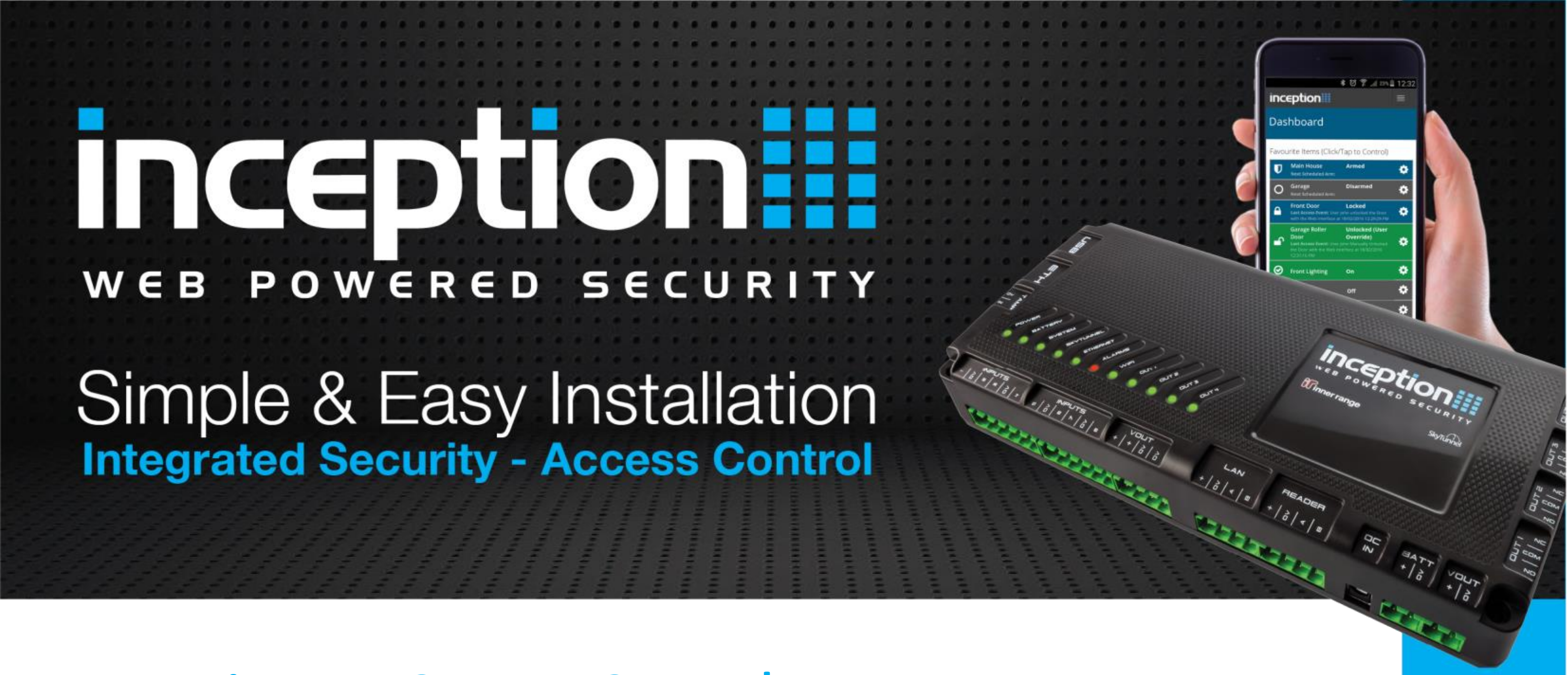

inception
WEB POWERED SECURITY
Simple & Easy Installation
Integrated Security - Access Control

Inception REST API Sample Usage

Contents

Introduction

This document aims to demonstrate some of the various ways the Inception REST API can be used to retrieve data and control entities in the system, with specific examples of common use cases, including the extensive real-time update capabilities available through long polling HTTP requests.

It is recommended that the reader already has some general level of experience with sending HTTP requests, interaction with REST APIs, and JSON serialized data.

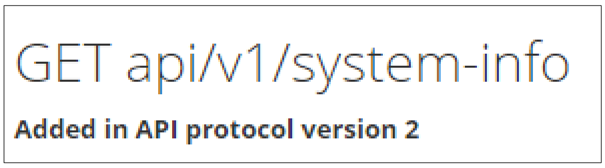
The information in this document is relevant to the Inception REST API protocol **v8** released in Inception firmware version **4.0**, previous firmware versions may not support certain API features used in examples.

The complete documentation for the Inception REST API can be found on-board on any Inception device at the URL "http://[inception-address]/ApiDoc", where [inception-address] is your Inception's hostname or IP address. If you don't have access to an Inception controller, the documentation is also publicly accessible via Skytunnel at https://skytunnel.com.au/Inception/API_SAMPLE/ApiDoc. The REST API documentation also contains a downloadable Postman collection of example API requests, which shows some practical samples of HTTP requests that can be sent to the Inception API.

Retrieving the API Protocol version

Before using the Inception REST API to interact with an Inception device, it is recommended to first query for the device's API protocol version, to ensure that the API methods you are planning to use are supported by the currently loaded firmware version. This is done by sending a GET request to the "**api/protocol-version**" URL on the targeted Inception, and reading the protocol version number in the response. Checking the API protocol version does not require an authenticated session ID in the HTTP request.

Certain API methods are marked in the API documentation with a minimum supported API protocol version like in the following example. Make sure that the Inception device's protocol version is equal to or higher than the minimum protocol versions for the methods you are planning on using, or you won't be able to use those API methods without first upgrading the Inception device's firmware.


GET api/v1/system-info

Added in API protocol version 2

Follow these steps to retrieve the current API protocol version of the device.

1. Send a GET request to "**api/protocol-version**".
2. Depending on how old the Inception firmware is, you may receive a valid JSON response, or a 404 Not Found response if the firmware is too old.
    a. If the response contains a valid JSON response, the "ProtocolVersion" property will contain the API protocol version number. Compare this with the required protocol version of the API methods you are planning to use to see which methods are supported, and which ones are too new (i.e. required version > current protocol version). If certain API methods you wish to use are not supported, you can choose not to use those methods or to upgrade the Inception firmware to a newer version which supports those methods.

```
{
    "ProtocolVersion": 2
}
```

   b. The request may return a 404 Not Found reponse if the firmware is running an early version of the REST API. This may indicate that the firmware is running an early version of the REST API (before the protocol version endpoint was added), or that the firmware does not include the REST API at all. In either case, it is recommended that the device's firmware be upgraded before attempting to use REST API features to ensure maximum compatibility.

Authentication/Logging In

Most of the Inception API methods require an authenticated session ID in the request's Cookie header before the server will respond with the requested data. It is necessary to authenticate with valid credentials before the rest of the API can be used. See the API Overview documentation page for more general information on setting up API User credentials to enable authenticated requests to be made.

1. Send a POST request to "**api/v1/authentication/login**" on the targeted Inception's hostname/IP address with a JSON request body like the following (where [username] and [password] are the web credentials of an API User that exists on the targeted Inception):

```
{
      "Username": "[username]",
      "Password": "[password]"
}
```

2. If the authentication was successful, the server will respond with a JSON object containing a new session ID (the "UserID" field), similar to the following. If not, the Result value will be "Failure", with a reason given (incorrect credentials, system locked out), and no UserID.

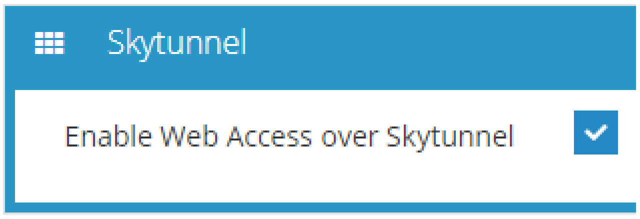
```
{
    "Response": {
        "Result": "Success",
        "Message": "OK",
        "FailureReason": 0
    },
    "UserID": "ab6d680e-e481-4dca-a151-11dbc4c1e2ac"
}
```

3. You can now send authorised REST API requests by putting the response's UserID string into the Cookie header of subsequent requests (i.e. set the Cookie HTTP header value to "**LoginSessId=[sessionID]**"). This session ID will be valid for the next 10 minutes, and will be refreshed for another 10 minutes every time you make a request to the server. If the session expires, you will have to repeat the process to authenticate again and receive a new session ID.
   a. Alternatively, if you are in an environment where you cannot edit HTTP request headers or cookies, the session ID can also be appended to the URL as a query string parameter with every API request you send (e.g. GET "**api/v1/control/area?session-id=[id]**").

Using the REST API over SkyTunnel

The Inception REST API can be accessed over Inner Range's free SkyTunnel service from anywhere in the world if the controller is configured to allow web access over SkyTunnel (enabled by default).

To enable API usage over SkyTunnel, log into the Inception web interface and go to the [Configuration > General > Network] page and make sure the "Enable Web Access over SkyTunnel" checkbox in the "SkyTunnel" section is ticked.



Enable Web Access over Skytunnel ✓

API requests can be made through SkyTunnel simply by changing the root of the URL to "https://www.skytunnel.com.au/inception/[serial]". For example, if your controller's serial number is "**IN001234**", you can retrieve the API protocol version of the controller's firmware over SkyTunnel by sending a GET request to "https://www.skytunnel.com.au/inception/IN001234/api/protocol-version".

NOTE: Due to popularity, Inner Range's free SkyTunnel service has been upgraded to include multiple servers to support increased scale and geolocation optimization.This change affects integrations that communicate with the API over SkyTunnel. Requests sent to a SkyTunnel URL (for example: **https://www.skytunnel.com.au/inception/IN001234**) may respond with a 307 Redirect status code containing the instance-specific URL in the "Location" header. There is no guarantee that an Inception controller will always remain connected to the same SkyTunnel instance, so integrations must make sure that redirect responses are followed correctly.

Requesting Entity Information

Example: Getting Area Information

1. Authenticate as an API User (see "Authentication/Logging In" section), and save the session ID from the Cookie header and use it for the following requests.
2. Send a GET request to "**api/v1/control/area**" (with the authorised session ID in the Cookie header).
3. The server will respond with a JSON object array containing entries for each Area in the Inception system that the API User has permission to interact with. The entities' "ID" field value can be used in API requests to query for more information or to control the Area whose ID it is.

```
[
    {
        "ReportingID": 1,
        "ID": "09492660-72ed-4807-bc59-c1fef83981a5",
        "Name": "Default Area"
    },
    {
        "ReportingID": 2,
        "ID": "26556643-819b-4c41-8e6f-e804114c986b",
        "Name": "Area B"
    }
]
```

4. Alternatively, you can also send a GET request to "**api/v1/control/area/summary**" if you want a combined response containing supplementary information relating to Areas in the system, including the Area's associated inputs, its entry/exit delay times, and its current state.

```
{
    "Areas": {
        "481d30ac-9108-45e9-b8df-80fe98a2e349": {
            "EntityInfo": {
                "ReportingID": 1,
                "ID": "481d30ac-9108-45e9-b8df-80fe98a2e349",
                "Name": "Default Area"
            },
            "AssociatedInputs": [
                {
                    "Input": "07590f0f-e958-4c25-917f-62cf5e46214c",
                    "InputName": "CustomInput 1",
                    "ProcessGroup": "71696099-c2fb-4522-b033-9213c5a8858e"
                }
            ],
            "ArmInfo": {
                "EntryDelaySecs": 45,
                "ExitDelaySecs": 60,
                "DeferArmDelaySecs": 3600,
                "AreaWarnTimeSecs": 60,
                "MultiModeArmEnabled": false
            },
            "CurrentState": 2048
        }
    }
}
```

Controlling Entities

Example: Arming an Area (Standard Mode)

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Find the ID of the Area that you want to arm (see the previous "Getting Area Information" example), the placeholder [areaID] will represent this area's ID in the following steps.
3. Send a POST request to "**api/v1/control/area/[areaID]/activity**" with a JSON request body of:

```
{
    "Type": "ControlArea",
    "AreaControlType": "Arm"
}
```

4. If the Arm activity was successfully created, the server will return a "Success" JSON response with the new Activity's ID present in the body. The returned ActivityID can be used to monitor the progress of the activity (see the "Monitoring the progress of an Area Arm activity" section further below). If the request failed due to insufficient permission or another reason, the response result will be "Failure" and no ActivityID will be returned.

```
{
    "Response": {
        "Result": "Success",
        "Message": "OK"
    },
    "ActivityID": "b1ba25f2-118a-4170-b4d0-57432b6196fc"
}
```

Example: Controlling an Output (turn on)

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Find the ID of the Output that you want to turn on (this can be found by using the "**GET api/v1/control/output**" endpoint); the placeholder [outputID] will represent this output's ID in the following steps.
3. Send a POST request to "**api/v1/control/output/[outputID]/activity**" with a JSON request body of:

```
{
    "Type": "ControlOutput",
    "OutputControlType": "On"
}
```

4. If the activity was successfully submitted, a "Success" JSON response will be returned by the server, and the ActivityID can used to monitor the activity's progress. If the request failed, the response result will be "Failure" and no ActivityID will be returned.

```
{
    "Response": {
        "Result": "Success",
        "Message": "OK"
    },
    "ActivityID": "d9b90072-745b-49eb-be01-10ae9a8c1792"
}
```

Activities

Example: Virtually Badging a Card/Credential at a Door Reader

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Find the ID of the Reader that you want to virtually badge the credential at (this can be found by using the “**GET api/v1/control/door/[id]/attached-readers**” endpoint on the Door that the Reader is attached to); the placeholder [readerID] will represent this reader’s ID in the following steps.
3. Find the Credential Template and Credential Number of the User credential that you want to virtually badge at the reader. Credential data can be retrieved from the User’s “Credentials” property (**GET api/v1/config/user/[id]**). The placeholders [templateID] and [cardNumber] will represent the Credential Template ID and the Card Number respectively in the following steps.
4. Send a POST request to “**api/v1/activity**” with a JSON request body of:

```
{
      "Type": "BadgeCredentialAtReader",
      "CredentialData": {
            "CredentialTemplate": "[templateID]",
            "Data": "[cardNumber]"
      },
      "Entity": "[readerID]"
}
```

5. The access attempt will be carried out by the Inception system as though the credential was physically badged at the reader. If the activity was successfully submitted, a “Success” JSON response will be returned by the server, and the ActivityID can used to monitor the activity’s progress. If the request failed, the response result will be “Failure” and no ActivityID will be returned.

```
{
    "Response": {
        "Result": "Success",
        "Message": "OK"
    },
    "ActivityID": "d9b90072-745b-49eb-be01-10ae9a8c1792"
}
```

Example: Virtually Presenting a User PIN at a Door Reader

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Find the ID of the Reader that you want to virtually badge the credential at (this can be found by using the “**GET api/v1/control/door/[id]/attached-readers**” endpoint on the Door that the Reader is attached to); the placeholder [readerID] will represent this reader’s ID in the following steps.
3. Find the User PIN that you want to virtually present at the door reader. User PINs cannot be retrieved from User data in the REST API as they are write-only for security purposes; you will need to know what the User PIN was set to beforehand. The placeholder [userPin] will represent the user’s PIN in the following steps.
4. Send a POST request to “**api/v1/activity**” with a JSON request body of:

```
{
    "Type": "SendPINDataToReader",
    "Entity": "[readerID]",
    "PINData": "[userPin]"
}
```

5. The access attempt will be carried out by the Inception system as though the PIN was physically entered at the reader. If the activity was successfully submitted, a "Success" JSON response will be returned by the server, and the ActivityID can used to monitor the activity's progress. If the request failed, the response result will be "Failure" and no ActivityID will be returned.

```
{
    "Response": {
        "Result": "Success",
        "Message": "OK"
    },
    "ActivityID": "6a6fa14e-1f99-4483-be32-687c99057f66"
}
```

User Management

Example: Querying for Users

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a GET request to "**api/v1/config/user**". The server will respond with a list of all the Users in the system.

```
[
    {
        "ID": "69f57d11-d153-411e-b963-498d99352df7",
        "Name": "NewTest User",
        "DateTimeUpdated": "2019-05-29T15:05:10.4210209+10:00"
    },
    {
        "ID": "211c9106-3f92-439e-9f6e-32e597456b7b",
        "Name": "John Smith",
        "DateTimeUpdated": "2019-05-29T16:19:42.0570493+10:00"
    }
]
```

3. Optional: By default, only User ID, Name and Last Modified Time properties are returned in the query. Additional properties can be selectively included by setting the "includedProperties" query parameter to the desired properties, e.g. "**api/v1/config/user?includedProperties=Permissions,Credentials**" will include the Permissions and Credentials properties in the JSON response. A list of all available property names can be found here. Additionally, the results can be filtered to only include Users with new changes since a certain time, in the case where an integration needs to periodically sync User data from Inception but only requires the latest changes. This can be done with the "modifiedSince" query parameter, like "**api/v1/config/user?modifiedSince=2019-05-29T15:00:00**".

Example: Getting a single User's Data

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a GET request to "**api/v1/config/user/[userID]**", where userID is the ID of the User whose configuration data you want to retrieve.
3. The server will respond with a JSON object (API docs link) containing the User's data (example partially shortened).

```
{
    "DateTimeCreated": "2019-05-20T13:26:46.7984609+10:00",
    "DateTimeUpdated": "2019-05-28T14:30:28.260618+10:00",
    "ID": "448f141c-8658-4eaf-906d-22a644cc8ba2",
    "Name": "API User",
    "Notes": "",
    "EmailAddress": "",
    "SecurityPin": "",
    "Permissions": [
        {
            "Item": "09492660-72ed-4807-bc59-c1fef83981a5",
            "Allow": true
        },
        …
    ],
    …
    "UserCancelled": false
}
```

Example: Creating a New User

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a POST request to "**api/v1/config/user**" with a JSON request body similar to the following example, for a User named "John Smith" who has a PIN of 1234 and will expire on 28 May 2019 5:00PM server time (any properties not supplied in the JSON object will be filled in with default values).

```
{
     "Name": "John Smith",
     "Notes": "Sample user",
     "SecurityPin": "1234",
     "EnableExpiry": true,
     "ExpireAfter": "2019-05-28T17:00:00"
}
```

3. You should receive a response from the server containing the entire new JSON User object if the User was successfully created. If there was an error creating the User, due to duplicate Security PIN, invalid data, etc., the server will respond with an error and a text description of the problem.

```
{
    "DateTimeCreated": "2019-05-29T15:41:58.2848989+10:00",
    "DateTimeUpdated": "2019-05-29T15:41:58.2848989+10:00",
    "ID": "f8eecdc9-5088-42cc-b63d-230eb091c847",
    "Name": "John Smith",
    "Notes": "Sample user",
    "EmailAddress": null,
    "SecurityPin": "********",
    "Permissions": [],
    "UseExtendedUnlockTimes": false,
    "Credentials": [],
    "RemoteFobs": [],
    "TerminalProfile": "fe4e00f4-24b2-4dda-9355-761d37448650",
    "WebLoginEnabled": false,
    "WebLoginUsername": null,
    "WebLoginPassword": null,
    "WebPagePermissions": "0e44dc2c-b0d2-4fa2-80af-f5d076f090e9",
    "PermanentCache": false,
    "EnableExpiry": true,
    "ValidFrom": "0001-01-01T00:00:00",
    "ExpireAfter": "2019-05-28T17:00:00",
    "WhenToCancel": 0,
    "UserCancelled": false
}
```

Example: Deleting a User

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a DELETE request to "**api/v1/config/user/[userID]**" where [userID] is the ID of the User you want to delete.
3. The server will respond with an empty 200 OK response if the deletion was successful, or a 404 Not Found if the user did not exist.

Example: Updating User Data (full overwrite)

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a PUT request to "**api/v1/config/user/[userID]**" where [userID] is the ID of the User whose data you want to replace, with a JSON request body containing the replacement User data. **NOTE:** All User fields that are omitted from the JSON request body will be reset to default values when overwriting, if you want to retain existing field values while updating only specific data fields, see the "Patching User Data" example.
3. The server will respond with an empty 200 OK response if the update was successful, or a description of the error if the update failed.

Example: Patching User Data (partial update)

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a PATCH request to "**api/v1/config/user/[userID]**" where [userID] is the ID of the User you want to patch, with a JSON request body containing the patch data. The example below will assign a new Notes field value, a new Permission Group (where [permissionGroupID] is a valid Permission Group guid) and Security PIN to the User, while leaving the rest of the User's data unchanged:

```
{
     "Notes": "User has been modified",
     "SecurityPin": "5678",
     "Permissions": [
          {
               "Item": "[permissionGroupID]"
          }
     ]
}
```

3. The server will respond with an empty 200 OK response if the patch request was successful, or a description of the error if the request failed.

Example: Adding or Updating a User Photo (new in protocol version 8)

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a POST request to "**api/v1/config/user/[userID]/photo**" where [userID] is the ID of the User you want to upload the photo for. The POST request should contain a multipart/form-data type body which contains the image file contents. The method to add a multipart/form-data body may differ depending on

the HTTP client you are using, but the raw HTTP body should look similar to the following example if you are uploading a PNG file:

```
POST /api/v1/config/user/{id}/photo HTTP/1.1
Accept: application/json
Cookie: LoginSessId=(removed)
User-Agent: (removed)
Host: (removed)
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Type: multipart/form-data; boundary=--------------------------
459156084925464413565125
Content-Length: 1340824

----------------------------459156084925464413565125
Content-Disposition: form-data; name="image"; filename="UserPhoto.png"
Content-Type: image/png

PNG


(raw image bytes here)
```

3. The server will respond with an empty 200 OK response if the photo upload was successful, or a description of the error if the request failed.

Example: Deleting a User Photo (new in protocol version 8)

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a DELETE request to "**api/v1/config/user/[userID]/photo**" where [userID] is the ID of the User whose photo you want to delete.
3. The server will respond with an empty 200 OK response if the request was successful, or a description of the error if the request failed.

Example: Generating an unused PIN for a user (new in protocol version 11)

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a GET request to "api/v1/config/user/generate-unused-pin", with optional values for the minLength/maxLength query parameters if desired. If a custom min/max PIN length is specified, it must be within the system's configured PIN length range ("Access Configuration" section of the Configuration > General > System page in the web interface), or the request will be rejected as invalid.
3. The server will respond with a 200 OK response similar to the following if a PIN was successfully generated, or a 500 error if there was a problem generating the PIN:

```
{
    "Success": true,
    "Pin": "71400526",
    "Error": ""
}
```

4. The generated PIN can now be used for user creation or editing (see the Creating a New User example for more information).

Review Event Queries

Example: Retrieve the oldest 50 Review Events

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a GET request to "**api/v1/review?limit=50**". The "limit" query parameter sets the maximum number of events the server will return, and the "offset" parameter can be used to skip a number of events, e.g. for paginated results. If no value is given for the "limit" parameter, the default limit is set to 100. The "dir" parameter can also be used to choose where to start ("dir=asc" for ascending/oldest events first, "dir=desc" for descending/newest events first), results are returned in ascending order by default.
3. The server will respond with the oldest 50 Review Events (example results shortened):

```
{
    "Offset": 0,
    "Count": 50,
    "Data": [
        {
            "ID": "bd678364-8221-4410-bf61-fec872c90499",
            "Description": "System Started",
            "MessageCategory": 1,
            "When": "2019-05-30T15:08:56.7991171+10:00",
            "WhenTicks": 636947897367991171,
            "Who": "",
            "WhoID": "00000000-0000-0000-0000-000000000000",
            "What": "No Reset Reason (Windows)",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        {
            "ID": "ca96e12f-a20b-444b-aab9-ade452008c60",
            "Description": "System Application Version",
            "MessageCategory": 2,
            "When": "2019-05-30T15:08:56.8061178+10:00",
            "WhenTicks": 636947897368061178,
            "Who": "",
            "WhoID": "00000000-0000-0000-0000-000000000000",
            "What": "1.0.0.0",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        …
      ]
}
```

4. The next contiguous block of Review Events can be retrieving by making the same request except increasing the number of the "offset" query parameter by the previous amount of the "limit" parameter, e.g. "**api/v1/review?offset=50&limit=50**" to retrieve the second-oldest block of 50 events.

Example: Retrieve Access and Security-related Events created between two Dates

1. Authenticate as API User, save session ID from cookie and use it for future requests.

2. Send a GET request similar to "**api/v1/review?categoryFilter=Access,Security&start=2019-04-21T09:00:00&end=2019-04-25T17:00:00**" for events between 21 April 9:00AM and 25 April 5:00PM, for example.
    a. The "categoryFilter" query parameter determines which categories of events are returned, using a list of comma-separated category names. The list of possible categories can be found here.
    b. The "start" and "end" parameters are ISO 8601 formatted date/time values specifying the query range. These parameters can be used when the search direction is descending, except the "start" must be more recent than the "end" as results are returned in reverse.
3. The server will respond with the first 100 (the default query limit) Access and Security-related events from the given time period. If there are more than 100 events in the period, you can send the request again, but with an increased "offset" parameter value to fetch the next block of results.

```
{
    "Offset": 0,
    "Count": 100,
    "Data": [
        {
            "ID": "4e7e499c-d41d-48e7-b0aa-7021f0e8644c",
            "Description": "Web Login was Successful by User",
            "MessageCategory": 141,
            "When": "2019-05-30T15:31:57.4650017+10:00",
            "WhenTicks": 636947911174650017,
            "Who": "Installer",
            "WhoID": "fbec7c22-500d-4f3b-b94e-4c19ff501f9f",
            "What": "",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        {
            "ID": "7aeed56c-bb69-4b08-9576-e300e7b127c8",
            "Description": "Item Created",
            "MessageCategory": 1500,
            "When": "2019-05-30T15:33:51.0470017+10:00",
            "WhenTicks": 636947912310470017,
            "Who": "System",
            "WhoID": "00000000-0000-0000-0000-000000000000",
            "What": "User - Beau Munoz",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        …
      ]
}
```

Example: Retrieve User Area Arm/Disarm Events created between two Dates

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Send a GET request similar to "**api/v1/review?messageTypeIdFilter= 5000,5201&start=2019-04-21T09:00:00&end=2019-04-25T17:00:00**" for events between 21 April 9:00AM and 25 April 5:00PM, for example.

a. The “messageTypeIdFilter” query parameter determines which event type IDs are returned in the query, specified by a comma-separated list of integer values. A list of all message type IDs can be found here.

3. The server will respond with a block of up to 100 events that match the filter and the time period. If there are more than 100 events in the period, you can send the request again, but with an increased “offset” parameter value to fetch the next block of results.

```
{
    "Offset": 0,
    "Count": 100,
    "Data": [
        {
            "ID": "16c1e41e-43de-4af0-9307-152a85122e10",
            "Description": "Area Armed by User",
            "MessageCategory": 5000,
            "When": "2019-06-03T13:31:15.7335622+10:00",
            "WhenTicks": 636951294757335622,
            "Who": "Installer",
            "WhoID": "fbec7c22-500d-4f3b-b94e-4c19ff501f9f",
            "What": "Default Area",
            "WhatID": "481d30ac-9108-45e9-b8df-80fe98a2e349",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        {
            "ID": "b4b122c8-aab9-4436-a5ce-f2f49c150c24",
            "Description": "Area Disarmed by User",
            "MessageCategory": 5201,
            "When": "2019-06-03T13:31:16.794059+10:00",
            "WhenTicks": 636951294767940590,
            "Who": "Installer",
            "WhoID": "fbec7c22-500d-4f3b-b94e-4c19ff501f9f",
            "What": "Default Area",
            "WhatID": "481d30ac-9108-45e9-b8df-80fe98a2e349",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        …
    ]
}
```

Example: Retrieve all Events involving a certain User

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Find the ID of the User you are interested in retrieving events for , and send a GET request similar to “**api/v1/review?involvedEntityIdFilter=[userID]&start=2019-04-21T09:00:00&end=2019-04-25T17:00:00**” (where “userID” is the ID of the User in question) for events between 21 April 9:00AM and 25 April 5:00PM, for example.
    a. The “involvedEntityIdFilter” query parameter allows you to filter event queries to include only those which reference particular entity IDs (Users, Areas, Doors, etc.) in their WhoID, WhatID or WhereID fields. Multiple IDs can be included as a comma-separated list if needed.
3. The server will respond with a block of up to 100 events that match the filter and the time period. If there are more than 100 events in the period, you can send the request again, but with an increased

“offset” parameter value to fetch the next block of results. (Example response with [userID] of “fbec7c22-500d-4f3b-b94e-4c19ff501f9f”).

```
{
    "Offset": 0,
    "Count": 100,
    "Data": [
        {
            "ID": "46a9048d-1f79-4cc8-b805-e7bdd78955a5",
            "Description": "Item Changed",
            "MessageCategory": 1501,
            "When": "2019-06-03T13:30:35.7834783+10:00",
            "WhenTicks": 636951294357834783,
            "Who": "Installer",
            "WhoID": "fbec7c22-500d-4f3b-b94e-4c19ff501f9f",
            "What": "User - Installer",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        {
            "ID": "16c1e41e-43de-4af0-9307-152a85122e10",
            "Description": "Area Armed by User",
            "MessageCategory": 5000,
            "When": "2019-06-03T13:31:15.7335622+10:00",
            "WhenTicks": 636951294757335622,
            "Who": "Installer",
            "WhoID": "fbec7c22-500d-4f3b-b94e-4c19ff501f9f",
            "What": "Default Area",
            "WhatID": "481d30ac-9108-45e9-b8df-80fe98a2e349",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        },
        …
    ]
}
```

Example: Retrieve First 50 Events older than a Reference Event

1. Authenticate as API User, save session ID from cookie and use it for future requests.
2. Retrieve a review event to use as a reference event by following one of the previous examples, and store the ID and WhenTicks values of the event.
3. Send a GET request to “**api/v1/review?limit=50&dir=desc&referenceId=[msgId]&referenceTime=[msgTime]**”, where **[msgId]** is the ID of the reference event and **[msgTime]** is the WhenTicks value of the reference event.
4. The server will respond with a block of up to 50 events that occurred before the reference event, ordered from newest to oldest.

```
{
    "Offset": 0,
    "Count": 50,
    "Data": [
        {
            "ID": "9d9c9bfa-c3bb-4d4e-aa7a-a9f43f5c5952",
            "Description": "Area Event Created",
            "MessageCategory": 5505,
            "When": "2019-07-10T10:07:04.2642453+10:00",
            "WhenTicks": 636983140242642453,
            "Who": "",
            "WhoID": "00000000-0000-0000-0000-000000000000",
            "What": "Armed",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "Area 3",
            "WhereID": "7419bc70-8a7d-4f4b-8d18-fd1c7f734202"
        },
        ...
    ]
}
```

Introduction to Long Polling

Inception’s REST API uses HTTP long polling to deliver real-time updates for entity states, review events, activity progress, etc. without the need to manually poll for updates. HTTP long polling involves opening a HTTP request connection to a certain endpoint on the server as normal, but with the expectation that the server will not necessarily respond immediately; rather, it will hold the request open until the relevant response data becomes available (e.g. new state changes or new events). In Inception’s case, requests are held open for up to a minute, at which point the server will reply with an empty response if no new information became available in that minute. From there, the client can re-send the request with the same JSON body contents to continue waiting for updates. Using this technique removes the need to potentially make a multitude of expensive manual polling requests for state information from the server.

Inception’s long poll requests are repeatable so that information is not lost in between requests, for example: a request for all area state updates since a given timestamp will always return with every area whose state has changed since that timestamp.

Typical usage of the API will usually involve having at least 1 HTTP long poll request open waiting for updates (entity states, review events), while other synchronous requests (e.g. retrieving Area/User information, sending control commands) are sent and processed on a different thread while the long poll request is waiting to return.

Multiple sub-requests can be bundled into a single HTTP request body to save having to leave multiple requests hanging open. For example, you could have 3 separate sub-requests to monitor Area, Door, and Output states and combine them into the body of a single HTTP long polling request to the “**api/v1/monitor-updates**” endpoint on an Inception device, and the request will return with updates as soon as they become available for any of the 3 sub-requests. For an example of this usage, see the “Monitoring Area states and Output states” section below.

For details on the JSON data types used in Inception’s Update Monitor/long polling requests, see the Update Monitor page of the API documentation.

Monitoring for Long Poll Updates

Example: Monitoring Area States

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Send a POST request to “**api/v1/monitor-updates**” with a JSON request body of:

```
[
    {
        "ID": "Monitor_AreaStates",
        "RequestType": "MonitorEntityStates",
          "InputData": {
                "stateType": "AreaState",
                "timeSinceUpdate": "0"
          }
     }
]
```

3. Hold the request open for up to 60 seconds until a response is received from the server.
    a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events (i.e. go back to step 2).
    b. If the response body contains data, continue to step 4.
4. Read the array of state updates in the response (in the Result.stateData field) and process the data as needed. The sample response will look similar to the JSON snippet below. The “PublicState” field contains the set of bit flags that represent the Area’s current state, the “Info1” field contains information about the Area’s next scheduled arm time (if applicable), and the “Info2” field contains the number of Users that are currently in the Area. See the API Docs for information on how to interpret the Area’s state flags.

```
{
    "ID": "Monitor_AreaStates",
    "Result": {
        "updateTime": "16158053",
        "stateData": [
            {
                "ID": "09492660-72ed-4807-bc59-c1fef83981a5",
                "stateValue": 8,
                "PublicState": 2048,
                "Info1": null,
                "Info2": "0"
            }
        ]
    }
}
```

5. To continue monitoring for newer area state updates, take the value from the “updateTime” field of the response (“16158053” in this example) and use it to replace the value of the “timeSinceUpdate” field of your previous request’s JSON body, and send the new request. The request will be handled in the same way as the previous one, except it will only monitor for updates that are newer than this timestamp. Sending new long poll requests in this way will ensure that new updates are delivered contiguously in chronological order.

Example: Monitoring User States & Locations

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Send a POST request to “**api/v1/monitor-updates**” with a JSON request body of:

```
[
     {
          "ID": "Monitor_UserStates",
          "RequestType": "MonitorEntityStates",
          "InputData": {
                "stateType": "UserState",
                "timeSinceUpdate": "0"
          }
```

```
        }
]
```

3. Hold the request open for up to 60 seconds until a response is received from the server.
    a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events (i.e. go back to step 2).
    b. If the response body contains data, continue to step 4.
4. Read the array of state updates in the response (in the Result.stateData field) and process the data as needed. The sample response will look similar to the JSON snippet below. The "Info1" field contains the name of the User's current location if applicable. The User's PublicState flags can be interpreted with the API documentation page for UserPublicStates.

```
{
    "ID": "Monitor_UserStates",
    "Result": {
        "updateTime": "21884069",
        "stateData": [
            {
                "ID": "b4d4a1e5-a848-4535-86b9-687be491614a",
                "stateValue": -1,
                "PublicState": 0,
                "Info1": "Lobby",
                "Info2": ""
            },
            {
                "ID": "8410f634-07e7-4116-8ed1-ca73ea7c78f3",
                "stateValue": -1,
                "PublicState": 0,
                "Info1": "",
                "Info2": ""
            }
        ]
    }
}
```

5. Take the timestamp value from the response's "updateTime" field and put it in the original request's "timeSinceUpdate" field, and repeat from step 2 to wait for new updates for as many times as desired.

Example: Monitoring both Area states and Output states

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Send a POST request to "**api/v1/monitor-updates**" with a JSON request body containing 2 sub-requests like the following. **NOTE:** The "ID" field of each JSON sub-request object must be unique when monitoring multiple types of updates from a single HTTP request, in order to determine which server responses correspond to which sub-request.

```
[
        {
                "ID": "Monitor_AreaStates",
                "RequestType": "MonitorEntityStates",
                "InputData": {
                        "stateType": "AreaState",
                        "timeSinceUpdate": "0"
                }
        },
      {
            "ID": "Monitor_OutputStates",
            "RequestType": "MonitorEntityStates",
            "InputData": {
                  "stateType": "OutputState",
                  "timeSinceUpdate": "0"
            }
      }
]
```

3. Hold the request open for up to 60 seconds until a response is received from the server.
    a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events (i.e. go back to step 2).
    b. If the response body contains data, continue to step 4.
4. Read the "ID" field of the response to determine which sub-request this update is relevant to (in this example, the response was for the "Monitor_AreaStates" request). The response body will only ever contain a response to a single sub-request. Read the array of state updates in the response (in the Result.stateData field) and process the data as needed.

```
{
    "ID": "Monitor_AreaStates",
    "Result": {
        "updateTime": "8128806",
        "stateData": [
            {
                "ID": "26556643-819b-4c41-8e6f-e804114c986b",
                "stateValue": 8,
                "PublicState": 2048,
                "Info1": null,
                "Info2": "0"
            },
            {
                "ID": "09492660-72ed-4807-bc59-c1fef83981a5",
                "stateValue": 4,
                "PublicState": 2064,
                "Info1": "Area will automatically Arm at …",
                "Info2": "0"
            }
        ]
    }
}
```

5. After processing the response data, take the "updateTime" field value from the response and replace the "timeSinceUpdate" field value in the relevant sub-request ("Monitor_AreaStates" in this example), before resending the whole request again in a new HTTP long poll request. Updating the specific sub-request parameter fields in this way after each response allows each one to be processed independently, but combined in a single HTTP request to avoid the overhead of waiting simultaneously on multiple requests to return.

Example: Monitoring the progress of an Area Arm activity

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Start an Area Arm activity (see the "Arming an Area" example) and retrieve the activity ID (will be referred to as "[activityID]").

3. Send a POST request to “**api/v1/monitor-updates**” with a JSON request body of:

```
[
    {
        "ID": "Monitor_ArmActivity",
        "RequestType": "ActivityProgress",
        "InputData": {
            "activityId": "[activityID]",
            "receivedMsgs": ""
        }
    }
]
```

4. Hold the request open for up to 60 seconds until a response is received from the server.
    a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events (i.e. go back to step 3).
    b. If the response body contains data, continue to step 5.
5. Process the activity update messages (in the Result.Messages field). The “Type” sub-field in each Message object is an ActivityProgressState enum, where a value of 2 or 3 (Success or Failure respectively) indicates that the activity has run to completion and will not need to be monitored for any future messages.

```
{
    "ID": "Monitor_ArmActivity",
    "Result": {
        "ActivityId": "[activityID]",
        "AllSentMsgIds": "311d7547-1571-464e-91a2-34489a7b71a9",
        "Messages": [
            {
                "Type": 1,
                "Message": "Area arming process started",
                "Current": 1,
                "Total": 5,
                "Data": null
            },
            {
                "Type": 1,
                "Message": "Arming Area(s)",
                "Current": 4,
                "Total": 5,
                "Data": null
            },
            {
                "Type": 2,
                "Message": "Areas armed",
                "Current": 1,
                "Total": 1,
                "Data": null
            }
        ]
    }
}
```

6. If there are more messages to come for this activity (i.e. a “Finished” message has not been received), take the value of the “AllSentMsgIds” field in the response and place it in the “receivedMsgs” field of

your original request. Repeat from step 3 again with the updated request body, until the activity is completed and has no more progress messages to send.

Example: Monitoring real-time Review Events

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Before sending the long poll request, get the most recent review event ID to use as a reference point to only receive newer events, by sending a GET request to “**api/v1/review? dir=desc&limit=1**”. Record the event’s ID and WhenTicks fields.

```
{
    "Offset": 0,
    "Count": 1,
    "Data": [
        {
            "ID": "604fc423-891c-4d66-abaa-9576ca58b3b8",
            "Description": "Alarm Event Failed…",
            "MessageCategory": 5801,
            "When": "2019-07-10T10:07:04.2642453+10:00",
            "WhenTicks": 636983140242642453,
            "Who": "",
            "WhoID": "00000000-0000-0000-0000-000000000000",
            "What": "Area Open/Close",
            "WhatID": "00000000-0000-0000-0000-000000000000",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        }
    ]
}
```

3. Send a POST request to “**api/v1/monitor-updates**” with a JSON request body similar to the following, where [eventId] and [eventTime] represent the previously retrieved event’s ID and WhenTicks values respectively:

```
[
     {
          "ID": "Monitor_ReviewEvents",
          "RequestType": "LiveReviewEvents",
          "InputData": {
               "referenceId": "[eventId]",
               "referenceTime": "[eventTime]"
          }
     }
]
```

4. Hold the request open for up to 60 seconds until a response is received from the server.
   a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events .
   b. If the response body contains data, continue to step 5.
5. Process the array of review messages (in the Result field).

```
{
    "ID": "Monitor_ReviewEvents",
    "Result": [
        {
            "ID": "a3c50fcf-5797-4964-8b19-cb46e7a57a71",
            "Description": "Area Disarmed by User",
            "MessageCategory": 5201,
            "When": "2019-07-10T10:34:29.9797368+10:00",
            "WhenTicks": 636983156699797368,
            "Who": "Installer",
            "WhoID": "fbec7c22-500d-4f3b-b94e-4c19ff501f9f",
            "What": "Area 3",
            "WhatID": "7419bc70-8a7d-4f4b-8d18-fd1c7f734202",
            "Where": "",
            "WhereID": "00000000-0000-0000-0000-000000000000"
        }
    ]
}
```

6. If you want to poll for more review events, send the POST request again like in step 3, but update the [eventId] and [eventTime] parameters to match the ID and WhenTicks value of the most recently received event. Repeat this process for as long as you want to monitor for new events.

Example: Monitoring real-time Review Events (only Security and Access events)

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Before sending the long poll request, get the most recent review event ID to use as a reference point to only receive newer events (see step 2 of the "Monitoring real-time Review Events" example).
3. Send a POST request to "**api/v1/monitor-updates**" with a JSON request body similar to the following, where [eventId] and [eventTime] represent the previously retrieved event's ID and WhenTicks values respectively. The allowed category types for the "categoryFilter" field can be found on the Update Monitor API docs page.

```
[
     {
          "ID": "Monitor_ReviewEvents",
          "RequestType": "LiveReviewEvents",
          "InputData": {
               "referenceId": "[eventId]",
               "referenceTime": "[eventTime]"
               "categoryFilter": "Security,Access"
          }
     }
]
```

4. Hold the request open for up to 60 seconds until a response is received from the server.
    a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events .
    b. If the response body contains data, continue to step 5.
5. Process the array of review messages (in the Result field).

```
{
    "ID": "Monitor_ReviewEvents",
    "Result": [
        …
    ]
}
```

6. Re-send the POST request with the newest event ID and WhenTicks value in the referenceId and referenceTime fields if you want to retrieve later events.

Example: Monitoring real-time Review Events for a specific Item

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Before sending the long poll request, get the most recent review event ID to use as a reference point to only receive newer events (see step 2 of the "Monitoring real-time Review Events" example).
3. Send a POST request to "**api/v1/monitor-updates**" with a JSON request body similar to the following, where [eventId] and [eventTime] represent the previously retrieved event's ID and WhenTicks, and [itemId] represents the ID of the Area/Door/User/etc. you wish to filter the events for. You can also specify multiple item IDs by passing them in the form of a comma-separated string.

```
[
      {
            "ID": "Monitor_ReviewEvents",
            "RequestType": "LiveReviewEvents",
            "InputData": {
                  "referenceId": "[eventId]",
                  "referenceTime": "[eventTime]"
                  "involvedEntityIdFilter": "[itemId]"
      }
]
```

4. Hold the request open for up to 60 seconds until a response is received from the server.
   a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events .
   b. If the response body contains data, continue to step 5.
5. Process the array of review messages (in the Result field).

```
{
    "ID": "Monitor_ReviewEvents",
    "Result": [
        …
    ]
}
```

6. Re-send the POST request with the newest event ID and WhenTicks value in the referenceId and referenceTime fields if you want to retrieve more events.

Example: Monitoring specific Review Event types

1. Authenticate API User, save session ID from cookie and use it for future requests.
2. Before sending the long poll request, get the most recent review event ID to use as a reference point to only receive newer events (see step 2 of the "Monitoring real-time Review Events" example).
3. Send a POST request to "**api/v1/monitor-updates**" with a JSON request body similar to the following, where [eventId] and [eventTime] represent the previously retrieved event's ID and WhenTicks, and the messageTypeIdFilter field contains a comma-separated list of event type IDs that you want to filter events for. This example is using **2006** (Door Access Granted for User) and **2007** (Door User Access Denied – No Permission); the full list of event type IDs can be found in the API Docs. Optionally, the "involvedEntityIdFilter" parameter can also be included to further filter the events to ones involving specific Items.

```
[
      {
            "ID": "Monitor_ReviewEvents",
            "RequestType": "LiveReviewEvents",
            "InputData": {
                  "referenceId": "[eventId]",
                  "referenceTime": "[eventTime]"
                  "messageTypeIdFilter": "2006,2007"
      }
]
```

4. Hold the request open for up to 60 seconds until a response is received from the server.
   a. If the response body is empty (no new updates to report), you can re-send the request unmodified to wait for new events.
   b. If the response body contains data, continue to step 5.
5. Process the array of review messages (in the Result field).

```
{
    "ID": "Monitor_ReviewEvents",
    "Result": [
        …
    ]
}
```

6. Re-send the POST request with the newest event ID and WhenTicks value in the referenceId and referenceTime fields if you want to retrieve more events.