General integration guide

Overview

The Argus REST API consists of a number of API modules, for different parts of the system.

Each module has a name and a version, e.g. `/events/v1` or `/case/v2`.

Compatibility and Deprecation

When an API module is publicly released, the request and response model are guaranteed to stay consistent and backwards compatible.
Breaking changes will not be done in a released version, instead we will release a new version of the module, which may have a different structure and contract.

We *may* add fields and functionality to a published module by adding endpoints or adding input/output fields, but that will always be done in a backwards compatible manner.

Deprecation and changes will be announced on the mailinglist argus-api-changes@mnemonic.no. Please contact mnemonic to be added to this mailinglist.
In addition, deprecations will be announced on the API documentation page for the relevant API module.

Authentication and API-keys

To use the API, the user has to obtain an API-key. API-keys should be issued for each use case, and if possible restricted to the roles/customers needed only for that use case.

API-keys are based on user permissions, so any API-keys permissions will be constrained by the permissions of its user.
In addition, each API-key can be constrained to a subset of roles and/or customers.

An API-key is constrained both on *time* and *source network,* e.g. can be valid for 3 months, and only valid from one single IP or set of IP networks/addresses.
API-keys which have expired on time can be renewed for another time period.

Obtaining an API-key

Users can create or renew their own API-keys in the Argus GUI under User Preferences, if API-key usage has been enabled for their user.
If API-key usage is not enabled for your user, contact mnemonic.

To obtain unconstrained API-keys (not limited in time and/or IP-range), this has to be done by an administrator, and is normally not granted for end users.

Using an API-key

To use an API-key, add the API-key as a HTTP header `Argus-API-Key.`

Using e.g. CURL, this can be done e.g.

```
curl --header "Argus-API-Key: 1/2/mysecretapikey" https://api.mnemonic.no
/assets/v1/host
```

Using an invalid or expired API-key will cause Argus to respond with a *401 Unauthorized* message.
If the key is expired, or used from the wrong source IP, the response should contain an error message to that effect.

Session authentication

If needing to run a REST client with an authenticated session (e.g. to run with higher Security Level than given by use of API-key), the REST client has to perform a session authentication.

To do this, choose an authentication method which is

- enabled for your user (see https://api.mnemonic.no/currentuser/v1/authmethods)
- available from your current location (see https://api.mnemonic.no/authentication/v1/methods)

The documentation for authenticating using this method can be found in swagger, under `authentication/v1/method/authenticate.`
When successful, the authentication request will set the cookie `argus_session`. By including this cookie in the subsequent requests, the created session will be used.
When using session authentication, the client should delete the session at the end, using

```
DELETE https://api.mnemonic.no/authentication/v1/logout
```

Access restrictions

Each operation towards Argus is restricted by Access Control rules. If the requested operation violates an Access Control rule, the user will be rejected with a *403 Access Denied*.
This may be because the operation is not permitted for the requested object, or because the operation is not permitted for the user at all.

In addition, some operations are not permitted to be used using API-key, because they require a higher Security Level.

To use privileged operations, the user needs to have such permissions enabled, and use a session authentication with a higher Security Level. Contact mnemonic for details.

API documentation

All published endpoints are documented using Swagger. The documentation can be seen here.

If logged in, the endpoints can be used directly in Swagger without using an API-key, since the request will use the current session.

If entering an API-key in the swagger API-key field, this API-key will be used for all requests.
Please note that due to the fact that API-keys have a lower Security Level than two-factor sessions, some operations may be rejected using API-key even though they worked using the logged-in session.

Thus, all testing to create scripts based on API-keys should be done using API-keys also in Swagger.

General endpoint structure

The Argus API uses a REST structure, with the general layout:

Result set

All endpoints have a common result set format on the form

```
{
  "responseCode" : 200,
  "limit" : 25,
  "offset" : 0,
  "count" : 113,
  "size" : 25,
  "data" : [...]
}
```

The value `data` contains the real result from the endpoint, which may be a list or a single object, depending on the endpoint.
For list result endpoints, an empty result will be returned as an empty list. For a single result endpoints, an empty result may return data value `null`, or return error code 404 if the endpoint is not valid.

For searching endpoints, the value `count` will give the total number of results for the given filtering parameters. The `size` will simply state the number of values returned in a list result. For a single result, size should be 0.
If counting is not implemented (such as for single object endpoints), the value of `count` will be 0.
If counting takes too long time, it may time out, in which case the value of `count` will be -1.

Info objects

Returned objects that have a reference to other objects, typically use *Info objects* to provide information about the referenced objects.

An info object is a reduced version of the referenced object, typically containing the ID and the key properties (name, customer) of the referenced object.

Clients which need to show more of the referenced object than what is available in the info object, should fetch that object from its own endpoint. If this is done a lot, it may be wise to implement a cache for this purpose.

Error codes

There are some general error codes used across all endpoints:

- 200 - OK - No error, request completed successfully
- 201 - Created - No error, request completed successfully, creating a new entity
- 401 - Unauthorized - The session or API-key provided is not valid, so the service cannot validate the request to any users permissions
- 403 - Access denied - The current session or API-key is not permitted access to the requested operation. This may be a specific constraint (no access to the specified object), or a general constraint (no access to the requested operation at all)
- 404 - Not found - If requesting a specific resource, the system may return 404 if this resource is not found / not available. When listing/filtering entities, the result will be empty if no matching values are found.
- 405 - Method not allowed - If requesting an endpoint with unsupported content type specifications, this may return error 405. Make sure the content-type header is correctly set.
- 412 - Incorrect parameters - This code may be used to indicate that the request is invalid, either because some constraint is not satisfied, or because some of the provided parameters are invalid. See Parameter errors.

In addition, endpoints may have their specific error codes, which are documented in the API-docs.

Parameter errors

If request contains invalid/wrong parameters, the service will return a 412 error code.
The result set will then contain a "field error" message:

```
{
  "responseCode" : 412,
  ...
  "messages": [
    {
      "message": "Invalid value for field",
      "messageTemplate": "invalid.field.value",
      "type": "FIELD_ERROR",
      "field": "fieldname",
      "parameter": null,
      "timestamp": 1491810313412
    }
  ],
  "currentPage": 0
}
```

If the error is more general, and not associated to a specific field (such as a specific constraint encountered when processing the request), the result set will contain a "action error" message and no specific field name.

Retrieving objects

GET operations are guaranteed read-only, meaning that they will not change data in any way.
Endpoints retrieving single entities, or lists of entities are using the GET method.
The general URL structure for this is

`GET /module/version/entity?queryParameters`
which will list all available entities of this type, optionally filtered by the provided parameters

or

`GET /module/version/entity/id`
which will fetch a single entity with the given ID.

Advanced search

The exception is advanced search endpoints which accept a JSON body with advanced search criteria, these are using POST.
The general URL structure for the search endpoints are

`POST /module/version/entity/search`

with a POST body containing the search criteria

```
{
  "field" : [ "value1", "value2"...],
  ...
}
```

Adding new objects

Endpoints for adding new data are using the POST method. The general URL structure for

`POST /module/version/entity`

with a JSON body defining the new object:

```
{
  "field" : "value",
  "booleanField" : true,
  "otherBooleanField" : false,
  "collectionField" : ["val1","val2"],
  ...
}
```

If creating the object violates any constraints (such as name uniqueness etc), the operation will fail with HTTP 412 (Parameter error).

Add-endpoints return the created object, which also includes the "id"-field of the created object.

On objects with "created" or "lastUpdated"-fields, these fields are automatically set by the service.

See API documentation for JSON structure.

Updating objects

Endpoints for updating existing data are using the PUT method. The general URL structure for this is

`PUT /module/version/entity/id`

with a JSON body defining the new object.

```
{
  "field" : "updated value",
  "booleanField" : false,
  "removeFromCollectionField": ["val1","val2"],
  "addToCollectionField": ["val3","val4"]
  ...
}
```

Generally the update operations have a list of optional fields, and only specified fields will be updated.
If an object field "field" is set, the value of that field will be set on the specified object.

In the example above (relative to the object created in the previous "add"-operation), the "field" and "booleanField" is set to a new value, while "otherBooleanField" is not touched.
The collectionField is updated by removing two existing values, and adding two new values.

Update-endpoints return the updated object, after the specified field changes have been applied.
Objects with "lastUpdated"-fields are automatically updated by the service.

See API documentation for JSON structure.

Simple searching

Simple search is using the GET method, and is accepting a set of basic query parameters for filtering, sorting, and limit/offset.

Filtering

Filtering parameters are appended as query parameters, e.g.
`GET /assets/v1/host?customerID=1`
will search for host assets for customer 1

Many filtering parameters accept multiple values, the default behaviour is to search for any of the provided values, e.g.
`GET /assets/v1/host?customerID=1&customerID=2`
will search for host assets for customer 1 or 2.

Combining different search criteria generally has an AND logic, e.g.
`GET /assets/v1/host?customerID=1&keywords=myhost`
will mean to search for hosts for customer 1 which also matches the keyword myhost.

See API documentation for valid filtering parameters.

Limit, offset and sorting

By default, any endpoint has a default limit of 25 elements.
Searching endpoints (simple or advanced) will generally limit output to 25 rows, but allows user to override limit to get more/fewer rows, and set an offset to perform pagination.

The query parameters `limit` and `offset` can be added to override the default.
By setting `limit=0`, the endpoint will perform an unlimited search, up to the system defined maximum limit.

> If the user sets a limit above the maximum limit, the endpoint will return an argument error.
> If the user requests an unlimited search, which would return more than the system defined maximum limit, the endpoint will return HTTP 412 (argument error).
>
> Currently the system defined maximum result limit is 100000.

By setting `offset=X`, the endpoint will return the number of results defined by limit, after skipping the first X entries.

Endpoints have a default sort order. To change the sorting, endpoints have a "sortBy" parameter, which allows changing the default sort property and direction,
e.g. `sortBy=id` will sort by ID ascending, while `sortBy=-id` will sort by ID descending. Adding multiple sortBy parameters will define a chained sorting order, e.g. `sortBy=timestamp&sortBy=id` will sort all entities by timestamp first, then by ID.

See API documentation for valid sorting parameters.

Example:

- `limit=10` will return the first 10 results (as defined by the default sort order)
- `limit=10&offset=10` will return the next 10 results after the 10 first results (as defined by the default sort order)
- `limit=10&sortBy=timestamp` will return the first 10 results when ordered by timestamp
- `limit=0` will return all available results. If the requested search would return more than the system defined limit, this will return an argument error.

Pagination

If fetching large amounts of data, using pagination can help speed up the process by conserving both client and system resources, and will give quicker response times.

By performing an initial search with a "moderate limit" (e.g. 1000 rows), the returned count allows the client to determine how much data there is to fetch, and paginate through it.

Example:

- `GET /assets/v1/host?customerID=1&limit=1000`
  may result

```
{ "count":19350,"size":1000,"data":[...]}
```

- Then, the client can perform 19 additional GET-operations to fetch the rest:
  `GET /assets/v1/host?customerID=1&limit=1000&offset=1000`
  `GET /assets/v1/host?customerID=1&limit=1000&offset=2000`
  `GET /assets/v1/host?customerID=1&limit=1000&offset=3000`
  ...

Advanced search

Advanced search accepts all available search parameters as a JSON structure, on the form

```
{
  "singleParameter" : "value",
  "booleanParameter" : true,
  "singleNumericParameter" : 1,
  "multiParameter" : ["value1","value2"],
  "multiNumericParameter" : [1,2,3]
}
```

The general logic of such a search criteria object is that all search parameters are ANDed together. This means that resulting objects should comply with ALL these parameters.
Parameters which accept multiple values generally means that resulting objects must have any of these values.

The example above could be expressed in "SQL" as

```
... WHERE singleParameter=value AND booleanParameter=true AND
singleNumericParameter=1
AND multiParameter IN (value1, value2) AND multiNumericParameter IN (1,2,3)
```

Include/exclude flags

A general concept in many entities are *flags*. Flags are a set of boolean markers on each entity, generally output on the form

```
{
  "flags" : [ FIRST_FLAG , SECOND_FLAG ],
  ...
}
```

meaning that these two flags are set on this entity.

Advanced search criteria for entities with flags allow two parameters:

```
  {
  "includeFlags" : [ FIRST_FLAG ],
  "excludeFlags" : [ SECOND_FLAG ],
  ...
}
```

Meaning that this criteria should only match entities which have FIRST_FLAG set, and exclude all entities which have SECOND_FLAG set.
This can be combined with multiple include- and exclude-flags in each criteria.

Deleted objects

Deleted objects are filtered out by default, and are generally not accessible to users without special privileges.

To also search for deleted objects (provided the user has access to this), use the `includeDeleted` opti on (default false), if supported by the endpoint.

Keyword search

Keyword search is a common concept for many endpoints.

A keyword is a substring present in one of the supported keyword fields in the entity. A keyword search lets the client provide a list of keywords to search for, along with search strategies to determine *how* to search.

An endpoint using keyword search will accept a *keyword field strategy*, indicating which fields to search for this keyword. The strategy may be a list of fields (to search multiple fields), or the value "all" to search all supported fields.

The endpoint also contains a *keyword match strategy*, which determines if multiple keywords should be handled as an *any* or an *all* search, i.e. if results may have any of the given keywords, or if each result must have all of them.

- If the match strategy is *any*, then *each* of the keywords in the list must be present in *any one of* t he fields specified in the field strategy.
- If the match strategy is *all*, then *every* of the keywords in the list must be present in *any one of* th e fields specified in the field strategy.
- The different keywords may be in different (or multiple) fields, as long as the field is in the field strategy list.

This example will search for the words "firstword" and "secondword", returning any entity which contains either word in the fields "name" or "description".

```
{
  "keywords" : ["firstword","secondword"],
  "keywordMatchStrategy" : "any",
  "keywordFieldStrategy" : ["name","description"]
}
```

The example above could be expressed in "SQL" as

```
... WHERE
(
  ( name MATCHES "firstword" OR description MATCHES "firstword")
  OR
  ( name MATCHES "secondword" OR description MATCHES "secondword")
)
```

Changing keywordMatchStrategy to `all`

```
{
  "keywords" : ["firstword","secondword"],
  "keywordMatchStrategy" : "all",
  "keywordFieldStrategy" : ["name","description"]
}
```

This would change the "SQL" to

```
... WHERE
(
  ( name MATCHES "firstword" OR description MATCHES "firstword")
  AND
  ( name MATCHES "secondword" OR description MATCHES "secondword")
)
```

Time search

See Using Argus Search APIs - Time fields for details on how to search based on time.

User search

Most search endpoints for objects with user fields time will have a "user" parameter, along with a "userFieldStrategy" parameter, allowing to specify which user fields to search in.

This example will search for all objects where createdByUser OR lastUpdatedByUser contains one of the users "bob", "alice", "eric"

```
{
  "user": ["bob","alice","eric"]
  "userFieldStrategy" : ["createdByUser","lastUpdatedByUser"]
}
```

Subcriteria

Many advanced search endpoints support subcriteria. This allows constructing advanced queries with inclusions and exclusions.

Subcriteria have the same structure as the main criteria, allowing a nested structure:

```
{
  "customerID": [1],
  "subCriteria": [
    {"status" : ["closed"], "startTimestamp" : 1470000000000},
    {"status" : ["pendingCustomer"]},
    {"keyword" : ["interesting"], "endTimestamp" : 1490000000000}
  ]
}
```

The example above could be expressed in "SQL" as

```
... WHERE customerID IN (1) AND
(
  (status IN (closed) AND timestamp >= 1470000000000)
  OR
  (status IN (pendingCustomer)
  OR
  (keyword MATCHES (interesting) AND timestamp <= 1490000000000)
)
```

In this example, the three subcriteria are OR'ed together, while parameters inside the criteria are AND'ed together.
The criterion set in the root criteria applies to both subcriteria.

To change the logic of the subcriteria, use `exclude:true` or `required:true` .

Excluding subcriteria

In the example above, replacing the last subcriteria with

```
    {"keyword" : ["interesting"], "endTimestamp" : 1490000000000, "exclude" :
true}
```

would change the search logic to

```
... WHERE customerID IN (1) AND
(
  (
    (status IN (closed) AND timestamp >= 1470000000000)
    OR
    (status IN (pendingCustomer)
  )
  AND NOT
  (keyword MATCHES (interesting) AND timestamp <= 1490000000000)
)
```

Required subcriteria

In the example above, replacing the last subcriteria with

```
    {"keyword" : ["interesting"], "endTimestamp" : 1490000000000, "required"
: true}
```

would change the search logic to

```
... WHERE customerID IN (1) AND
(
  (
    (status IN (closed) AND timestamp >= 1470000000000)
    OR
    (status IN (pendingCustomer)
  )
  AND
  (keyword MATCHES (interesting) AND timestamp <= 1490000000000)
)
```