self._session['accessKey'].encode('utf-8'),\
                        string_to_sign.encode('utf-8'),hashlib.sha1)
        headers['Authorization']='AWS %s:%s'\
            %(self._session['id'],base64.b64encode(h.digest()))




    # Communicate with server
    if self._use_ssl:
        conn=httplib.HTTPSConnection(self._host)
    else:
        conn=httplib.HTTPConnection(self._host)
    conn.set_debuglevel(self._debug_level)
    conn.request(method,url,body,headers)
    response_obj=conn.getresponse()

    # Interpret response
    server_time=response_obj.getheader('Date')
    response_body=response_obj.read()
    if self._debug_level:
        print "reply body: "+response_body
    try:
        resource=json.loads(response_body)
    except ValueError as e:
        resource={'message':e.message+"\nServer response:\n"+response_body}
```

```
        return (response_obj.status,resource,server_time)


# Example of how the AAHWebService class can be used
if __name__=="__main__":
    ws=AAHWebService('https://dc.example.com:443/api/v1',\
                         'cardAdministrator01','secret',1)

    (status,resource)=ws.send_request("GET","/users/jdoe")
    print json.dumps(resource,indent=4,sort_keys=True)
    assert(status==200)

    (status,resource)=ws.send_request("POST","/users/jdoe",\
                                          {'givenName':'Jane'})
    print json.dumps(resource,indent=4,sort_keys=True)
    assert(status==200)
```

7 USE CASES

7.1 Session management and request signatures

These use cases show how the *sessions* resource described section 5.44 resource is used and apply the algorithm outlined in section 4.5.

7.1.1 Create a session

Prerequisites: An operator with permission to use the Web API.

A session is created in order to obtain a *session ID* and a *session access key*. The client sends a POST request with a set of credentials.

```
POST /api/v1/sessions
{
    "username": "cardAdministrator01",
    "password": "secret"
}
```

The server verifies that the credentials are correct. Then it creates a session, assigns an ID and an access key to it, and returns these in the response.

```
201 Created
Location: https://dc.example.com/api/v1/sessions/342ba291
{
    "id": "342ba291",
    "accessKey": "AQIDBAUGBwg="
}
```

The client can now use the ID and the access key to sign requests, as shown in use cases 7.1.2 and 7.1.3.

7.1.2 Sign a request without a body

Prerequisites: Use case 7.1.1.

This use case describes how to sign the following GET request.

```
GET /api/v1/cards?validTime=20130105T1200&cardHolder=jdoe HTTP/1.1
Date: Wed, 16 Jan 2013 15:23:02 +0000
```

The header has no `Content-MD5` field and no `Content-Type` field, so the second and third lines of the string that should be signed will be empty. There is also no `X-Aah-Date` field, so the canonicalized headers part is omitted. The parameters `validTime` and `cardHolder` are rearranged so they become lexicographically ordered, and the resulting string is:

```
GET\n
\n
\n
Wed, 16 Jan 2013 15:23:02 +0000\n
/api/v1/cards?cardHolder=jdoe&validTime=20130105T1200
```

This string, encoded as UTF-8, has a length of 91 characters. It is signed with the *session access key* `AQIDBAUGBwg=` (which was received in use case 7.1.1) using HMAC-SHA1. The resulting signature in hexadecimal is 341b5e6ed197d4d4dbb2148e67909e2aeedab68e. Encoded as Base 64 it becomes `NBtebtGX1NTbshSOZ5CeKu7ato4=`. This value is placed together with the *session ID* to form the header `Authorization AWS 342ba291:NBtebtGX1NTbshSOZ5CeKu7ato4=`.

7.1.3 Sign a request with a body

Prerequisites: Use case 7.1.1.

This use case describes how to sign the following PUT request.

```
PUT /api/v1/users/skierkegaard HTTP/1.1
Date: Wed, 4 Jan 1995 03:23:02 +0500
Content-Type: application/json; charset=utf-8
Content-Length: 93
X-Aah-Date: Wed, 16 Jan 2013 15:23:02 +0000

{\n
  "surname": "Kierkegaard",\n
  "givenName": "Søren",\n
  "userGroup": "Guests in suite 102"\n
}
```

The body in this example contains 88 printable characters and 4 line feed characters, resulting in 92 Unicode characters. (Note that there are 2 space characters at the beginning of each indented line and that the line breaks in this example are single line feed characters. Unlike the HTTP header, which always uses both carriage return and line feed in the line breaks, the line breaks of the body can be of any type.)

When the body is encoded as UTF-8, ϕ (U+00F8) is encoded as *0xc3 0xb8* so the 92 characters will occupy 93 octets. The MD5 digest of these octets is 943120b729dd663786412ac352cebd13. Encoded as Base64 it becomes `lDEgtyndZjeGQSrDUs69Ew==`.

The MD5 digest should be included both as a `Content-MD5` field in the header and as the second line of the string that should be signed. This request has a specified `Content-Type` field, so this is included on the third line.

The `Date` of this request does not match the server time, so the client has chosen to include the `X-Aah-Date` field. Therefore, the fourth line is empty and the `X-Aah-Date` header is included in its canonical form before the resource path.

```
PUT\n
lDEgtyndZjeGQSrDUs69Ew==\n
application/json; charset=utf-8\n
\n
x-aah-date:Wed, 16 Jan 2013 15:23:02 +0000\n
/api/v1/users/skierkegaard
```

This string, encoded as UTF-8, has a length of 131 characters. It is signed with the *session access key* `AQIDBAUGBwg=` (which was received in use case 7.1.1) using HMAC-SHA1. The resulting signature in hexadecimal is 73feaef17f9c90ed767010f73d8f707734b431fe. Encoded as Base64 it becomes `c/6u8X+ckO12cBD3PY9wdzS0Mf4=`. This value is placed together with the *session ID* to form the `Authorization` field. The `Content-MD5` field also need to be added to the header, as shown below.

```
Content-MD5: lDEgtyndZjeGQSrDUs69Ew==
Authorization: AWS 342ba291:c/6u8X+ckO12cBD3PY9wdzS0Mf4=
```

7.1.4 Terminate a session

Prerequisites: Use case 7.1.1.

The client may terminate the session instead of relying on the session being removed after a period of inactivity.

```
DELETE /api/v1/sessions/342ba91

200 OK
```

7.2 Create and read user information

These use cases show how to create a new user and how to read the information about an existing user.

7.2.1 Create a new user

This section describes how to create a new user. The client sends a PUT request with set of properties for the user as shown in the below example. For details of each peroperty, please see the section 5.51.

```
PUT /api/v1/users/user1
{
 "autoDelete": false,
 "cellPhone": "12345",
 "pin":"1224",
 "confirmPin":"1224",
 "defaultReportTime": "03:00",
 "email": "email@email.com",
 "givenName": "Name",
 "hidden": false,
 "surname": "SurName",
 "userDetails1": "detail1",
 "userDetails2": "detail2",
 "userDetails3": "detail3",
 "userDetails4": "detail4",
 "userGroup": "UserGroup1"
}

The server fills unspecified properties with default values and attempts to create a new user.
If successful, information about the new user will be stored in the database.
The new resource is returned in the response as given below.

2015-09-17T15:19:41 Response:
201 Created

{
    "autoDelete": false,
    "cellPhone": "12345",
    "defaultReportTime": "03:00",
    "email": "email@email.com",
    "extendedExpireTime": null,
    "givenName": "Name",
    "hidden": false,
```

```
    "id": "user1",
    "surname": "SurName",
    "userDetails1": "detail1",
    "userDetails2": "detail2",
    "userDetails3": "detail3",
    "userDetails4": "detail4",
    "userGroup": "UserGroup1"
}
```

7.2.2 Modify information for existing user

The client instructs the server to update the information of an existing user in the system. The client sends a POST request with set of properties as shown in below example to update the information of user. For details please see the section 5.51.

```
POST /api/v1/users/user1
{
  "id": "user1",
 "cellPhone": "9876543",
 "defaultReportTime": "05:00",
 "email": "email@email.com",
 "userDetails1": "detail1",
 "userDetails2": "detail2",
 "userDetails3": "detail3",
 "userDetails4": "detail4",
 "userGroup": "UserGroup1"
}

If successful, information will be updated in the database and
the following response will be returned:

2015-09-17T15:20:12 Response:
200 OK

{
    "autoDelete": false,
    "cellPhone": "9876543",
    "defaultReportTime": "05:00",
    "email": "email@email.com",
    "extendedExpireTime": null,
    "givenName": "Name",
    "hidden": false,
    "id": "user1",
    "surname": "SurName",
    "userDetails1": "detail1",
    "userDetails2": "detail2",
    "userDetails3": "detail3",
    "userDetails4": "detail4",
    "userGroup": "UserGroup1"
}
```

7.3 Create and read guest cards

These use cases show how anonymous guest cards can be created either directly at an encoder or automatically at the door.

7.3.1 Encode a new card

Prerequisites: A door named 101 and an encoder named EA1.

The client instructs the server to create a new card resource by specifying guest door 101 but does not specify a **cardHolder**. The client also adds the parameter `action` and `encoder`, telling the server to encode the card directly.

```
POST /api/v1/cards?action=encode&encoder=EA1
{
    "expireTime": "20130102T1210",
    "format": "rfid48",
    "doorOperations": [ { "operation": "guest", "doors": ["101"] } ]
}
```

The server fills unspecified properties with default values and attempts to encode a card. If successful, it obtains the *serial number* of the card from the encoder and stores all information about the card in the database. The new resource is returned in the response.

```
201 Created
Location: https://dc.example.com/api/v1/cards/135792
{
    "id": "135792",
    "cardHolder": "guest",
    "created": "20121220T1432",
    "startTime": "20121220T1400",
    "expireTime": "20130102T1300",
    "cancelled": false,
    "format": "rfid48",
    "doorOperations": [ { "operation": "guest", "doors": ["101"],
                          "doorGroups": [] } ],
    "numberOfIssuedCards": 0,
    "serialNumbers": [ "01020304050607" ]
}
```

7.3.2 Read the contents of an encoded card

Prerequisites: An encoder named EA1 and an encoded card.

The client instructs the server to read the card on the encoder.

```
POST /api/v1/cards?action=readCard&encoder=EA1
```

The server reads the card on the specified encoder and returns all information about it to the client, in the same format as that of a response to a GET request for a card resource.

```
200 OK
{
    "id": "135792",
    "cardHolder": "guest",
    ...
    "serialNumbers": [ "01020304050607" ]
}
```

7.3.3 Read serial number of an empty card

Prerequisites: An encoder named EA1 and an empty card.

The client instructs the server to read the card on the encoder.

```
POST /api/v1/cards?action=readCard&encoder=EA1
```

The server doesn't recognize the card as one belonging to this system, and therefore only returns the serial number.

```
200 OK
{
    "serialNumbers": [ "08090a0b0c0d0e" ]
}
```

7.3.4 Prepare a card for auto-update

Prerequisites: A door named 101.

The client instructs the server to create a new card resource by specifying guest door 101 but does not specify a **cardHolder**. The client knows the *serial number* of the physical card and includes this in the request.

```
POST /api/v1/cards
{
    "expireTime": "20130102T1210",
    "format": "rfid48",
    "doorOperations": [ { "operation": "guest", "doors": ["101"] } ],
    "serialNumbers": [ "01020304050607" ]
}
```

The server fills unspecified properties with default values and stores information about the card in the database. The card can immediately get encoded at an auto-update station. The new resource is returned in the response.

```
201 Created
Location: https://dc.example.com/api/v1/cards/135792
{
    "id": "135792",
    "cardHolder": "guest",
    "created": "20121220T1432",
    "startTime": "20121220T1400",
    "expireTime": "20130102T1300",
    "cancelled": false,
    "format": "rfid48",
```

```
    "doorOperations": [ { "operation": "guest", "doors": ["101"],
                          "doorGroups": [] } ],
    "numberOfIssuedCards": 1,
    "serialNumbers": [ "01020304050607" ]
}
```

The server will try to send the card data to the involved guest room doors, but this may not happen immediately because the door may not be online at the moment.

7.3.5 Issue a card that is a joiner to another card

Prerequisites: Use case 7.6.1 or 7.6.4.

Two guest cards that are issued at different times can access the same room if the later card is specified as a *joiner* to the first one.

The first card is issued without specifying a **joiners** field. If the ID of this card is unknown, it can e.g. be found using the method described in use case 7.9.1.

The client puts the ID of the first card as an item in the **joiners** list.

```
POST /api/v1/cards?fields=joiners
{
    "expireTime": "20130105T1200",
    "format": "rfid48",
    "joiners": ["135792"],
    "doorOperations": [ { "operation": "guest", "doors": ["101"] } ]
}
```

The server verifies that at least one of the doors in **doorOperations** are shared between the cards. Since the client supplied the parameter `fields=joiners`, the response object will contain the resulting card resource, *including* the **joiners** field.

```
201 Created
Location: https://dc.example.com/api/v1/cards/135822
{
    "id": "135822",
    ...
    "joiners": ["135792"]
}
```

The **joiners** list of the second card includes the ID of the first card. The first card also includes the ID of the second card in the **joiners**, which can be investigated with a GET request that includes the parameter `fields=joiners`.

```
GET /api/v1/cards/135792?fields=joiners

200 OK
{
    "id": "135792",
    ...
    "joiners": ["135822"]
}
```

7.3.6 Issue a card that overrides another card

Prerequisites: Use case 7.6.1 or 7.6.4.

The client tries to issue a card that overlaps in time and with the doors of another, already issued, card.

```
POST /api/v1/cards?action=encode&encoder=EA1
{
    "expireTime": "20130102T1210",
    "format": "rfid48",
    "doorOperations": [ { "operation": "guest", "doors": ["101"] } ]
}
```

The server detects this and returns an error

```
403 Forbidden
{
    "code": 40312,
    "message": "Guest card overlaps with existing card.",
    ...
}
```

The client decides to override this card, rather than issuing a joiner card, and repeats the request but includes the `override` parameter. This time, the server carries out the request.

```
POST /api/v1/cards?action=encode&encoder=EA1&override=true
{
    "expireTime": "20130102T1210",
    "format": "rfid48",
    "doorOperations": [ { "operation": "guest", "doors": ["101"] } ]
}

201 Created
{
    ...
}
```

Once the new card has been used in a guest room door, the overridden guest card will be denied access to that door.

7.4 Create guest card advanced

These use cases show how to create guest card advanced.

The following parameters and properties can be used for creation of a guest card advanced.

7.4.1 Parameters

action - The value of this parameter should be `encode` to create the guest card advanced.

encoder - Set to the encoder on which cards will be encoded.

7.4.2 Properties

startTime - The date and time of when the card becomes valid. Defaults to the current time.

expireTime - The date and time of when the card becomes invalid.

format - The format of the guest card advanced must be TLCode (for details please refer to the section 5.17.3) Note: This is a mandatory property for creation of guest card advanced.

doorOperations - List of objects specifying which doors the card has access to. Each object in the list has the following properties:

operation - Door operation for the doors in this object. It must be set to `guest` for guest rooms and `normal` for doors to common rooms. Atleast one guest door should be provided for guest card advanced.

doors - List of doors that should be opened with the specified operation. The first door in the list of the first object with **operation** `guest` has the special role of being the main door of the card.

The client instructs the server to create a guest card advanced (e.g specifying door 101.) The client also needs to provide a value of the **format** property as **TLCode** for guest, and add the parameter `action` as `encode`.

To create a guest card advanced for a given door and encode the card on the provided encoder, the client needs to provide the value for **operation** as **guest** atleast for one door and the value for **encoder** as **name of encoder on which card will encode**.

```
POST /cards?action=encode&encoder=Enc1&overwriteValidCard=true
```

```
{
    "doorOperations": [
        {
            "doors": [
                "101"
            ],
            "operation": "guest"
        }
    ],
    "expireTime": "20150527T2359",
    "format": "TLCode",
}
```

After creation of a card, the response will contain the information on the card as follows:

```
2015-05-27T09:23:44 Response:
201 Created

{
    "cancelled": false,
    "cardHolder": "Guest",
    "created": "20150527T092325",
    "discarded": false,
    "doorOperations": [
        {
            "doorGroups": [],
            "doors": [
                "101"
            ],
            "operation": "guest"
        },
        {
            "doorGroups": [],
            "doors": [
---
            ],
            "operation": "normal"
        }
    ],
    "expireTime": "20150528T0000",
    "format": "rfid201",
    "id": "171406",
    "numberOfIssuedCards": 1,
    "pendingAutoUpdate": false,
--
    "startTime": "20150527T092330"
}
```

7.5 Create mobile key for guest

These use cases show how to create a mobile key for a guest. The key will be sent to the guest's mobile phone.

The following parameters and properties can be used for creation of mobile key.

7.5.1 Parameters

action - The value of this parameter should be `mobileAccess` to create the mobile key for a door.

autoJoin - If set to `true`, this mobile key will join with any overlapping card/key for the same door. Cannot be used in combination with `override`.

override - If set to `true`, this mobile key will override any overlapping card/mobile key if exists. Cannot be used in combination with `autoJoin`.

7.5.2 Properties

startTime - The date and time of when the mobile key becomes valid. Defaults to the current time.

expireTime - The date and time of when the mobile key becomes invalid.

format - Format of the card. It must be rfid48 for a mobile key (for details please refer to section 5.17.3). Note: This is a mandatory property for creation of a mobile key.

doorOperations - List of objects specifying to which doors the mobile key has access. Each object in the list has the following properties:

operation - Door operation for the doors in this object. It must be set to `guest` for guest rooms and `normal` for doors to common rooms.

doors - List of doors that should be opened with the specified operation. The first door in the list of the first object with **operation** `guest` has the special role of being the main door of the card.

endPointID - Unique endPointID assigned to the guest's mobile phone.

accountName - Starting from version 1.18.0, multiple credential services accounts can be configured in Visionline server. So, this property provides the unique name of the credential services account to be used. If one or more credential services accounts are configured in Visionline then it is mandatory to mark one of them as default account. If this field is missing from the request then Visionline server will use the default account for the requested operation.

label - It is an optional property. This field provides "Label" field data for mobile keys. This field may contain plain text as well as predefined placeholders. The placeholders are surrounded by percentage % sign. VISIONLINE replaces these placeholders with their actual values before issuing mobile key. The final text will be truncated to 32 characters. The valid placeholders are:

ROOMNUM - Replaced with main guest room number or name
ROOMLIST - Replaced with comma separated list of guest rooms
ROOMRANGE - Replaced with hyphen separated room range or if some rooms do not fit intoany range then they are added as comma separated list.
SITENAME - Name of the property
CARDNUM - Credential ID
UUID - BLE UUID
USERID - the value passed in the UI field

Example:
CCA;EA0;MK17282772;GR101;CO201506021200;AM1;IOCLERK1;LBRoom#%ROOMNUM%. UserID: %USERID%;
Result:
Label: "Room#101. UserID: xy-am-123"

description - It is an optional property. This field provides "Description" field data for mobile keys. This field may contain plain text as well as predefined placeholders. The placeholders are surrounded by percentage % sign. VISIONLINE replaces these placeholders with their actual values before issuing mobile key. The final text will be truncated to 32 characters. The valid placeholders are:

ROOMNUM - Replaced with main guest room number or name
ROOMLIST - Replaced with comma separated list of guest rooms
ROOMRANGE - Replaced with hyphen separated room range or if some rooms do not fit intoany range then they are added as comma separated list.
SITENAME - Name of the property
CARDNUM - Credential ID
UUID - BLE UUID

USERID - the value passed in the UI field

Example:
CCA;EA0;MK17282772;GR101;CO201506021200;AM1;IOCLERK1;DSRoom#%ROOMNUM% at hotel: %SITENAME%;
Result:
Description: ”Room#101 at hotel: Beachfront”

url - It is an optional property. This field provides “URL” field data for mobile keys. This field may contain plain text as well as predefined placeholders. The placeholders are surrounded by percentage % sign. VISION-LINE replaces these placeholders with their actual values before issuing mobile key. The valid placeholders are:

ROOMNUM - Replaced with main guest room number or name
ROOMLIST - Replaced with comma separated list of guest rooms
ROOMRANGE - Replaced with hyphen separated room range or if some rooms do not fit intoany range then they are added as comma separated list.
SITENAME - Name of the property
CARDNUM - Credential ID
UUID - BLE UUID
USERID - the value passed in the UI field

Example:
CCA;EA0;MK17282772;GR101;CO201506021200;AM1;IOCLERK1;URhttp://www.assa.com/Guest%ROOMNUM%;
Result:
URL: ”www.assaabloy.com/Guest101”

7.5.3 Create mobile key with override

Prerequisites: A door named e.g. 101 and an endPointId named e.g. endPointId1.

The client instructs the server to create a new mobile key for the door by specifying door 101. The client also needs to provide a value of the **format** property as **rfid48** for guest, and add the parameter `action` as `mobileAccess`.

The server will create the mobile key for a given door and directly send the mobile key to the guest’s mobile phone. The client needs to provide the value for **operation** as **guest** and the value for **override** as **true**.

```
POST /cards?action=mobileAccess&override=true
{
    "doorOperations": [
        {
            "doors": [
                "101"
            ],
            "operation": "guest"
        }
    ],
    "expireTime": "20150522T2359",
    "format": "rfid48",
    "endPointID":"endPointId1",
    "accountName":"AssaAbloyCredentialAccount"
}
```

The server fills unspecified properties with default values and attempts to create a mobile key for the given door. If successful, it returns **201 Created** and directly sends the mobile key to the guest's mobile phone.

The request contains value of the property "accountName", hence mobile key will be sent to the endpoint "endPointId1" using credential account "AssaAbloyCredentialAccount" configured in Visionline.

If user has configured "Label" field in VISIONLINE as **"Room %ROOMNUM% at %SITENAME%"** and as this request does not include "label" property, so VISIONLINE will use configured field and send the "label" text as **"Room 101 at DreamLand"** with mobile keys.
If user has not configured "Label" field in VISIONLINE and as this request also does not include "label" property then VISIONLINE will send the default "Label" text as **"Room 101"**.

If user has configured "Description" field in VISIONLINE as **"%CARDNUM%;%USERID%"** and as this request does not include "description" property, so VISIONLINE will use configured field and send the "Description" text as **"1000177;XAB5413888"** with mobile keys.
If user has not configured "Description" field in VISIONLINE and as this request also does not include "description" property then VISIONLINE will send the default "Description" text as **"1000177"**.

After issue of the mobile key, the response will contain the credential Id of the key as follows:

```
2015-05-29T12:55:58 Response:
201 Created

{
    "credentialID": 1000177
}
```

7.5.4 Create mobile key as joiner key

Prerequisites: A door named e.g. 101 and an endPointId named e.g. endPointId1

The client instructs the server to create a new mobile key for the door by specifying door e.g. 101 (existing overlapping card/mobile key and new mobile key, both should work at the door). The client also needs to provide a value of the **format** property as **rfid48** for guest, and add the parameter `action` as `mobileAccess`.

The server will create the mobile key for a given door and directly send the mobile key to the guest's mobile phone. The client also needs to provide a value of the **operation** as **guest**. Both the existing card/mobile key and the new created mobile key should work at the door.

```
POST /cards?action=mobileAccess&autoJoin=true
{
    "doorOperations": [
        {
            "doors": [
                "101"
            ],
            "operation": "guest"
        }
    ],
    "expireTime": "20150522T2359",
    "format": "rfid48",
    "endPointID":"endPointId1"
}
```

The server fills unspecified properties with default values and attempts to create a mobile key for the given door. If successful, it returns **201 Created** and directly sends the mobile key to the guest's mobile phone. Both existing card/mobile key and the new created mobile key should work at the door.

The request does not contains property "accountName", hence mobile key will be sent to the endpoint "endPointId1" using default credential account configured in Visionline.

If user has configured "Label" field in VISIONLINE as **"Room %ROOMNUM% at %SITENAME%"** and as this request does not include "label" property, so VISIONLINE will use configured field and send the "label" text as **"Room 101 at DreamLand"** with mobile keys.
If user has not configured "Label" field in VISIONLINE and as this request also does not include "label" property then VISIONLINE will send the default "Label" text as **"Room 101"**.

If user has configured "Description" field in VISIONLINE as **"%CARDNUM%;%USERID%"** and as this request does not include "description" property, so VISIONLINE will use configured field and send the "Description" text as **"1000177;XAB5413888"** with mobile keys.
If user has not configured "Description" field in VISIONLINE and as this request also does not include "description" property then VISIONLINE will send the default "Description" text as **"1000177"**.

After issue of the mobile key, the response will contain the credential Id of the key as follows:

```
2015-05-29T12:55:58 Response:
201 Created

{
    "credentialID": 1000177
}
```

7.5.5 Create mobile key with label, description and url meta data

Prerequisites: A door named e.g. 101 and an endPointId named e.g. endPointId1.

The client instructs the server to create a new mobile key for the door by specifying door 101. The client also needs to provide a value of the **format** property as **rfid48** for guest, and add the parameter `action` as `mobileAccess`. The client also provies values of the **label** property as **Room#%ROOMNUM%**, **description** property as **Credential ID#%CARDNUM%** and **url** property as **http://www.assa.com/Guest%USERID%/Welcome**.

The server will create the mobile key for a given door and directly send the mobile key to the guest's mobile phone. The client needs to provide the value for **operation** as **guest** and the value for **override** as **true**.

```
POST /cards?action=mobileAccess&override=true
{
    "doorOperations": [
        {
            "doors": [
                "101"
            ],
            "operation": "guest"
        }
    ],
    "expireTime": "20150522T2359",
    "format": "rfid48",
    "endPointID":"endPointId1",
    "label":"Room#%ROOMNUM%",
    "description":"Credential ID#%CARDNUM%",
```

```
    "url":"http://www.assa.com/Guest%USERID%/Welcome"
}
```

The server fills unspecified properties with default values and attempts to create a mobile key for the given door. If successful, it returns **201 Created** and directly sends the mobile key to the guest's mobile phone.
The request does not contains property "accountName", hence mobile key will be sent to the endpoint "endPointId1" using default credential account configured in Visionline.
The content of the **label**, **description** and **url** fields are processed by VISIONLINE and any placeholders found in these fields are replaced with actual values. The final text sent with mobile key will be:
Lable: "Room#101"
Description: "Credential ID#11000021"
URL: "http://www.assa.com/Guest123456789/Welcome"

After issue of the mobile key, the response will contain the credential Id of the key as follows:

```
2015-05-29T12:55:58 Response:
201 Created

{
    "credentialID": 1000177
}
```

7.6 Create and read staff cards

These use cases show how staff cards can be created either directly at an encoder or automatically at the door.

7.6.1 Encode a new card

Prerequisites: A door named e.g. 101 and an encoder named e.g. EA1.

The client instructs the server to create a new card resource by specifying door e.g 101 and a **cardHolder**. The client also needs to provide value of the **format** property as **TLCode** (which means that the card is greater than **rfid48**). The client also add the parameters `action` and `encoder` and tells the server to encode the card directly. The client can also provide the value for the **timeSchedule** parameter (optional). The value of **operation** can be provided as **normal**.

7.6.2 Properties

startTime - The date and time of when the card becomes valid. Defaults to the current time and for staff card this date should not be in the future.

expireTime - The date and time of when the staff card becomes invalid.

format - Format of the card. It must be **TLCode** for a staff card (for details please refer to section 5.17.3). Note: This is a mandatory property for creation of a staff card.

doorOperations - List of objects specifying to which doors the card has access. Each object in the list has the following properties:

operation - Door operation for the doors in this object. It must be set to `guest` for guest rooms, `normal` for doors to common rooms and `on/off` for on/off doors. If any entry has operation value as **guest** than card will be treated as Advanced guest card insted of staff card.

doors - List of doors that should be opened with the specified operation.

timeSchedule - Name of time schedule.

cardHolder - Name of the card holder and It is a mandatorl property.

counters - It is an optional property. List of objects specifying the names of counters applicable on card.

```
POST /api/v2/cards?action=encode&encoder=EA1
{
    "cardHolder": "sym",
    "expireTime": "20130102T1210",
    "format": "TLCode",
    "doorOperations": [ { "operation": "normal", "doors": ["101"] } , { "operation": "on/off",
    "doors": ["107"] } ],
    "timeSchedule":"Card always",
    "counters":{"counterNames":["Counter1", "Counter2"]}
}
```

The server fills unspecified properties with default values and attempts to encode a card. If successful, it obtains the *serial number* of the card from the encoder and stores all information about the card in the database. The new resource is returned in the response.

```
201 Created
Location: https://dc.example.com/api/v2/cards/135792
{
    "id": "135792",
    "cardHolder": "sym",
    "created": "20121220T1432",
    "startTime": "20121220T1400",
    "expireTime": "20130102T1300",
    "cancelled": false,
    "format": "rfid98",
   "counters": [
        {
            "counterName": "Counter1",
            "counterValue": 2,
            "decValue": 1,
            "doors": [
                "103"
            ],
            "initialValue": 2,
            "lastUse": "n/a",
            "resetTime": "14:50",
            "timeSchedule": "AMS Checking"
        },
        {
            "counterName": "Counter2",
            "counterValue": 2,
            "decValue": 1,
            "doors": [
                "105",
                "106"
```

```
            ],
            "initialValue": 2,
            "lastUse": "n/a",
            "resetTime": "09:36",
            "timeSchedule": "timeSchedule2"
        }
    ],
    "doorOperations": [
{
"operation": "normal",
 "doors": ["101","103","105","106"],
"doorGroups": []
},
          {
"doorGroups": [],
"doors": ["107"],
"operation": "on/off"
}
 ],
    "numberOfIssuedCards": 1,
    "serialNumbers": [ "01020304050607" ],
    "timeSchedule":"Card always"
}
```

7.6.3 Read the contents of an encoded staff card

Prerequisites: An encoder named e.g. EA1 and an encoded card.

The client instructs the server to read the staff card on the encoder.

```
POST /api/v1/cards?action=readCard&encoder=EA1
```

The server reads the card on the specified encoder and returns all information about it to the client, in the same format as that of a response to a GET request for a card resource.

```
200 OK
{
    "id": "135792",
    "cardHolder": "sym",
    ...
    "serialNumbers": [ "01020304050607" ],
    "timeSchedule":"Card always"
}
```

7.6.4 Prepare a staff card for auto-update

Prerequisites: A door named e.g. 101.

The client instructs the server to create a new card resource by specifying door e.g. 101 and a **cardHolder**. The client knows the *serial number* of the physical card and includes this in the request. This operation can also be used to change the credential of the existing card if already encoded.

```
POST /api/v2/cards
{
     "cardHolder": "sym",
    "expireTime": "20130102T1210",
    "format": "TLCode",
    "doorOperations": [ { "operation": "normal", "doors": ["101"] } ],
    "timeSchedule":"Card always",
    "serialNumbers": [ "01020304050607" ]
}
```

The server fills unspecified properties with default values and stores information about the card in the database. The card can immediately get encoded at an auto-update station. The new resource is returned in the response.

```
201 Created
Location: https://dc.example.com/api/v2/cards/135792
{
    "id": "135792",
    "cardHolder": "sym",
    "created": "20121220T1432",
    "startTime": "20121220T1400",
    "expireTime": "20130102T1300",
    "cancelled": false,
    "format": "rfid98",
    "doorOperations": [ { "operation": "normal", "doors": ["101"],
                          "doorGroups": [] } ],
    "numberOfIssuedCards": 1,
    "serialNumbers": [ "01020304050607" ],
    "timeSchedule":"Card always"
}
```

Note: Staff cards cannot be auto-updated in locks and thus nothing is sent to the locks.

7.7 Modify cards

These uses cases show different ways of modifying cards after they were created.

7.7.1 Change the doors of an issued card

Prerequisites: Use case 7.6.1 or 7.6.4.

POST request contains the full list of changed doors that the card should have access to.

```
POST /api/v1/cards/135792
{
    "doorOperations": [ { "operation": "guest", "doors": ["101","150","102"] } ]
}
```

The server stores the changes and sends back the current contents of the card resource.

```
200 OK
{
    "id": "135792",
    "doorOperations": [ { "operation": "guest", "doors": ["101","102","150"],
                          "doorGroups": [] } ],
    ...
}
```

As in use case 7.6.4, the changes will be available at an auto-update station immediately, but it may take some time until it reaches the guest room doors.

7.7.2 Deactivate a card at a specific time

Prerequisites: Use case 7.6.1, 7.6.4 or 7.8.5.

If an already issued card should no longer have access to any doors after a certain time, the client specifies this by modifying **expireTime** with a POST request.

```
POST /api/v1/cards/135792
{
    "expireTime": "20130116T1717"
}
```

The server stores the changes and sends back the current contents of the card resource.

```
200 OK
{
    "id": "135792",
    "expireTime": "20130116T1800",
    ...
}
```

As in use case 7.6.4, the new expire time will not reach the card until it is used at its guest room door, after that door has been available online.

7.7.3 Extend the access time for a card

Prerequisites: Use case 7.6.1, 7.6.4 or 7.8.5.

If an already issued card should be valid for a longer time than initially set, this is done in exactly the same way as use case 7.7.2, but with an **expireTime** that occurs later than the current value.

7.7.4 Cancel a card that was misplaced

Prerequisites: Use case 7.6.1, 7.6.4, 7.8.1 or 7.8.5.

Instead of shortening the **expireTime**, as shown in use case 7.7.2, a card can be *cancelled*, by setting the **cancelled** property to *true*.

```
POST /api/v1/cards/135792
{
    "cancelled": true
```

```
}

200 OK
{
    "id": "135792",
    "cancelled": true,
    ...
}
```

The server sends the order to cancel the card to both guest room doors and entrance doors that this card have access to. Thus, the card will not need to be encoded at a guest room door before it is blocked from accessing the entrance doors, as is the case in use case 7.7.2.

If the card is never detected, the order to cancel it will remain in the entrance doors and guest room doors until the **expireTime** encoded on the card has passed. This places a limit on the number of cards that can be cancelled at once, a limit which does not exist when the card is restricted using a shorter **expireTime** leaving the **cancelled** property at *false*.

7.7.5 Update joiners of a card automatically

Prerequisites: 7.3.5

If the client wants to add doors to a card that has joiners, the modified card will override the old joiner cards. It must then recreate the overridden cards as joiners of the overriding card. This can be performed with a single request using the `updateJoiners` query parameter.

From use case cards 135792 and 135822 are joiners for door 101. The client adds door 103 to card 135792 using the following request.

```
POST /api/v1/cards/135792?updateJoiners=true
{
    "doorOperations": [ { "operation": "guest", "doors": ["101", "103"] } ]
}
```

The server sends the card image for 135792 to doors 101 and 103. This image overrides card 135822 in door 101. However, the server also sends a new image for card 135822 to doors 101 and 160, and with this image the cards are joiners again. The server responds with the modified details of card 135792.

```
200 OK
{
    "id": "135792",
    "doorOperations": [ { "operation": "guest", "doors": ["101", "103"],
                          "doorGroups": [] } ],
    ...
}
```

7.8 User groups and guest cards

These use cases give example of operations involving creating users and user groups, and using these resources when creating and modifying cards.

7.8.1 Create a user group without uniform access

Prerequisites: A room named "Pool room" and a blocking group named "Loyal guests".

A user group named "Loyal guests" is created by the client. By setting **uniformAccess** to *false*, the group can be used for keeping a list of doors or for placing users in a certain *blocking group*.

```
PUT /api/v1/userGroups/Loyal%20guests
{
    "uniformAccess": false,
    "doorOperations": [ { "operation": "normal", "doors": ["Pool room"] } ],
    "blockingGroup": "Loyal guests"
}


201 Created
Location: https://dc.example.com/api/v1/userGroups/Loyal%20guests
{
    "id": "Loyal guests",
    "uniformAccess": false,
    "doorOperations": [ { "operation": "normal", "doors": ["Pool room"],
                          "doorGroups": [] } ],
    "blockingGroup": "Loyal guests"
}
```

7.8.2 Issue a card using doors from a user group

Prerequisites: Use case 7.8.1.

The doors of the user group "Loyal guests" are read with a GET request.

```
GET /api/v1/userGroups/Loyal%20guests

200 OK
{
    "doorOperations": [ { "operation": "normal", "doors": ["Pool room"],
                          "doorGroups": [] } ],
    ...
}
```

The contents of the property **doorOperations** can then be appended to the **doorOperations** of a card that shall have access to these doors.

```
POST /api/v1/cards
{
    "expireTime": "20130102T1210",
    "format": "rfid48",
    "doorOperations": [ { "operation": "guest", "doors": ["101"] },
                        { "operation": "normal", "doors": ["Pool room"] } ]
}
```

The server creates the new card resource with some of its doors taken from the user group, but apart from this, there is no link between the card and the user group. The new resource is returned in the response.

```
201 Created
Location: https://dc.example.com/api/v1/cards/135792
{
    "id": "135792",
    "cardHolder": "guest",
    ...
    "doorOperations": [ { "operation": "guest", "doors": ["101"], "doorGroups": []  },
                        { "operation": "normal", "doors": ["Pool room"],
                          "doorGroups": []  } ]
}
```

The conditions at the end of use case 7.6.4 applies here as well.

7.8.3 Create a user group with uniform access

Prerequisites: Doors named 102, 103, 105, 106 and a blocking group named "guests".

A user group named "Guests in suite 102" is created by the client. By setting **uniformAccess** to *true*, any card issued to a **cardHolder** belonging to this group will receive **doorOperations** from here.

```
PUT /api/v1/userGroups/Guests%20in%20suite%20102
{
    "uniformAccess": true,
    "doorOperations": [ { "operation": "guest", "doors": ["102","103","105","106"] } ],
    "blockingGroup": "guests"
}
```

```
201 Created
Location: https://dc.example.com/api/v1/userGroups/Guests%20in%20suite%20102
{
    "id": "Guests in suite 102",
    "uniformAccess": true,
    "doorOperations": [ { "operation": "guest", "doors": ["102","103","105,"106"],
                          "doorGroups": []  } ],
    "blockingGroup": "guests"
}
```

7.8.4 Create a user with extended expire time and uniform access

Prerequisites: Use case 7.8.3.

The client creates a user belonging to a group with uniform access enabled. All card issued to this user will receive **doorOperations**from the user group. The user also has **extendedExpireTime**, which will automatically extend **expireTime** of the card when required.

```
PUT /api/v1/users/jdoe
{
    "surname": "Doe",
    "givenName": "John",
    "userGroup": "Guests in suite 102",
    "extendedExpireTime": "20170615T1200"
}
```

```
201 Created
Location: https://dc.example.com/api/v1/users/jdoe
{
    "id": "jdoe",
    "surname": "Doe",
    "givenName": "John",
    "userGroup": "Guests in suite 102",
    "extendedExpireTime": "20170615T1200"
}
```

7.8.5 Issue a card to a user with extended expire time and uniform access

Prerequisites: Use case 7.8.4.

The client issues a card to user `JohnDoe`, known by his ID as `jdoe`. Since he belongs to a group with *uniform access*, **doorOperations** may not be specified in the request. Since the user has **extendedExpireTime**, the client need not specify **expireTime**. Instead it will be assigned an appropriate value by the system, never exceeding **extendedExpireTime**. In this case, the client supplies the serial number of the card. If this is omitted, the card must be encoded according to use case 7.6.1.

```
POST /api/v1/cards
{
    "cardHolder": "jdoe",
    "format": "rfid48",
    "serialNumbers": ["01020304050607"]
}
```

The system-assigned values of the properties **doorOperations** and **expireTime** are displayed in the response.

```
201 Created
Location: https://dc.example.com/api/v1/cards/135792
{
    "id": "135792"
    "cardHolder": "jdoe",
    "created": "20121220T1432",
    "startTime": "20121220T1400",
    "expireTime": "20130701T0000",
    "doorOperations": [ { "operation": "guest", "doors": ["102","103","105","106"],
                          "doorGroups": []  } ],
    ...
    "numberOfIssuedCards": 1,
    "serialNumbers": ["01020304050607"]
}
```

The conditions at the end of use case 7.6.4 applies here as well.

7.8.6 Change the doors for all cards within a user group with uniform access

Prerequisites: Use case 7.8.5 and a room named 108.

The **doorOperations** property of cards belonging to a user within a group with uniform access cannot be modified using a POST to the card resource. Instead, the **doorOperations** of the user group need to be modified.

```
POST /api/v1/userGroups/Guests%20in%20suite%20102
{
    "doorOperations": [ { "operation": "guest", "doors": ["102","103","108"] } ]
}

200 OK
{
    "id": "Guests in suite 102",
    "doorOperations": [ { "operation": "guest", "doors": ["102","103","108"],
                          "doorGroups": []  } ],
    ...
}
```

All cards with **cardHolders** belonging to this group will be modified through auto-update, according to the end conditions of use case 7.6.4.

7.8.7 Change group of a user within a group with uniform access

Prerequisites: Use case 7.8.5 and a user group named "Guests in suite 110" created according to use case 7.8.3.

The client changes the **userGroup** of a user from one group with *uniform access* to another group with *uniform access*.

```
POST /api/v1/users/jdoe
{
    "userGroup": "Guests in suite 110"
}

200 OK
{
    "id": "jdoe",
    "surname": "Doe",
    "givenName": "John",
    "userGroup": "Guests in suite 110"
}
```

The doors of this user's card will get updated through auto-update.

7.9 Find cards

This use case shows how to search for cards in the system based on certain properties.

7.9.1 Find the card of a card holder

Prerequisites: Use case 7.8.5.

When modifying card resources directly, the client need to know the **id** of the card. If the user ID of the **cardHolder** is known, the client can find any card issued to this user that are or will be valid at a certain time.

```
GET /api/v1/cards?validTime=20130105T1200&cardHolder=jdoe
```

The server returns a list of object, regardless of the number of matching items it found. Each object only contains the **id** property.

```
200 OK
[
    {
        "id": "135792"
    }
]
```

The user is created according to use case 7.8.4 with **userGroup** set to `no room`. The card can now be created by following use case 7.8.5.

The card is connected to the user `jdoe`, but since group "no room" has no doors, the card cannot access anything. Later, when the rooms have been determined, the **userGroup** is changed according to use case 7.8.7 and the card will get encoded when it is used at its new guest room door.

7.10 Door groups

This use case shows how to find out information about a door group.

7.10.1 Find out which doors that belong to a door group

Prerequisites: Doors 101 and 102 belonging to door group 1st floor

If the name of a door group is known, the client and list the doors belonging to that door group .

```
GET /api/v1/doors?doorGroup=1st%20floor
```

If the door group exists, the server returns a list of object. Each object only contains the **id** property.

```
200 OK
[
    {
        "id": "101"
    }
    {
        "id": "102"
    }
]
```

7.11 Network nodes

These use cases show how to obtain information about the current state of the network.

7.11.1 Request information about all nodes in the network at once

Prerequisites: Use case 7.1.1.

The client sends a request for all nodes and receives a list of objects.

```
GET /api/v1/nodes

200 OK
[{
    "id": "ROOT",
    "children": ["00230000","00250000","00210000"],
    "ieeeAddress": null,
    ...
},{
    "id": "00230000",
    "children": [ "00230055" ],
    "ieeeAddress": "12:34:56:78:9a:bc:de:f0",
    ...
},{
    ...
}]
```

7.11.2 Request information about a single node in the network

Prerequisites: Use case 7.1.1.

The client determines the ID of the node of interest. The ID must be either `ROOT` or an ID returned in the **children** list of another *node* resource. The response to the request is a single object representing the node.

```
GET /api/v1/nodes/00230000

200 OK
{
    "id": "00230000",
    "children": [ "00230055" ],
    "ieeeAddress": "12:34:56:78:9a:bc:de:f0",
    ...
}
```

7.12 Sending commands to doors

These use cases show how commands that result in modifying the behaviour of the door can be sent over the network.

7.12.1 Unlock a door

Prerequisites: Use case 7.1.1.

The client sends a request containing a single property.

```
POST /api/v1/doors/101?timeout=30
{
    "open": true
}
```

The server sends the command to the door, and when it has been completed, the server responds `200 OK`.

7.12.2 Set stand open

Prerequisites: Use case 7.1.1.

The client sends a request containing a single property.

```
POST /api/v1/doors/101?timeout=30
{
    "standOpenTime": "20130822T1300"
}
```

The server sends the command to the door, and when it has been completed, the server responds `200 OK`.

7.12.3 Clear stand open

Prerequisites: Use case 7.1.1.

The client sends a request containing a single property.

```
POST /api/v1/doors/101?timeout=30
{
    "standOpenTime": null
}
```

The server sends the command to the door, and when it has been completed, the server responds `200 OK`.

7.13 Sending checkout commands

These use cases show how checkout commands can be sent over the network.

7.13.1 Check out all guests from a door

Prerequisites: Use case 7.1.1.

The client sends a request containing a single property.

```
POST /api/v1/doors/101
{
    "checkedOut": true
}
```

The server immediately responds `200 OK`. The guest is checked out from the door as soon as the server is able to communicate with the door. This command will also revoke any mobile keys that are issued for the given door.

7.13.2 Check out all guests from multiple doors

Prerequisites: Use case 7.1.1.

```
POST /api/v1/doors
{
    "doors": ["101","102"],
    "checkedOut": true
}
```

The server immediately responds `200 OK`. The guest is checked out from the doors as soon as the server is able to communicate with the doors. This command will also revoke any mobile keys that are issued for the given doors.

7.13.3 Check out all guests with grace time

Prerequisites: Use case 7.1.1.

```
POST /api/v1/doors/101
{
    "checkedOut": true,
    "checkoutTime": "20161216T1200"
}
```

The server immediately responds `200 OK`. If the door is online then all the guests are checked out from the doors after expiration of given checkout time. This command will also revoke any mobile keys that are issued for the given doors after expiration of given checkout time.

7.14 Set-up callback

These use cases show how to set-up a callback for events and cards.

7.14.1 Events

If you pass specific event codes when you set-up the callback you will only receive those events else you will receive all events. You can also specify which fields that should be included in the callback if you do not want all fields.

Available fields for events are:

doorID - the room number of the door

registrationNumber - the card registration number if event was generated by a card

eventTime - the time when the event occurred

eventCode - the number of the event

cardNameID - the type of card that generated the event

repeatFlags - a flag that will be set if the same event occurred multiple times within the same minute

This example will filter on the below events:

304 - Alert, Battery level low

912 - The door is opened

916 - The door is left open too long

```
POST /callback
{
    "resources": {
        "events": {
            "eventCode": [
                304,
                912,
                916
            ],
            "fields": [
                "doorID",
                "eventCode",
                "eventTime"
            ]
        }
    }
}
```

7.14.2 EMI Events

You can specify which fields that should be included in the callback if you do not want all fields.

Available fields for EMI events are:

doorID - the room number of the door

eventCode - the number of the event

registrationNumber - the card registration number if event was generated by a card

expireTime - the expiration time of the card

mainGuestRoom - the main guest room of the guest card

userGroupID - the number of the user group of the card

suiteRooms - the adjacent rooms available to the guest card

firstEntry - flag that is set when the first guest of a party enters

```
POST /callback
{
    "resources": {
        "emiEvents": {
            "fields": [
                "doorID",
                "eventCode"
            ]
        }
    }
}
```

7.14.3 Cards

You can specify which fields that should be included in the callback if you do not want all fields.

Available fields for cards are:

registrationNumber - the card registration number if event was generated by a card

status - the status of the card where 0=cancelled, 1=valid, 2=overridden, 3=not issued, 4=expired, 5=overwritten, 6=discarded

cancelTime - the time when the card was cancelled

mainGuestRoom - the main guest room is the card is of type 'guest'

cardNameID - the type of card

issueTime - the time when the card was made

startTime - the time when the card gets valid

expireTime - the time when the card expires

expireTimeOnline - the time when the card expires (when new time set online)

serialNumber - the serialNumber of the physical RFID card

```
POST /callback
{
    "resources": {
        "cards": {
            "fields": [
                "expireTime",
                "cancelTime",
                "mainGuestRoom",
                "cardNameID"
            ]
        }
    }
}
```

7.14.4 Multiple resources

You can register for multiple resources in the same request.

```
POST /callback
{
    "resources": {
        "events": {
        },
        "cards": {
        }
    }
}
```

7.14.5 Response

If the request succeeds then the response will contain the id of the callback.

```
200 OK
{
    "id": "182e5c2a-99d5-4bd3-8021-07d436b67243"
}
```

7.15 Remove callback

If an existing callback is no longer required, it should be removed to free resources on the server. To remove an existing callback, the id of the callback should be included in a DELETE request.

```
DELETE /callback/<id>
```

7.16 Get callback

Callbacks are handled through long polls. That means the integrator will have to send a GET request on the callback id.

```
GET /callback/<id>
```

If there is something in the callback queue a response will be sent back immediately, else the call will hang for up-to 4 minutes before returning a response (which may be empty if nothing reached the queue in time).

```
GET /callback/dfe8f9ac-22b2-4bed-ad4a-534f2ed1d69a
```

```
200 OK
{
    "id": "dfe8f9ac-22b2-4bed-ad4a-534f2ed1d69a",
    "resources": {
        "cards": [
            {
                "cancelTime": "",
                "cardNameID": 0,
                "expireTime": "20151216T1200",
                "mainGuestRoom": 101
            }
        ]
    }
}
```

8 REFERENCES

[1] ASSA ABLOY Hospitality, *PMS Plus Protocol*, DP-060-026, 2012-12-13.

[2] Fielding, R. et al., *RFC 2616 Hypertext Transfer Protocol – HTTP/1.1*, June 1999.
http://www.ietf.org/rfc/rfc2616.txt

[3] AWS documentation, *Signing and Authenticating REST Requests*, 2013-01-10.
http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAuthentication.html

[4] Berners-Lee, T. et al., *RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax*, January 2005.
http://www.ietf.org/rfc/rfc3986.txt

[5] Krawczyk, Bellare & Canetti, *RFC 2104 - HMAC: Keyed-Hashing for Message Authentication*, February 1997.
http://www.ietf.org/rfc/rfc2104.txt

[6] Burrows, J. et al., *FIPS PUB 180-1 Secure Hash Standard*, 1995-04-17.
http://www.itl.nist.gov/fipspubs/fip180-1.htm

[7] Python Software Foundation, *Python Programming Language - Official Website*, 2013-02-11.
http://www.python.org

9 DOCUMENT HISTORY

Rev	Date	Author	Description
0	2014-03-11	MAx	Initial version released. Supports issuing and modifying guest cards using resources *cards*, *encoders*, *users* and *userGroups*, monitoring alarms using resources *alarms* and *alarmTypes*, monitoring network nodes using the resource *nodes*, and sending opening commands to doors using the resource *doors*.
1	2014-07-03	MAx	Manage blocking groups using the resource *blockingGroups*, auto-update cards on encoder, update joiner cards automatically, allow aborting of encoder operation.
2	2014-12-17	Amarjit	Added 'timeSchedules' resource and create and read operation for staff card.
3	2015-05-22	Amarjit	Added section for how to create mobile key through web api.
4	2015-05-29	Amit	Added section for how to create mobile key with label, description and url meta data.
5	2015-06-05	Amit	Updated description of mobile key description property.
6	2015-06-23	Amarjit	Added 'counters' resource and updated the section users, userGroups and read anc create staff card.
7	2015-09-17	Amarjit	Added example for how to create a new user and how to update the information about an existing user.
8	2015-10-13	Magnus	Added new sections on callback.
9	2016-01-21	Amit	Added 'mobileAccess' resource and updated section 'Create mobile key for guest' to add field 'accountName' and updated examples to show it's usage.
10	2016-03-16	Amarjit	Remove the 's' from usersGroups because in all the requests the actual word do not contain this 's'.
11	2016-05-06	Amarjit	Updated for checkedOut command in door(s).
12	2016-05-25	Magnus	*/sessions/*{*id*} was misspelled.
13	2016-05-25	Amit	Added new user case section 'Sending checkout commands' for checkout commands. Also updated description of corresponding properties.