ARVATO SYSTEMS

# NAMS

## Implementation Guideline
## NAMS API

## Blueprint

## NAMS-8.04

**DOC-306016-10 Implementation Guideline NAMS API NAMS-8.04**

Confidential
Shareable with customers

Project: NAMS-8.04
Document Number: 306016
Version: 10
Replaces Version: 09
Country: Blueprint
Environment: n/a

# Table of Contents

# 1 Document History

| Version | Change Description | Author | Release |
|---|---|---|---|
| 08 | Updates for NAMS-8.02. See chapter "Changes since previous NAMS version". | XX | NAMS-8.02 |
| 09 | Updates for NAMS-8.03. See chapter "Changes since previous NAMS version". | XX | NAMS-8.03 |
| 10 | Updated document references. There have been no changes made to the NAMS APIs in this release. | XX | NAMS-8.04 |

# 2 Document References

## 2.1 Internal References

| Document ID | Version | Title |
|---|---|---|
| 300026 | 02 | DOC-300026-02 Glossary NMVS |
| 304007 | 20 | DOC-304007-20 Implementation Guideline NMVS CORE-6.00 |
| 306018 | 10 | DOC-306018-10 OpenAPI Specification NAMS API NAMS-8.04 |
| 306020 | 04 | DOC-306020-04 Delegated Login Flow Code Sample |

## 2.2 External References

| Document ID | Version | Title |
|---|---|---|
| n/a | n/a | n/a |

## 2.3 Attachments

| ATT | Title |
|---|---|
| n/a | n/a |

# 3 Introduction

## 3.1 Purpose & Scope

The document has been written for software suppliers who want to integrate the NAMS APIs into their software product. It describes the operations offered by the APIs, their inputs, outputs, and general behaviour.

Readers of this document are expected to be familiar with software development, web services, JSON, OpenAPI and REST.

## 3.2 Terminology

| Name | Short Description |
|---|---|
| Client System | The system using the NAMS API. |
| EMVO | European Medicines Verification Organisation |
| EMVS | European Medicines Verification System |
| Inter Market Transactions (IMT) | If a product or the combination of batch ID and serial number is not known in a national system, a request is made to the EU-HUB. From there, possible other NMVS who might know the product are contacted. |
| National Medicines Verification System (NMVS) | A system in the European Medicines Verification landscape that serves as the verification platform for one country. |
| National Medicines Verification Organisation (NMVO) | Organisation in a specific European country, which is accountable for the local NMVS. |
| National Alert Management System (NAMS) | A system in the European Medicines Verification landscape that serves as the alert analysis platform for one country. |

*Table 1 Terminology*

## 3.3 Validity of the Document

This document might change with a new NAMS release and/or new version of the web services.

This document is valid for:

NAMS Release: 8.04

Web service versions:

- Alert API: V1, V2
- Attachment Upload API: V1
- Configuration API: V1

## 3.4 Changes since previous NAMS version

In *alert-v2*, a new attribute **`causeCategoryDescription`** holds a human-readable description of the **`causeCategory`** attribute. Note that this text can be defined by an administrator, with the following consequences:

- The value can change at *any* time.
- No length constraints can be given, clients must be ready to accept empty or very long texts.

The language in which the description is to be returned can be controlled with the **`Accept-Language`** request header, see Internationalization.

## 3.5 Ecosystem Overview

The following illustration shows where the National AMS shows up in the general EMVS ecosystem. Pictured is just a subset of all the APIs and interactions in place, allowing us to focus on what's relevant from the perspective of a National AMS. Some key takeaways:

- A *NAMS* instance is always associated with the *NMVS* instance of a particular country.
- An *NMVS* instance raises alerts (because of a potential counterfeit case) and actively sends them to the *European Hub*.
- The *European Hub* actively sends the alerts to – among others – the *AMS Hub*.
- *NAMS* instances import alerts both from their associated *NMVS* instance and from the *AMS Hub*, which provides each *NAMS* instance with those alerts *their NMVS* instance was involved in.

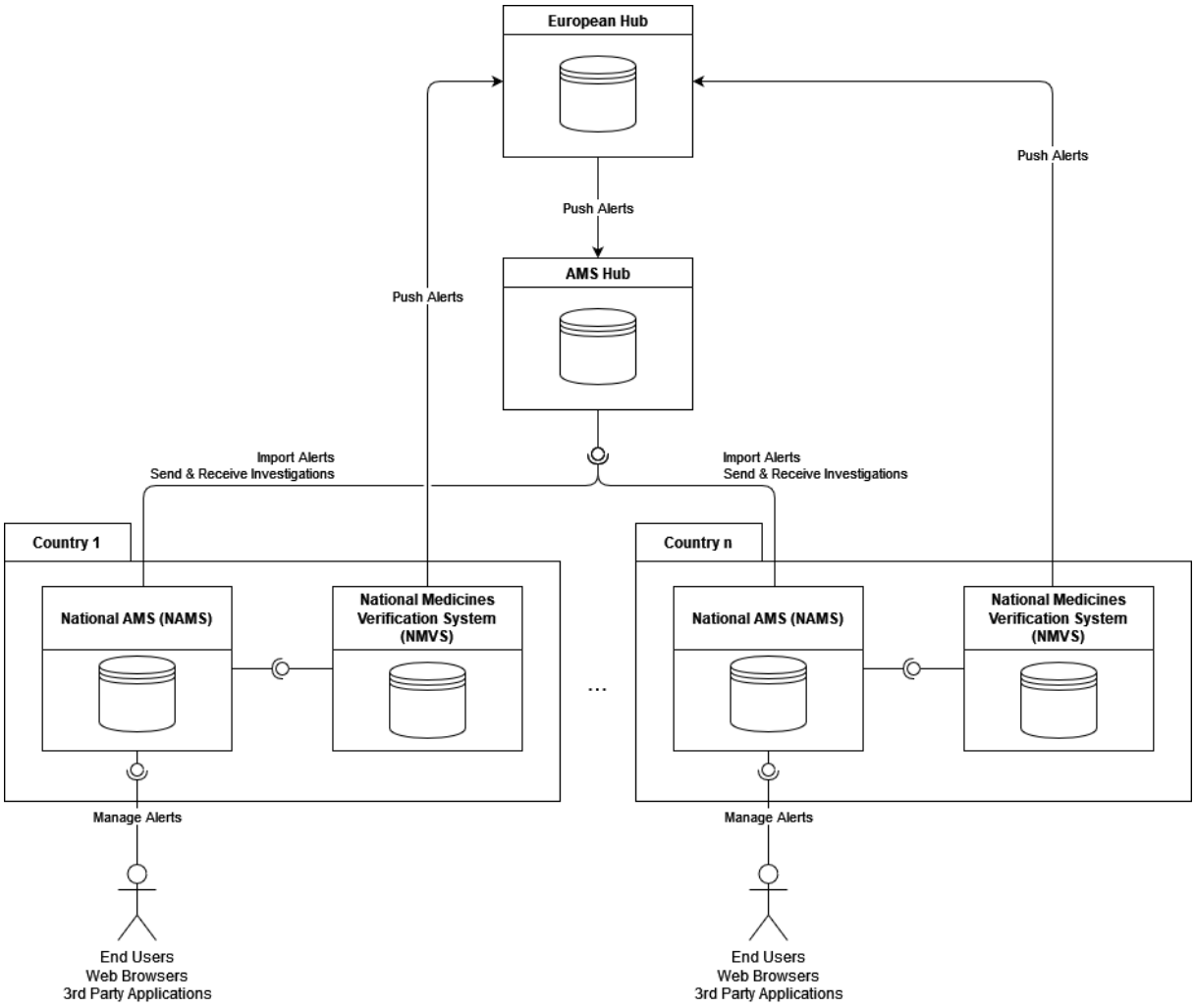


*Figure 1 Role of NAMS Instances Within the EMVS Ecosystem*

Remove

# 4 How to start

This chapter will describe how to connect to and authenticate against the NAMS APIs.

## 4.1 URLs

All NAMS services are hosted at a domain with the following pattern:

`nams-<stage>-<country>.nmvs.eu`

The authorization server is hosted at a domain with the following pattern:

`ws-auth-<stage>-<country>.nmvs.eu`

Examples:

- NAMS: https://nams-prod-de.nmvs.eu
- Authorization Server: https://ws-auth-prod-de.nmvs.eu
- NAMS: https://nams-int-pl.nmvs.eu
- Authorization Server: https://ws-auth-int-pl.nmvs.eu

The services are available at the following URLs:

| Service | URL |
|---|---|
| Web UI | https://nams-<stage>-<country>.nmvs.eu/web |
| Alert API v1 | https://nams-<stage>-<country>.nmvs.eu/api/alert/external/v1 |
| Alert API v2 | https://nams-<stage>-<country>.nmvs.eu/api/alert/external/v2 |
| Attachment Upload API v1 | https://nams-<stage>-<country>.nmvs.eu/api/attachment/external/v1 |
| Configuration API v1 | https://nams-<stage>-<country>.nmvs.eu/api/config/external/v1 |
| API Documentation (Swagger) | https://nams-<stage>-<country>.nmvs.eu/openapi/swagger-ui.html |
| Alert API Specification (OpenAPI 3) | https://nams-<stage>-<country>.nmvs.eu/docs/alert-v1<br>https://nams-<stage>-<country>.nmvs.eu/docs/alert-v2 |
| Attachment Upload API Specification (OpenAPI 3) | https://nams-<stage>-<country>.nmvs.eu/docs/attachment-upload-v1 |
| Configuration API Specification (OpenAPI 3) | https://nams-<stage>-<country>.nmvs.eu/docs/config-v1 |
| Authorization Server Token Endpoint | https://ws-auth-<stage>-<country>.nmvs.eu/WS_AUTH/services/token |
| Authorization Server Keys Endpoint | https://ws-auth-<stage>-<country>.nmvs.eu/WS_AUTH/services/jwk/keys |
| Authorization Server Authorization Endpoint | https://ws-auth-<stage>-<country>.nmvs.eu/WS_AUTH/services/authorize |
| Authorization Server Decision Endpoint | https://ws-auth-<stage>-<country>.nmvs.eu/WS_AUTH/services/authorize/decision |

*Table 2 URLs*

## 4.2 Authentication

### 4.2.1 API Authentication

To access the APIs, clients need an *access token* issued by the authorization server via the *OAuth Client Credentials* grant.

Clients must have both valid NMVS user credentials (username & password), as well as a client certificate issued by the NMVS instance.

To obtain an access token, issue the following request:

```
POST /WS_AUTH/services/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&scope=orga_type%20region_code&client_id=nmvs-
username&client_secret=nmvs-password
```

*Code Block 1 Example Access Token Request.*

Supply your NMVS credentials using the **client_id** and **client_secret** parameters.

> You *must* request the orga_type scope, and, for users with organisation type NMVO or AUTHORITY (NCA), you *must also* request the region_code scope, otherwise, API requests will be rejected.

Upon success, you'll receive an access token response similar to the following:

```
HTTP/1.1 200
Content-Type: application/json

{
  "access_token": "eyJraWQiOiI2Mzk...",
  "token_type": "Bearer",
  "expires_in": 900,
  "scope": "orga_type region_code"
}
```

*Code Block 2 Example Access Token Response.*

When making API requests, send the access token along with the **Authorization** HTTP header as follows:

```
GET /api/alert/external/v2/some/path
Authorization: Bearer eyJraWQiOiI2Mzk...
```

*Code Block 3 Example API Request with Bearer Token.*

Access tokens are valid for 15 minutes (900 seconds) by default, but this is configurable and may differ per instance. Sending requests with an expired access token will result in error responses like this:

```
HTTP/2 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", error_description="Jwt expired
at 2022-08-08T07:27:23Z",
error_uri="https://tools.ietf.org/html/rfc6750#section-3.1"
```

*Code Block 4 Example API Error Response for Expired Access Token.*

It is the responsibility of the client to request a new token once the previous token has expired.

If the access token does not contain the **orga_type** claim (or does not contain the **region_code_elements** claim), calls to the API will result in error responses like this:

```
HTTP/2 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", error_description="The
orga_type claim is not valid",
error_uri="https://tools.ietf.org/html/rfc6750#section-3.1"
```

*Code Block 5 Example API Error Response for Access Token without orga_type.*

Access tokens are in *JWT* Format (JSON Web Token). They're signed using *JSON Web Signature* (JWS). Signing keys are rotated periodically; the current *JSON Web Key Set* (JWKS) is available at the authorization server's keys endpoint and can be used to validate the token signature.

### 4.2.2 Delegated Login Flow

*This section is irrelevant for clients connecting directly to the NAMS APIs. It describes an alternate login flow enabling users of the web UI to access the NAMS without the need to provide credentials*.

Regular Flow

To demonstrate the feature, let us first review the regular login flow. Users of the NAMS web UI must present NMVS credentials and a client certificate to access the NAMS and to log in. Users perform the *Authorization Code Flow with PKCE*, which consists of the following steps (see RFC 7636 for specifics):

1. On the NAMS landing page, the user clicks the "Login" button.
2. The browser generates a random secret `code_verifier` and the `code_challenge`, which is the base64-encoded SHA-256 digest of the `code_verifier`. It stores the `code_verifier` in its local storage for later use.
3. The browser sends an *authorization request* to the NMVS authorization server. The request includes the following parameters:
    a. `client_id=ams-public-pkce` (Indicates that this authorization request was initiated for a NAMS login).
    b. `response_type=code` (Indicates that we want to perform an Authorization Code Flow to obtain an authorization code).
    c. `scope=location_id region_code orga_type refreshToken` (Specifies the user attributes to encode in the access token, and that a refresh token should be issued).
    d. `redirect_uri=https://nams-<stage>-<country>.nmvs.eu/web/app/index.html` (Specifies where to redirect the user after successful authentication. This URI is registered in the authorization server).
    e. `code_challenge=CODE_CHALLENGE_FROM_STEP_2` (Will be associated to this particular log in flow and validated later when the browser requests an access token).
    f. `code_challenge_method=S256` (Indicates that the `code_challenge` was produced using SHA-256).
4. If the parameters are valid/supported, the authorization server presents a login page, where the user enters their NMVS credentials.
   The request that submits the credentials is called *decision request*.
5. The authorization server validates the credentials and generates an *authorization code*. It adds the authorization code as a query parameter to the redirect URI specified in the authorization request, producing `https://nams-<stage>-<country>.nmvs.eu/web/app/index.html?code=AUTHORIZATION_CODE`.
   The authorization server redirects the browser to that URL.
6. The NAMS UI loads and obtains the authorization code from the query parameter.

7. The browser sends a *token request* to the authorization server. The request includes the following parameters:
    a. `client_id=ams-public-pkce` (Indicates again that this request is part of a login flow for NAMS).
    b. `grant_type=authorization_code` (Indicates that the browser wants to exchange an authorization code for an access token).
    c. `code=AUTHORIZATION_CODE` (Specifies the authorization code obtained in step 7).
    d. `code_verifier=CODE_VERIFIER_FROM_STEP_2` (Proves that this browser instance was the same instance that started the flow in step 1).
8. The authorization server checks that the authorization code is known and unused and validates that the SHA-256 digest of the code verifier is equal to the code challenge submitted with the authorization request in step 3.
9. The authorization server invalidates the authorization code and creates an access token (and a refresh token if that was requested in the authorization request) and returns that to the browser.
10. The browser uses the access token to issue requests to the NAMS API, and – if given – uses the refresh token to obtain a new access token once the previous token expires.

### Delegated Login Flow

The specific use case this flow was designed for is the software installed on pharmacy terminals, which has NMVS credentials and client certificates to call the NMVS APIs. With the *Delegated Login Flow*, steps 2 through 5 will be performed by that software. It then opens a web browser, passing the code verifier and the authorization code as fragment parameters to the NAMS UI, which uses them to complete the authorization flow starting with step 7.

The modified flow works as follows:

1. Within their pharmacy software, perhaps as an option after a transaction raised an alert, the user invokes the "Open Alert in NAMS" (or similar) functionality.
2. The pharmacy software generates the **`code_verifier`** and **`code_challenge`** parameters. These parameters must comply with the PKCE specs. The **`code_verifier`** must be a random, unique secret consisting of 43 to 128 (inclusive) characters from the following alphabet: **`A-Za-z0-9-._~`**
   The recommended method for creating the **`code_verifier`** is to generate 32 to 96 random bytes and base64url-encoding them.
   Next, the software creates the **`code_challenge`** by base64url-encoding the SHA-256 digest of the **`code_verifier`**: **`code_verifier = base64url(sha256(ascii(code_verifier)))`**
3. The pharmacy software sends the authorization request to the NMVS authorization server. The request includes the same parameters as in the regular flow. This can be a GET request:

```
GET /WS_AUTH/services/authorize?client_id=ams-public-
    pkce&response_type=code&scope=location_id%20region_code%20orga_type%2
    0refreshToken&redirect_uri=https%3A%2F%2Fnams-<stage>-
    <country>.nmvs.eu%2Fweb%2Fapp%2Findex.html%2Fapp%2Findex.html&code_ch
    allenge=CODE_CHALLENGE_FROM_STEP_2&code_challenge_method=S256
Accept: application/json
```

*Code Block 6 GET Authorization Request*

Or a POST request:

```
POST /WS_AUTH/services/authorize
Accept: application/json
Content-Type: application/x-www-form-urlencoded

client_id=ams-public-
    pkce&response_type=code&scope=location_id%20region_code%20orga_type%2
    0refreshToken&redirect_uri=https%3A%2F%2Fnams-<stage>-
    <country>.nmvs.eu%2Fweb%2Fapp%2Findex.html%2Fapp%2Findex.html&code_ch
    allenge=CODE_CHALLENGE_FROM_STEP_2&code_challenge_method=S256
```

*Code Block 7 POST Authorization Request*

By specifying **`Accept: application/json`**, the authorization server is instructed to respond with a machine-readable JSON response instead of rendering an HTML login page.

4. The authorization server returns an HTTP response with a JSON payload that contains quite a few parameters, but for brevity, all that are not relevant for this flow have been omitted from the following example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: JSESSIONID=sessionid

{
  "authenticityToken": "AUTHENTICITY_TOKEN",
  "replyTo": "https://ws-auth-<stage>-
    <country>.nmvs.eu/WS_AUTH/services/authorize/decision"
}
```

*Code Block 8 Authorization Response*

The **authenticityToken** will be needed in the next step and is used by the authorization server to ensure that both the authorization and the decision requests are performed by the same client.

The **replyTo** parameter indicates the endpoint to send the decision request to. Also note the session cookie returned by the authorization server. We'll need that in the next step.

5. The pharmacy software sends the decision request:

```
POST /WS_AUTH/services/authorize/decision
Content-Type: application/x-www-form-urlencoded
Cookie: JSESSIONID=sessionid

username=NMVS_USERNAME&password=NMVS_PASSWORD&session_authenticity_token
    =AUTHENTICITY_TOKEN&oauthDecision=allow
```

*Code Block 9 Decision Request*

6. The authorization server validates the user credentials and upon success generates the authorization code and returns that with an HTTP 303 response:

```
HTTP/1.1 303 See Other
Location: https://nams-<stage>-
    <country>.nmvs.eu/web/app/index.html?code=AUTHORIZATION_CODE
```

*Code Block 10 Decision Response*

7. The pharmacy software extracts the authorization code from the **code** query parameter and constructs a URL of the following form:

```
https://nams-<stage>-
    <country>.nmvs.eu/web/app/optional/path/to/desired/page#x-nams-
    delegated-login-flow&x-nams-delegated-login-flow-
    code=AUTHORIZATION_CODE&x-nams-delegated-login-flow-code-
    verifier=CODE_VERIFIER_FROM_STEP_2
```

*Code Block 11 Structure of NAMS URL Used for Delegated Login*

For example, to directly send the user to the details page of a specific alert, the pharmacy software constructs the following URL:

```
https://nams-<stage>-<country>.nmvs.eu/web/app/alert/DE-53cae6b6-1196-
    479e-b8e8-0b9889ab6eec#x-nams-delegated-login-flow&x-nams-delegated-
    login-flow-code=AUTHORIZATION_CODE&x-nams-delegated-login-flow-code-
    verifier=CODE_VERIFIER_FROM_STEP_2
```

*Code Block 12 Concrete Delegated Login URL*

8. The pharmacy software opens the constructed URL in a web browser on the local machine. *The pharmacy software is done at this point and the rest of the flow continues in the browser.*
9. The NAMS UI loads and inspects the fragment parameters of the current URL, finding the **`x-nams-delegated-login-flow`** marker parameter. It obtains the authorization code and the code verifier from the other fragment parameters.
10. The browser sends a token request (as in the regular flow), using the authorization code and code verifier from the fragment parameters.
11. The remaining steps are identical to steps 8, 9 and 10 of the regular flow.

Please refer to the example code to find an implementation of this flow in Java → DOC-306020-02 Delegated Login Flow Code Sample.

With this flow, the following endpoints will be accessible without needing to provide a client certificate:

- Everything under the NAMS instance's domain: https://nams-<stage>-<country>.nmvs.eu
- The following endpoints of the associated NMVS instance's authorization server:
    - /WS_AUTH/logout
    - /WS_AUTH/services/.well-known/oauth-authorization-server
    - /WS_AUTH/services/jwk/keys
    - /WS_AUTH/services/revoke
    - /WS_AUTH/services/token

Crucially, both the authorization and decision endpoints will still require a client certificate.

# 5 Technical Overview

## 5.1 Architecture

The NAMS APIs are implemented using REST with JSON and specified using OpenAPI V3. Clients authenticate with access tokens issued by the corresponding NMVS instance's OAuth authorization server.

The APIs can be explored and experimented with using the Swagger UI page (see chapter URLs).

## 5.2 Connection

Connection initiation starts from the client side to the NAMS. The NAMS is only accessible through HTTPS. Clients must comply with TLS version 1.3 (fallback to 1.2 is available) to connect with the API.

The NAMS API has been implemented to comply with the following specifications:

| Standard | Website(s) |
|---|---|
| JSON | https://www.rfc-editor.org/rfc/rfc7159.html |
| OpenAPI v3.1.0 | https://spec.openapis.org/oas/v3.1.0 |
| JSON Web Token (JWT) | https://www.rfc-editor.org/rfc/rfc7519.html |
| JSON Web Signature (JWS) | https://www.rfc-editor.org/rfc/rfc7515 |
| JSON Web Key (JWK) | https://www.rfc-editor.org/rfc/rfc7517 |
| OAuth 2.0 (specifically *Client Credentials Grant*, *Bearer Tokens* and *Proof Key for Code Exchange by OAuth Public Clients*) | https://www.rfc-editor.org/rfc/rfc6749.html#section-4.4<br>https://www.rfc-editor.org/rfc/rfc6750.html<br>https://www.rfc-editor.org/rfc/rfc7636.html |

*Table 3 Standards compliance*

These standards are mandatory for the communication with the NAMS API.

### 5.2.1 Client Certificates

The associated NMVS instance provides X.509 certificates. Clients must provide a certificate to issue requests to NAMS.

Please refer to the *NMVS Implementation Guideline* for more details on how to obtain, handle and renew client certificates.

### 5.2.2 Internationalization and Character Sets

Unless specified otherwise by **`Content-Type`** request headers, the NAMS API assumes that all request data is encoded in UTF-8. The same applies to NAMS API responses.

### 5.2.3 Connection Handling

This chapter defines connection handling requirements in more detail.

#### 5.2.3.1 Protocols

Both HTTP/2 and HTTP/1.1 are supported. Clients can use ALPN (Application-Layer Protocol Negotiation, https://en.wikipedia.org/wiki/ALPN) to negotiate the HTTP version to use.

#### 5.2.3.2 Persistent Connections

To reduce latency, it is advised to use HTTP persistent connections, i.e. HTTP/1.1 keep-alive or HTTP/2 connection multiplexing.

#### 5.2.3.3 Connection Termination

The client should drop connections if no activity is expected in a reasonable amount of time. Possibly immediately after a transaction has been executed.

#### 5.2.3.4 Reconnection Policy

If connection attempts to the NAMS fail, wait times between connection attempts must be added as the number of failed attempts increases (backoff).

## 5.3 Security

Clients must be in possession of NMVS credentials and an NMVS-issued client certificate to obtain access tokens. Your responsible NMVO can create these credentials.

Clients must then use the access token and the NMVS-issued client certificate when connecting to the NAMS. The access token's subject (username) must match the NMVS username the certificate was issued for.

Note that the NAMS does not provide any user/credential management. Users and their certificates are managed solely through the corresponding NMVS instance.

### 5.3.1 Bearer Token

Clients must provide a valid bearer token, issued by the corresponding NMVS instance's authorization server. See API Authentication for details on how to obtain one.

### 5.3.2 Client Certificate

NAMS & NMVS use client certificates as an additional security layer to legitimize user actions.

If no valid certificate is provided, an HTTP 403 error like the following will be returned:

```
HTTP/2 403 Forbidden
Server: nginx
Content-Type: text/html; charset=utf-8

<html>
<head><title>403 Forbidden</title></head>
<body>
<center><h1>403 Forbidden</h1></center>
<hr>
<center>nginx</center>
</body>
</html>
```

*Code Block 13 Error Response for missing/invalid/expired client certificate.*

Use of expired or revoked certificates will also result in HTTP 403.

If the presented certificate and bearer token were issued to different users, the following will be returned:

```
HTTP/2 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", error_description="Token does
not match client certificate."
```

*Code Block 14 Error Response for Certificate/Token Mismatch.*

## 5.4 List of APIs

At this point, the NAMS provides three APIs intended for use by 3rd party applications: The *Alert API*, the *Attachment Upload API* and the *Configuration API*.

The following sections describe the operations offered by the APIs.

### 5.4.1 List of Root URLs

The Alert API is accessible at the following root URLs:

| Endpoint Description | Root URL |
|---|---|
| Alert Management | https://nams-<stage>-<country>.nmvs.eu/api/alert/external/v2/alert |
| Predefinitions | https://nams-<stage>-<country>.nmvs.eu/api/alert/external/v2/predefinition |

*Table 4 List of Alert API Root URLs.*

The Attachment Upload API is accessible at the following root URL:

| Endpoint Description | Root URL |
|---|---|
| Attachment Upload | https://nams-<stage>-<country>.nmvs.eu/api/attachment/external/v1/attach |

*Table 5 List of Attachment Upload API Root URLs.*

The Configuration API is accessible at the following root URL:

| Endpoint Description | Root URL |
|---|---|
| Delegated Access | https://nams-<stage>-<country>.nmvs.eu/api/config/external/v1/attach |

*Table 6 List of Configuration API Root URLs.*

### 5.4.2 Data Types

#### Timestamps

Timestamps are always represented as UTC instants in ISO-8601 format. Fractional seconds are supported with up to 9 decimal places (nanosecond precision), note however, that timestamps are stored internally with a precision of 6 decimal places (microsecond precision).

Example without fractional seconds: `2022-08-09T13:44:01Z`

Example with fractional seconds: `2011-12-31T00:12:45.123456Z`

#### Other Types

Refer to the OpenAPI Specification for the full list of URLs and data types.

### 5.4.3 Internationalization

By default, a NAMS instance only supports English as a language for natural text. Administrators can configure additional languages, allowing users to view the NAMS web application in those languages.

API clients can select the response language by using the `Accept-Language` HTTP header, for example:

```
GET /api/external/...
Accept-Language: pl
...
```

*Code Block 15 Internationalization*

The above request instructs the API to return any translatable texts in Polish, if available. If no translation exists for the requested language, English is used as a fallback.

#### Current Limitations

- Translating texts into the selected language is supported in the following places (other texts will be returned in English, regardless of the requested language):
    - System-generated alert comment texts.
    - System-generated alert status transition reason texts.
    - Alert-level failure descriptions in bulk process results.
    - The `causeCategoryDescription` attribute in `AlertDto` and `AlertOverviewDto`.
    - Request schema violation error messages.

- The **`Accept-Language`** header specification (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Language) allows providing multiple languages and locales. If a client makes use of that, the NAMS API will select the *preferred* language, i.e. the *first* one listed, or – if q-factor weighting is used – the one with the *highest weight*. If the selected language is not supported, English is used as a fallback, even if a language with a lower precedence/weight would have been supported.

### 5.4.4 Trace Ids

Each request/response interaction is identified with a unique *trace id*. The id is returned to clients via the **`X-NAMS-Trace-Id`** HTTP header and consists of 32 alphanumeric characters. For example:

```
HTTP/2 200 OK
...
X-NAMS-Trace-Id: 65129781c31d4aae8fe3fd4811019128
...
```

*Code Block 16 Trace Id Response Header*

**The trace id can be used to troubleshoot issues with particular requests, specify it when contacting support.**

# 6 Function Overview / Business Processes

## 6.1 Overview

The NAMS is a ticketing system designed to help investigating and managing the alerts the corresponding NMVS is involved in, i.e. those alerts the corresponding NMVS raised and escalated to the European Hub.

An alert is raised when the NMVS encounters a potential counterfeit case, such as when a pharmacy scans a pack with an unknown serial number, or when a pharmacy attempts to decommission a pack that has already been decommissioned. For detailed information about the circumstances under which an NMVS raises alerts, please refer to the NMVS documentation.

To understand the NAMS use cases, the following is relevant to know:

- Alerts raised by the NMVS are periodically imported into the NAMS, where they can be managed via the NAMS web application or via the APIs.
- Once known to the NAMS, alert investigations (shared comments, status transitions, investigation status transitions) are periodically synchronized with other NAMS instances via the AMS Hub.
- NMVS users are grouped into distinct categories called *organisation types*. The organisation type is the distinguishing attribute that determines what a user is allowed to see and do in the corresponding NAMS.
  The list of organisation types is as follows: *AUTHORITY*, *CONCENTRATOR*, *EMVO*, *HOSPITAL*, *MAH*, *NAMS*, *NMVO*, *OPERATOR*, *PHARMACY*, *WHOLESALER*.

## 6.2 Authorization

NAMS uses multiple levels of authorization to check if a client can perform a particular API request, and what results they may receive:

1. Access List
2. Process-Level
3. Alert-Level
4. Alert Attribute-Level

Levels 1, 2 and 4 can be configured by NAMS administrators (users with organisation type NMVO) via the web application. Level 3 always applies regardless of configuration.

### 6.2.1 Access List Authorization

NAMS can be operated either in blacklist mode or in whitelist mode.

In blacklist mode, specific users (identified by their NMVS username) can be blocked from performing *any* API request.

In whitelist mode, only specific users (identified by their NMVS username) are allowed to perform requests, requests by any other user are blocked.

Users that are blacklisted (or not whitelisted respectively) will receive HTTP 403 errors like this when performing requests:

```
HTTP/2 403 Forbidden
WWW-Authenticate: Bearer error="insufficient_scope", error_description="The
request requires higher privileges than provided by the access token.",
error_uri="https://tools.ietf.org/html/rfc6750#section-3.1"
```

*Code Block 17 Error Response for Request by Blacklisted/Not Whitelisted User.*

### 6.2.2 Process-Level Authorization

NAMS administrators can specify the API operations users of each organisation type are allowed to call. For example, an administrator may deny PHARMACY users the ability to perform alert status changes. The error response returned when attempting to perform such a request is identical to one returned for an access list violation.

### 6.2.3 Alert-Level Authorization

Any API operation that refers to or affects alerts are subject to *alert-level authorization*. Details depend on the organisation type of the requesting client, as the following table shows. In general, users can only see/modify alerts where they have caused these alerts, or where they own/are authorized for the affected product, or where the user that caused the alert has granted access to these alerts (via the NAMS web GUI or the configuration API).
For example, a particular user of organisation type PHARMACY can only see those alerts that originated in their pharmacy, whereas a particular user of organisation type MAH can only see those alerts that affect a product they own.

Depending on the system configuration, NMVO and NCA users can either see all alerts, or only those that were raised inside the *region* they're responsible for.

| Organisation Type | Visible Alerts |
|---|---|
| NMVO/NCA | • When region-based alert management is *disabled*: All.<br>• When region-based alert management is *enabled*: Those where the requesting user's region is the same as or encompasses the region of the user that caused the alert.<br>See below for details on *regions*. |
| MAH | Those where the requesting user owns/is authorized for the *product* affected by the alert. |
| Any other | Those where the requesting user is the causer of the alert, or where the requesting user has been granted access by the causer of the alert. |

*Table 7 Which alerts users of different organization types are authorized for.*

Requests that affect a particular alert will return HTTP 404 if the requesting user is not authorized for the desired alert. Requests that affect multiple alerts (such as searching with filters) will simply not return those alerts the requesting user is not authorized for.

#### Region-Based Alert Management

NMVS administrators have the option to organize their users into *regions*. These regions will usually represent actual geographical regions. Regions are *hierarchical*: A region is either the *root region* which does not have a parent region, or is a child of another region. A user is always associated with exactly one region.

The following figure shows a region configuration using some of the German states and cities as an example.

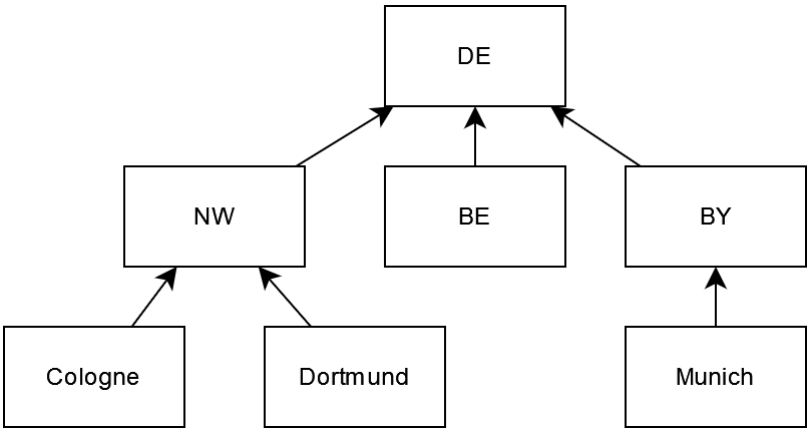


*Figure 2 Example Region Hierarchy.*

Users are associated with a particular region, which can – for example – represent their physical location (such as the city where a pharmacy is located), or their area of responsibility (such as a person/organisation being responsible for alerts raised in a particular state).

As an example, consider the following users and their associations with regions:

| Username | Region |
|---|---|
| NMVO/USER | DE |
| NCA-BY/USER | BY |
| NCA-NW/USER | NW |
| PHARM-COLOGNE/USER | Cologne |
| PHARM-DORTMUND/USER | Dortmund |
| PHARM-MUNICH/USER | Munich |

Now imagine alerts being raised by the pharmacies in Cologne, Dortmund and Munich. With region-based alert management *enabled*, these alerts would be visible as follows:

| Alert Raised By | Visible To |
|---|---|
| PHARM-COLOGNE/USER | NMVO/USER (associated with region *DE*, which is the parent of all other regions)<br>NCA-NW/USER (associated with region *NW*, which is the parent of *Cologne*)<br>PHARM-COLOGNE/USER (user that raised the alert) |
| PHARM-DORTMUND/USER | NMVO/USER (associated with region *DE*, which is the parent of all other regions)<br>NCA-NW/USER (associated with region *NW*, which is the parent of *Dortmund*)<br>PHARM-DORTMUND/USER (user that raised the alert) |
| PHARM-MUNICH/USER | NMVO/USER (associated with region *DE*, which is the parent of all other regions)<br>NCA-BY/USER (associated with region *BY*, which is the parent of *Munich*)<br>PHARM-MUNICH/USER (user that raised the alert) |

In general, with region-based alert management enabled, NMVO and NCA users can only see alerts that were raised in a region that is part of the subtree rooted at the user's region.

> The current user's region is extracted from the `region_code_elements` JWT claim. This claim is *required* for NMVO and NCA users and *must* therefore be requested by API clients in access token requests, see API Authentication for details.

### 6.2.4 Alert Attribute-Level Authorization

NAMS administrators have granular control about which alert attributes are visible to users, again depending on their organisation type. Thus, identical API requests performed by two users may yield different results, depending on which attributes the administrators have enabled for each organisation type. The following table demonstrates that concept: Both columns show responses to identical requests but performed by users with different alert attribute-level permissions. The left column only shows a few attributes, while the right column shows many more attributes.

Note that the JSON attributes are *omitted* if the necessary permission isn't granted (instead of being set to `null`, for example), which is the reason why most attributes are optional in the API specification, as their presence depends on the particular configuration.

| Response to User of Organisation Type With Many Attribute-Level Restrictions | Response to User of Organisation Type Without Attribute-Level Restrictions |
|---|---|
| {<br>"createdBy": "SYSTEM",<br>"createdAt": "2022-08-10T03:23:30.682774Z",<br>"modifiedBy": "SYSTEM",<br>"modifiedAt": "2022-08-10T03:23:56.366219Z",<br>"alertId": "DE-53cae6b6-1196-479e-b8e8-0b9889ab6eec",<br>"initiatingTransactionInformation": {},<br>"fulfillingTransactionInformation": {},<br>"intermarketTransactionType": "LOCAL",<br>"initiatingUserInfo": {},<br>"details": []<br>} | {<br>"createdBy": "SYSTEM",<br>"createdAt": "2022-08-10T03:23:30.682774Z",<br>"modifiedBy": "SYSTEM",<br>"modifiedAt": "2022-08-10T03:23:56.366219Z",<br>"alertId": "DE-53cae6b6-1196-479e-b8e8-0b9889ab6eec",<br>"status": "NEW",<br>"locationId": "1140202004",<br>"causerOrganisationType": "WHOLESALER",<br>"causerUserName": "WHOLESALER/MANAGER",<br>"causerSubUserId": null,<br>"errorCodeHub": "#A7",<br>"errorMessageHub": null,<br>"initiatingMarket": "DE",<br>"fulfillingMarket": "DE",<br>"initiatingTransactionInformation": {<br>"businessProcess": "G170",<br>"errorCodeNs": "NMVS_NC_PCK_19",<br>"errorMessageNs": null,<br>"correlationIdRequestor": "d9e5cd3f121d4455af952e58559e4410",<br>"correlationIdNs": "303dc615e78a4023826733fb28e5bf52"<br>},<br>"alertTimestamp": "2022-08-10T03:23:17.900738Z",<br>"escalationTimestamp": "2022-08-12T03:23:17.900738Z",<br>"givenProductCodeScheme": "GTIN",<br>"givenProductCode": "04151017897186",<br>"givenBatchId": "Charge_16741",<br>"givenBatchExpiryDate": "230810",<br>"givenSerialNumber": "EPS000000000023559",<br>"causeCategory": "SAME_USER_STATUS_CHANGE_SHORT_GAP",<br>"causeCategoryDescription": "Repeated status change with short gap",<br>"incidentId": "INC-43652"<br>// ... other attributes omitted for brevity<br>} |

*Table 8 Example Responses to the same alert request performed by users where administrators have configured different attribute permissions.*

### 6.2.5 Unconfirmed Terms of Use

In addition to the authorization levels described in the previous sections, users must also confirm the system's *Terms of Use* (if defined and confirmation is required). API requests performed by users that have not confirmed the currently valid terms of use will be rejected with HTTP 424.

See Terms of Use for details on how to retrieve and confirm them.

## 6.3 Use Cases / API Description

The Alert API is organized into two groups:

- Alert Management: Operations to search for alerts, get details for specific alerts, add comments, and perform (investigation) status transitions. Actions can be performed synchronously on individual alerts, or asynchronously on many alerts at once.
- Predefinition: Operations to fetch message codes and reason codes, which are required for adding comments and performing status transitions.

The following table shows the use cases implemented by the Alert API. Each use case is implemented using one or more API operations, each responsible for certain aspects of the use case.

| Use Case Number | Business Process | Operation Id(s) |
|---|---|---|
| Alert Management | | |
| P210 | Overview of Alerts | `getLegalFilters`, `findAlertsByFilter`, `findAlerts` |
| P211 | Get Alert Details | `findAlertByAlertId`, `findCommentsByAlertId` |
| P212 | Set Status on Alert | `changeAlertStatus` |
| P222 | Set Status on Bulk of Alerts | `bulkChangeAlertStatus` |
| P213 | Add Comment to Alert | `addCommentToAlert`, `addAttachmentsToAlert` |
| P223 | Add Comment to Bulk of Alerts | `bulkAddCommentToAlert` |
| P215 | Download of Attachment | `getAttachmentDownloadUri` |
| P216 | Add Attachments to Alert | `addAttachmentsToAlert` |
| P218 | Set Investigation Status on Alert | `changeAlertInvestigationStatus` |
| P228 | Set Investigation Status on Bulk of Alerts | `bulkChangeAlertInvestigationStatus` |
| P220 | Find Bulk Process Status | `getBulkJobs` |
| P221 | Retrieve Bulk Process Result | `getBulkJobStatus` |
| Predefinitions | | |
| P203 | Get Predefined Messages | `findPredefinedMessageCodes` |
| P204 | Get Predefined Reasons | `findPredefinedReasonCodes` |

*Table 9 List of Use Cases available via the Alert API.*

The Attachment Upload API consists of one group:

- Attachment Upload: One operation allowing to generate one-time-use URLs that can be used to upload attachments to alerts.

The following table shows the use cases implemented by the Attachment Upload API:

| Use Case Number | Business Process | Operation Id(s) |
|---|---|---|
| Attachment Upload | | |
| P217 | Get Alert Attachment Upload Link | `generateAlertAttachmentUploadUri` |

*Table 10 List of Use Cases available via the Attachment Upload API.*

The Configuration API consists of two groups:

- Delegated Access: Operations allowing users to manage which users are allowed to manage their alerts on their behalf.
- Terms of Use: Operations allowing users to retrieve and confirm the *Terms of Use* of the system.

The following table shows the use cases implemented by the Configuration API:

| Use Case Number | Business Process | Operation Id(s) |
|---|---|---|
| Delegated Access | | |
| P410 | Delegate Access | `getOwnAlertAccessGrants`, `addAlertAccessGrantee`, `removeAlertAccessGrantee` |
| Terms of Use | | |
| P401 | Get Terms of Use for NAMS | `findAll`, `findCurrent`, `confirmationRequired` |
| P402 | Accept Terms of Use for NAMS | `confirm` |

*Table 11 List of Use Cases available via the Configuration API.*

The next sections describe the use cases in more detail. To explore the API operations interactively, please refer to the Swagger UI.

### 6.3.1 Alert Management – P210 Overview of Alerts

This use case offers ways to retrieve lists of alerts with and without filtering.

> These API operations are subject to alert-level authorization and will only return results corresponding to alerts the requesting client is authorized for.

Operation Id `getLegalFilters`: Retrieving Legal Sort/Filter Keys

When filtering and sorting, legal keys must be supplied (otherwise, HTTP 400 "Bad Request" will be returned). To get a complete list of the allowed filter/sort keys, issue the following request:

```
GET /api/alert/external/v2/alert/filter
```

*Code Block 18 Request allowed filter/sort keys.*

Response excerpt containing three examples:

```
[
  // ...
  {
    "alias": "givenSerialNumber",
    "filterTypes": [
      "CONTAINS"
    ]
  },
  {
    "alias": "status",
    "filterTypes": [
      "NOT_IN",
      "IN"
    ]
  },
  {
    "alias": "alertTimestamp",
    "filterTypes": [
      "AFTER",
      "BEFORE"
    ]
  }
  // ...
]
```

*Code Block 19 Response valid filter/sort keys.*

The response is a JSON array containing all valid filter/sort keys, as well as the legal filter operators that can be used for them. The response excerpt gives us the following information about the three listed keys:

- We can use **`givenSerialNumber`**, **`status`** and **`alertTimestamp`** as sort keys.
- When used as filter keys, we can use **`givenSerialNumber`** for substring matches, **`status`** for list matches, and **`alertTimestamp`** for time matches.

The following table describes the available filter operators:

| Operator | Applicable Data Types | Description |
|---|---|---|
| `CONTAINS` | String | Filter value must be a substring of the matched attribute. |
| `CONTAINS_CASE_INSENSITIVE` | String | Filter value must be a substring of the matched attribute, ignoring case.<br>Supported for all filter keys where `CONTAINS` is legal as well, but not returned by the `getLegalFilters` operation in V1 or V2 for backward compatibility. |
| `AFTER` | Timestamp | Filter value is a timestamp and must be greater than or equal to the matched attribute. |
| `BEFORE` | Timestamp | Filter value is a timestamp and must be less than or equal to the matched attribute. |
| `IS` | String | Filter value must be equal to the matched attribute. |
| `IS_CASE_INSENSITIVE` | String | Filter value must be equal to the matched attribute, ignoring case.<br>Supported for all filter keys where `CONTAINS` is legal as well, but not returned by the `getLegalFilters` operation in V1 or V2 for backward compatibility. |
| `IN` | Enum, String | Filter value is a list of values, the matched attribute must be contained in the list. |
| `NOT` | String | Filter value must not be equal to the matched attribute. |
| `NOT_IN` | Enum, String | Filter value is a list of values, the matched attribute must not be contained in the list. |

*Table 12 The available filter operators and their meaning.*

The operators `CONTAINS_CASE_INSENSITIVE` and `IS_CASE_INSENSITIVE` were added *after* V2 of the alert API was introduced. To stay backward compatible with older clients, they are not returned by the `getLegalFilters` operation but can still be used for filter requests.

### Operation Id `findAlerts`: Retrieving Alerts Without Specifying Filters

This operation may be removed in a future API version, as its functionality is identical to the "Retrieving Alerts with Filters" operation.

```
GET
/api/alert/external/v2/alert/page?sortKey=SORTKEY&sortOrder=SORTORDER&page=PAGE
&size=SIZE
```

*Code Block 20 Retrieve alerts without specifying filters.*

Retrieves alerts in pages of the given size. The following query parameters specify how to return the results:

| Name | Description |
|---|---|
| `sortKey` | Key to sort results by. Optional, only takes effect when **`sortOrder`** is specified as well. |
| `sortOrder` | The sort order. Valid values: **`ASC`**, **`DESC`**. Optional, only takes effect when **`sortKey`** is specified as well. |
| `page` | Index of page to return. Must be ≥ 0. Optional, default is 0. |
| `size` | Number of alerts to return per page. Must be between 1 and 500 (inclusive). Optional, default is 20. |

*Table 13 Query parameters for page request.*

The following example retrieves the 25 most recent alerts, sorted in reverse chronological order:

```
GET
/api/alert/external/v2/alert/page?sortKey=alertTimestamp&sortOrder=DESC&page=0&
size=25
```

*Code Block 21 Retrieve most recent alerts.*

### Operation Id `findAlertsByFilter`: Retrieving Alerts with Filters

```
POST /api/alert/external/v2/alert/filter
Content-Type: application/json

(Filter request)
```

*Code Block 22 Retrieve alerts with filters.*

Use this operation to retrieve alerts in pages that match the given filters. The request payload is a JSON object with **`filter`** and **`pageRequest`** attributes. For example, the following request body retrieves the most recent alert raised before August 2022, where the given serial number contains "SN0001":

```
{
  "filter": [
    {
      "key": "alertTimestamp",
      "value": "2022-08-01T00:00:00Z",
      "filterType": "BEFORE"
    },
    {
      "key": "givenSerialNumber",
      "value": "SN0001",
      "filterType": "CONTAINS"
    }
  ],
  "pageRequest": {
    "sortKey": "alertTimestamp",
    "sortOrder": "DESC",
    "page": 0,
    "size": 1
  }
}
```

*Code Block 23 Request alerts raised before a specific date.*

All filters specified in the **`filter`** array are combined using logical AND. It is currently not possible to construct OR filters.

The **`pageRequest`** attribute's parameters are subject to the same restrictions as described in *Query Parameters for Page Request*. Note that it is valid to omit both **`filter`** and **`pageRequest`**; the following is a valid – albeit not very useful – request which results in 20 alerts in no specific order:

```
{}
```

### 6.3.2 Alert Management – P211 Get Alert Details

This use case is implemented by two operations, where one returns the attributes of a specific alert and the other returns the comments added to a specific alert.

> These API operations are subject to alert-level authorization. Performing them for alerts the requesting client is not authorized for, will result in HTTP 404 errors.

Operation Id `findAlertByAlertId`: Retrieving a Specific Alert

```
GET /api/alert/external/v2/alert/{alertId}
```

*Code Block 24 Request retrieve a specific alert.*

Use this operation to retrieve a specific alert by specifying its alert id as the **`alertId`** path variable, like so:

```
GET /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06
```

*Code Block 25 Retrieve a specific alert with alertId path variable.*

Operation Id `findCommentsByAlertId` : Retrieving the Comments of a Specific Alert

```
GET
/api/alert/external/v2/alert/{alertId}/comment?sortKey=SORTKEY&sortOrder=SORTOR
DER&page=PAGE&size=SIZE
```

*Code Block 26 Retrieve list of comments to a specific alert.*

Use this operation to retrieve the comments added to a specific alert by specifying its alert id as the **`alertId`** path variable, and page and sort parameters, like so:

```
GET /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-
3e8bfc109c06/comment?sortKey=commentTimestamp&sortOrder=DESC&page=2&size=50
```

*Code Block 27 Retrieve list of comments with alert id path variable.*

**`commentTimestamp`** is the only legal sort key, otherwise, the semantics of the query parameters are as described in *Query Parameters for Page Request*.

Shared (**`nationalComment: false`**) comments may contain up to five *attachments*. Those are files uploaded by clients to help investigate alerts. These may be photos of medicinal packs involved in an alert or out-of-band documents relevant for the investigation. The actual file contents are stored at the AMS Hub, where clients can request temporary URLs to download them. Thus, the *attachments* returned when invoking the above API operation only contain a filename and an attachment id. The NAMS offers a dedicated API operation to its clients that requests a download URL from the AMS Hub for the attachment id and then returns it. See P215 for details.

### 6.3.3 Alert Management – P212 Set Status on Alert

This use case is implemented by one operation (Operation Id **`changeAlertStatus`**), which changes the status of a specific alert:

```
PUT /api/alert/external/v2/alert/{alertId}/status
Content-Type: application/json

{
  "targetStatus": "<TARGET STATUS>",
  "reasonCode": "<REASON CODE>",
  "customReasonText": "<CUSTOM REASON TEXT>"
}
```

*Code Block 28 Change alert status.*

The alert to be transitioned is again specified using the **`alertId`** path variable. Alerts can be transitioned between any of the following states: **`ACTIVE`**, **`UNDER_INVESTIGATION`**, **`ESCALATED`**, **`CLOSED`**. Transitioning into status **`NEW`** is not allowed.

Additionally, attempting to transition an alert into a status it is already in will result in HTTP 400.

Locked Alerts

The NAMS web GUI offers users the functionality to perform status transitions for many alerts at once. These *bulk status transitions* may take multiple seconds to complete. To avoid unexpected outcomes, all alerts affected by such an operation are *locked* until it completes. Attempting to perform an alert status transition via the API for any alert participating in such a *bulk status transition*, will result in an HTTP 409 error.

### Reason Codes

To facilitate collaboration on alerts affecting multiple countries and across language barriers, as well as to enable consistent alert investigation across all connected systems, the concept of *Predefined Reasons* (see P204 and *Predefined Messages*, see P203) was introduced. When performing a status change, the user chooses one of these codes to explain *why* they did the change. Systems can then translate these codes into a meaningful message in their language of choice. The codes and default translations in English are defined by the EMVO and distributed to National AMS instances through the AMS Hub. The full list of codes known to a NAMS instance as well as their English default translations can be obtained via a dedicated API operation, see P204. Custom texts can also be given, by using the reason code RC-001 and submitting the text using the `customReasonText` attribute.

When transitioning into `UNDER_INVESTIGATION`, a reason code *can* be given. When transitioning into any other status, a reason code *must* be given.

When using the reason code `RC-001`, the `customReasonText` *must* be specified as well, when using any other reason code, the `customReasonText` attribute *must not* be specified.

In addition to the restrictions explained above, two additional restrictions apply:

### Audience

Each reason code has an `audience` attribute, it defines which stakeholder group (referring to participating systems within the larger AMS context - a NAMS has the audience type `NATIONAL`) is allowed to specify this code for a status transition. A NAMS (and by extension, 3rd party software using the NAMS API) can only use reason codes with audience `NATIONAL` or `ALL`.

### Excluded Target Status

Each reason code has zero, one or more *excluded statuses*. These are transition target statuses that reason code cannot be used for.

Violating any of these restrictions will result in HTTP 400 errors.

Some valid examples:

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "UNDER_INVESTIGATION"
}
```

*Code Block 29 Valid status change: Target status "under investigation".*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "UNDER_INVESTIGATION",
  "reasonCode": "RC-001",
  "customReasonText": "Starting work on this alert."
}
```

*Code Block 30 Valid status change: Target status "under investigation" with custom reason.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "CLOSED",
  "reasonCode": "RC-001",
  "customReasonText": "Looks like a false positive to me."
}
```

*Code Block 31 Valid status change: Target status "closed" with custom reason.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "CLOSED",
  "reasonCode": "RN-003"
}
```

*Code Block 32 Valid status change: Target status "closed" and valid reason code.*

Some **invalid** examples:

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "NEW"
}
```

*Code Block 33 Invalid Status Change: NEW is not allowed as a target status.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "UNDER_INVESTIGATION",
  "reasonCode": "RC-001"
}
```

*Code Block 34 Invalid Status Change: customReasonText missing, it is required for RC-001.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "CLOSED"
}
```

*Code Block 35 Invalid Status Change: reasonCode missing, it is required except for UNDER_INVESTIGATION.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "CLOSED",
  "reasonCode": "RN-003"
}
```

*Code Block 36 Invalid Status Change: RN-003 cannot be used with target status CLOSED.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "CLOSED",
  "reasonCode": "RO-001"
}
```

*Code Block 37 Invalid Status Change: RO-001 cannot be used by a NAMS.*

```
PUT /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-3e8bfc109c06/status
Content-Type: application/json

{
  "targetStatus": "CLOSED",
  "reasonCode": "UNKNOWN-CODE-1234"
}
```

*Code Block 38 Invalid Status Change: unknown reason code, only codes known to the NAMS can be used.*

We cannot provide a complete list of codes in this documentation because the codes, their translations and their configurations are dynamic and may change periodically and are not controlled by the NAMS. The NAMS itself must periodically fetch the codes from the AMS Hub, therefore, so do 3rd party clients that are using the NAMS API. The only code that is guaranteed to always exist and be a legal code to use by a NAMS is **`RC-001`**, used to supply custom reason texts. Additionally, codes will never be *removed*, but may become *unusable* for new status transitions.

> Codes that are removed in the AMS Hub will be flagged as *unusable* in the NAMS API. Attempting to use such a code will be treated in the same way as using a code intended for a different audience. Consult the `usable` attributes in the P204 response. It indicates whether the client is allowed to use the corresponding code when performing a status transition.

This API operation is subject to alert-level authorization. Performing it for alerts the requesting client is not authorized for will result in HTTP 404 errors.

### 6.3.4 Alert Management – P222 Set Status on Bulk of Alerts

This is the bulk variant of use case *P212*, implemented by operation **`bulkChangeAlertStatus`**. It changes the status of all given alerts:

```
PUT /api/alert/external/v2/alert/status
Content-Type: application/json

{
  "statusChange": {
    "targetStatus": "<TARGET STATUS>",
    "reasonCode": "<REASON CODE>",
    "customReasonText": "<CUSTOM REASON TEXT>"
  },
  "alertIds": [ "<ALERT ID 1>", ..., "<ALERT ID n>" ]
}
```

*Code Block 39 Change the status of multiple alerts.*

The alerts to be transitioned are specified in the **`alertIds`** array and the status change to be performed is specified in the **`statusChange`** attribute. The alert-level semantics are identical to those described in P212.

This is a bulk operation that is processed asynchronously. Use `getBulkJobStatus` to get the processing status and to retrieve results. Refer to the chapter on Bulk Processing for more details.
Alert-level validations are performed later at processing time, requests containing unknown alert ids will not be rejected.

### 6.3.5 Alert Management – P213 Add Comment to Alert

This use case is implemented by one operation (Operation Id **`addCommentToAlert`**), which adds a comment to a specific alert:

```
POST /api/alert/external/v2/alert/{alertId}/comment
Content-Type: application/json

{
  "messageCode": "<MESSAGE CODE>",
  "text": "<COMMENT TEXT>",
  "nationalComment": "<NATIONAL-ONLY FLAG>",
  "attachments": [
    {
      "filename": "<FILENAME>",
      "base64Binary": "<BASE64-ENCODED FILE CONTENT>"
    }
  ]
}
```

*Code Block 40 Add comment to specific alert.*

The alert to which to add the comment is again specified using the **`alertId`** path variable, for example:

```
POST /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-
3e8bfc109c06/comment
Content-Type: application/json

{
  "messageCode": "MC-00001",
  "text": "Hello world!",
  "nationalComment": false,
  "attachments": [
    {
      "filename": "file.txt",
      "base64Binary": "SGVsbG8gV29ybGQ="
    }
  ]
}
```

*Code Block 41 Add comment to alert with alertId path variable.*

### Message Codes

Message codes apply the same concept to comments as reason codes do to status changes. Again, the codes are periodically imported from the AMS Hub and can be retrieved by clients of the NAMS API, see P203. There are some subtle differences, though:

- Specifying a message code is *always* required.
- Specifying a message text is *required* when using the custom message code (**`MC-00001`**), and *optional* for other message codes.

The audience restrictions apply here as well, clients using the NAMS API must only use message codes intended for audiences **`NATIONAL`** and **`ALL`**.

### Attachments

Optionally, up to five file attachments with as size of up to 5MiB each can be added to a comment. Upon success, the API response contains unique ids that identify the attachments and that can be used to download them later, see P215.

Since attachments are stored with the AMS Hub, it is *not* allowed to add attachments to *national* comments.

National Comments

By default, comments are synchronized with the AMS Hub and then shared with the alert management system(s) of other countries that are involved with investigating the corresponding alert. In contrast, a *national comment* is *not* synchronized with the AMS Hub and can be used to give information only to users of the same NAMS instance.

To create a national comment, set the `nationalComment` flag to `true`. This attribute is optional and defaults to `false` when omitted. As described above, attachments cannot be added to national comments.

This API operation is subject to alert-level authorization. Performing it for alerts the requesting client is not authorized for will result in HTTP 404 errors.

### 6.3.6 Alert Management – P223 Add Comment to Bulk of Alerts

This is the bulk variant of use case *P213*, implemented by operation `bulkAddCommentToAlert`. It adds a comment to all the given alerts:

```
POST /api/alert/external/v2/alert/comment
Content-Type: application/json

{
  "comment": {
    "messageCode": "<MESSAGE CODE>",
    "text": "<COMMENT TEXT>",
    "nationalComment": "<NATIONAL-ONLY FLAG>"
  },
  "alertIds": [ "<ALERT ID 1>", ..., "<ALERT ID n>" ]
}
```

*Code Block 42 Add a comment to multiple alerts.*

The alerts to be processed are specified in the `alertIds` array and the comment to be added is specified in the `comment` attribute. The alert-level semantics are – with the exception of file attachments, which are *not* supported by this operation – identical to those described in P213.

This is a bulk operation that is processed asynchronously. Use `getBulkJobStatus` to get the processing status and to retrieve results. Refer to the chapter on Bulk Processing for more details.
Alert-level validations are performed later at processing time, requests containing unknown alert ids will not be rejected.

### 6.3.7 Alert Management – P215 Download of Attachment

This use case is implemented by one operation (Operation Id **`getAttachmentDownloadUri`**). It returns a URL that points to the location of the binary content of the given attachment id.

```
GET /api/alert/external/v2/alert/attachment/{attachmentId}
```

*Code Block 43 Download of attachment.*

The attachment is referenced using the **`attachmendId`** path variable. The response is a JSON payload with a single attribute value, providing the URL:

```
{
  "value": "https://stamshubuatwesteu.blob.core.windows.net/attachments/..."
}
```

*Code Block 44 Example response with attachment download URL.*

Attachments are hosted by the AMS Hub, therefore these URLs do not point to Arvato-controlled domains.
Additionally, the URLs are valid for a limited amount of time only and are thus not suitable for reuse over a longer period of time. It is recommended to either immediately download and store the data or request a new download URL if the same attachment is required again at a later point in time.

This API operation is subject to alert-level authorization. Performing it for attachments associated with alerts the requesting client is not authorized for will result in HTTP 404 errors.

### 6.3.8 Alert Management – P216 Add Attachments to Alert

This use case is implemented by one operation (Operation Id **`addAttachmentsToAlert`**) and can be used to add file attachments to a specific alert.

```
POST /api/alert/external/v2/alert/{alertId}/attachments
Content-Type: application/json

[
  {
    "filename": "<FILENAME>",
    "base64Binary": "<BASE64-ENCODED FILE CONTENT>"
  }
]
```

*Code Block 45 Add attachments to specific alert.*

The alert to which to add the attachments is again specified using the **`alertId`** path variable, for example:

```
POST /api/alert/external/v2/alert/XX-fffac61d-ee5e-4874-a191-
3e8bfc109c06/attachments
Content-Type: application/json

[
  {
    "filename": "file1.txt",
    "base64Binary": "SGVsbG8gV29ybGQ="
  },
  {
    "filename": "file2.png",
    "base64Binary":
"iVBORw0KGgoAAAANSUhEUgAAADAAAAAwCAMAAABg3Am1AAAAAXNSR0IArs4c6QAAAARnQU1BAACxjw
v8YQUAAABsUExURf///+Li4uTk5ODg4NnZ2bu7u9ra2lhYWDs7O1FRUWdnZwoKCmpqajU1NU9PT9XV1
f39/S4uLkVFRYeHh3BwcCcnJzw8PBUVFXp6egAAACkpKaqqqqjMzMy8vLzg4ODY2Nj8/Pz09PUdHR/v7
+2f2W+UAAAAJcEhZcwAADsMAAA7DAcdvqGQAAAB5SURBVEhL7dHLDoJADIXhmImgFUUG5iRdE3v8dmdO
QsKxdsem3OJM0+VdDzrlNRTtsnBDtD3LQ8BGbkSnXA4aPmMvIbgWctBwib2F4F7JQcM1tglB28lBww
9hCPpn8DIEb+zHEAzYryEYsT9rgH/4M1hNy+uc2wjRDDmyBSfjhp3NAAAAAElFTkSuQmCC"
  }
]
```

*Code Block 46 Example request adding two attachments to an alert.*

Attachments must always be associated with a *comment*. When uploading attachments like this, the system automatically generates a comment and associates the attachments with it. The response for this operation returns that comment and is therefore similar to that of a P213 response:

```
{
  "createdBy": "PHARMACY/USER",
  "createdAt": "2024-01-03T10:41:00Z",
  "commentId": "NA:1a86ff28-b6f5-4876-9a31-a5908edde146",
  "messageCode": "MC-00001",
  "text": "Attachment(s) added.",
  "nationalComment": false,
  "attachments": [
    {
      "id": "d828d6db-5c9d-42ea-8b94-bca4c9b1a77a",
      "filename": "file1.txt"
    },
    {
      "id": "b5ea4c32-a622-4c0f-840f-877a955bab26",
      "filename": "file2.png"
    }
  ]
}
```

*Code Block 47 Example response with added attachments.*

The `id` values of the returned attachments can be used in P215 to download the attachment content.

> This API operation is subject to alert-level authorization. Performing it for alerts the requesting client is not authorized for will result in HTTP 404 errors.

### 6.3.9 Alert Management – P218 Set Investigation Status on Alert

This use case is implemented by one operation (Operation Id **changeAlertInvestigationStatus**), which changes the investigation status *of the requesting user's organisation type* of a specific alert:

```
PUT /api/alert/external/v2/alert/{alertId}/investigation-status
Content-Type: application/json

{
  "targetStatus": "<TARGET STATUS>"
}
```

*Code Block 48 Set Investigation status on alert.*

The alert to be transitioned is again specified using the **alertId** path variable. Alerts can be transitioned between any of the following states: **INVESTIGATION_PENDING**, **NO_ROOT_CAUSE_ON_MY_SIDE**, **ROOT_CAUSE_ON_MY_SIDE**. Transitioning into status **BLANK** is not allowed and will be rejected with HTTP 400.

Attempting to transition an alert into an investigation status it is already in will also result in HTTP 400.

### Investigation Status

To be able to distinguish between the individual investigations being performed by the different stakeholders, an alert has – in addition to the overall *status* – four *investigation statuses*, represented by the following attributes:

- **endUserInvestigationStatus**
- **mahInvestigationStatus**
- **nmvoInvestigationStatus**
- **obpInvestigationStatus**

The type of these attributes is defined by an enum with the following possible values: **BLANK**, **INVESTIGATION_PENDING**, **NO_ROOT_CAUSE_ON_MY_SIDE**, **ROOT_CAUSE_ON_MY_SIDE**.

These four investigation statuses map onto the organisation types as follows:

| Investigation Status Attribute | Organisation Types |
|---|---|
| **endUserInvestigationStatus** | **CONCENTRATOR**, **HOSPITAL**, **PHARMACY**, **WHOLESALER** |
| **mahInvestigationStatus** | **MAH** |
| **nmvoInvestigationStatus** | **NMVO** |
| **obpInvestigationStatus** | *none* |

*Table 14 Mapping between organization types and investigation status attributes.*

When a user performs an investigation status change, their organisation type is used to determine which of the investigation statuses should be changed. Note that the **obpInvestigationStatus** cannot be changed by NAMS clients, as OBPs will never directly connect to NAMS instances. It is however possible to receive OBP investigation status changes via the AMS Hub.

### Investigation Status Transitions

Changing an investigation status will create *investigation status transitions*. These transitions are returned via the **`investigationStatusTransitions`** attribute. Each individual transition has – similarly to the overall alert status transitions – **`sourceStatus`** and **`targetStatus`** attributes. The **`actor`** attribute denotes which of the four investigation statuses was changed. As an example, consider the following snippet returned by a P211 request, showing investigation status transitions performed by various users:

```
{
  // other attributes omitted for brevity
  "alertId": "GB-69b7ec73-5baf-4b23-ae95-3594bdad0613",
  "obpInvestigationStatus": "NO_ROOT_CAUSE_ON_MY_SIDE",
  "mahInvestigationStatus": "BLANK",
  "nmvoInvestigationStatus": "ROOT_CAUSE_ON_MY_SIDE",
  "endUserInvestigationStatus": "INVESTIGATION_PENDING",
  "investigationStatusTransitions": [
    {
      "createdBy": "TEST-PHARMACY/USER",
      "createdAt": "2022-10-28T20:15:43.317973Z",
      "transitionId": "NA:37dd4bd0-05f3-4053-8ab9-1c9586fcbb37",
      "sourceStatus": "BLANK",
      "targetStatus": "INVESTIGATION_PENDING",
      "actor": "END_USER"
    },
    {
      "createdBy": "NMVO/USER",
      "createdAt": "2022-10-28T20:16:14.657070Z",
      "transitionId": "NA:4647ebb6-cc57-4c73-9eae-01b9c5458906",
      "sourceStatus": "BLANK",
      "targetStatus": "NO_ROOT_CAUSE_ON_MY_SIDE",
      "actor": "NMVO"
    },
    {
      "createdBy": "SI-OBP",
      "createdAt": "2022-10-28T20:22:19.724Z",
      "transitionId": "17d2a1ab-cf13-4ab3-88d3-4856d263e929",
      "sourceStatus": "BLANK",
      "targetStatus": "NO_ROOT_CAUSE_ON_MY_SIDE",
      "actor": "OBP"
    },
    {
      "createdBy": "FR-END_USER",
      "createdAt": "2022-10-28T20:22:19.724Z",
      "transitionId": "fb07d8c2-f8be-4ec3-a3a1-61ee53be9faa",
      "sourceStatus": "INVESTIGATION_PENDING",
      "targetStatus": "ROOT_CAUSE_ON_MY_SIDE",
      "actor": "END_USER"
    },
    {
      "createdBy": "TEST-PHARMACY/USER",
      "createdAt": "2023-01-12T15:27:28.005101Z",
      "transitionId": "NA:b608ad70-e207-4d94-93b3-f992556e7868",
      "sourceStatus": "ROOT_CAUSE_ON_MY_SIDE",
      "targetStatus": "INVESTIGATION_PENDING",
      "actor": "END_USER"
    },
    {
      "createdBy": "NMVO/USER",
      "createdAt": "2023-01-12T15:28:05.561909Z",
      "transitionId": "NA:945f22fe-6561-40ef-b5d5-4d0025426954",
      "sourceStatus": "NO_ROOT_CAUSE_ON_MY_SIDE",
      "targetStatus": "ROOT_CAUSE_ON_MY_SIDE",
      "actor": "NMVO"
    }
  ]
}
```


*Code Block 49 Investigation Status Transition.*

This API operation is subject to alert-level authorization. Performing it for alerts the requesting client is not authorized for will result in HTTP 404 errors.

### 6.3.10 Alert Management – P228 Set Investigation Status on Bulk of Alerts

This is the bulk variant of use case *P218*, implemented by operation **`bulkChangeAlertInvestigationStatus`**. It changes the investigation status *of the requesting user's organisation type* of all given alerts:

```
PUT /api/alert/external/v2/alert/investigation-status
Content-Type: application/json

{
  "investigationStatusChange": {
    "targetStatus": "<TARGET STATUS>"
  },
  "alertIds": [ "<ALERT ID 1>", ..., "<ALERT ID n>" ]
}
```

*Code Block 50 Change the investigation status of multiple alerts.*

The alerts to be transitioned are specified in the **`alertIds`** array and the status change to be performed is specified in the **`investigationStatusChange`** attribute. The alert-level semantics are identical to those described in P218.

This is a bulk operation that is processed asynchronously. Use `getBulkJobStatus` to get the processing status and to retrieve results. Refer to the chapter on Bulk Processing for more details.
Alert-level validations are performed later at processing time, requests containing unknown alert ids will not be rejected.

### 6.3.11 Alert Management – P220 Find Bulk Process Status

This use case is implemented by operation **`getBulkJobs`**. It returns the processing statuses of bulk jobs submitted by the requesting client, in pages of the given size:

```
GET /api/alert/external/v2/alert/bulk/job-
status?sortKey=SORTKEY&sortOrder=SORTORDER&page=PAGE&size=SIZE
```

*Code Block 51 Get most processing statuses of most recently submitted jobs.*

As usual, the query parameters specify the page to return. If omitted, the first 20 statuses are returned, sorted by submission time. The only valid **`sortKey`** is **`submittedAt`**, specify **`sortKey=submittedAt&sortOrder=DESC`** to get results in reverse-chronological order.

The following is an example response, showing a page of three job statuses:


```
{
  "content": [
    {
      "jobId": "bd891952-0f09-45bb-9044-65e004062085",
      "type": "CHANGE_ALERT_INVESTIGATION_STATUS",
      "status": "QUEUED",
      "submittedAt": "2024-08-22T13:43:40.939468Z",
      "startedAt": null,
      "completedAt": null,
      "resultAvailableUntil": null
    },
    {
      "jobId": "6582027a-cf0f-4d28-a516-f706bbe65744",
      "type": "ADD_COMMENT",
      "status": "RUNNING",
      "submittedAt": "2024-08-22T13:43:40.107312Z",
      "startedAt": "2024-08-22T13:43:47.261372Z",
      "completedAt": null,
      "resultAvailableUntil": null
    },
    {
      "jobId": "37ecd413-5bc3-444d-bc16-1ca41a589dc0",
      "type": "CHANGE_ALERT_STATUS",
      "status": "FAILED",
      "submittedAt": "2024-08-22T13:43:38.615271Z",
      "startedAt": "2024-08-22T13:43:42.428629Z",
      "completedAt": "2024-08-22T13:43:47.206120Z",
      "resultAvailableUntil": "2024-09-21T13:43:47.206120Z"
    }
  ],
  "totalElements": 49,
  "totalPages": 17,
  "number": 0,
  "last": false,
}
```


The result shows for each job:

- The *job id*, which can be used in **`getBulkJobStatus`** to retrieve the results.
- The *type*: This is an enum indicating the operation performed by the job. Possible values are **`CHANGE_ALERT_INVESTIGATION_STATUS`** for a bulk investigation status change, **`CHANGE_ALERT_STATUS`** for a bulk alert status change and **`ADD_COMMENT`** for a bulk comment job.
- The *status*: This is an enum indicating the current processing status. Possible values are **`QUEUED`**, for jobs that have been received, but are not currently being processed, **`RUNNING`** for jobs that *are* currently being processed, **`FAILED`** for jobs that have completed processing without any successful alerts, and **`SUCCESSFUL`** for jobs that have processed all alerts successfully.
- The *submission time*: This is the time the job was received.
- The *start time*: This is the time when processing was started.

- The *completion time*: This is the time when processing finished.
- The *result retention time*: This is the time until the result is available.

Refer to the chapter on bulk processing for more details. Also refer to the API specs and the Swagger UI for details on each request/response attribute.

### 6.3.12 Alert Management – P221 Retrieve Bulk Process Result

This use case is implemented by operation **`getBulkJobStatus`**. It returns the current processing status of bulk the bulk job identified by the given *job id*. If the job is *completed* (successfully or not), calling this operation will mark it as *retrieved*. Retrieved results will be deleted sooner than those that have not been retrieved.

```
GET /api/alert/external/v2/alert/bulk/{jobId}/job-status
```

*Code Block 52 Get the processing status of the specified job.*

The job for which the result should be returned is identified by the **`jobId`** path variable. The job id is a unique identifier that is generated at the time a bulk request is submitted. It can be obtained from the respective API response, or by using the **`getBulkJobs`** operation.

Responses are similar to the statuses returned by P220, but additionally contain the number of alerts, and – for completed jobs – a list of alert-level processing results. Here's an example response for a successfully completed bulk alert investigation status change:

```
{
  "jobId": "55a31177-a667-4d89-8071-d64c4d9b868f",
  "type": "CHANGE_ALERT_INVESTIGATION_STATUS",
  "status": "SUCCESSFUL",
  "submittedAt": "2024-08-22T09:20:29.960459Z",
  "startedAt": "2024-08-22T09:20:30.621462Z",
  "completedAt": "2024-08-22T09:20:32.490215Z",
  "resultAvailableUntil": "2024-08-24T09:25:40.346447Z",
  "numTotal": 3,
  "numProcessed": 3,
  "items": [
    {
      "alertId": "XX-fffac61d-ee5e-4874-a191-3e8bfc109c06",
      "status": "FAILED",
      "failureReason": "Alert with id 'XX-fffac61d-ee5e-4874-a191-3e8bfc109c06'
not found."
    },
    {
      "alertId": "XX-63a06175-ab81-44e6-bfad-4ed28b2a994b",
      "status": "FAILED",
      "failureReason": "Alert investigation already in requested state."
    },
    {
      "alertId": "XX-b036363b-12fb-4dc0-b538-f2a72db18b35",
      "status": "SUCCESS",
      "failureReason": null
    }
  ]
}
```


For items in status **FAILED**, the **failureReason** attribute *may* contain a message that explains the nature of the processing failure. Messages are in English by default, clients can use the **Accept-Language** header to request a different language. Refer to the section on Internationalization for details.

> Refer to the chapter on bulk processing for more details. Also refer to the API specs and the Swagger UI for details on each request/response attribute.

### 6.3.13 Predefinition – P203 Get Predefined Messages

This use case is implemented by one operation (Operation Id **findPredefinedMessageCodes**). It returns the list of predefined message codes and their configuration known to the NAMS:

```
GET /api/alert/external/v2/predefinition/message
```

*Code Block 53 Get predefined messages.*

Use this operation to get all the message codes known to the NAMS, and to see what codes the client software is allowed to use. For example, consider the following response:

```
[
  {
    "code": "MO-00001",
    "text": "Recommended escalation to NMVO/NCA because OBP cannot find any
errors. Please be aware of possible reporting obligations at national level",
    "audience": "OBP",
    "usable": false
  },
  {
    "code": "ME-00001",
    "text": "Recommended escalation to NMVO/NCA because EMVO cannot find any
errors. Please be aware of possible reporting obligations at national level",
    "audience": "EMVO",
    "usable": false
  },
  {
    "code": "MN-00001",
    "text": "Recommended escalation to NMVO/NCA because NMVO cannot find any
errors. Please be aware of possible reporting obligations at national level",
    "audience": "NATIONAL",
    "usable": true
  },
  {
    "code": "MN-00002",
    "text": "this is a predefined message added by an EMVO in UAT. Please be
aware of possible reporting obligations at national level",
    "audience": "NATIONAL",
    "usable": false
  },
  {
    "code": "MC-00001",
    "text": ". Please be aware of possible reporting obligations at national
level",
    "audience": "ALL",
    "usable": true
  }
]
```


*Code Block 54 Response predefined messages.*

We see that the NAMS is aware of five message codes, their default translations in English, and their intended audiences. The **`usable`** attribute shows whether a client would be allowed to specify the code when adding a comment. This is the case for *active* codes with audiences **`NATIONAL`** and **`ALL`**, as explained before. *Creating* comments with one of the other codes will result in HTTP 400 errors, but it is possible for comments with these codes to be *retrieved*, if they were created by other stakeholders and then imported from the AMS Hub, or were active previously, used for a comment and then deactivated.

### 6.3.14 Predefinition – P204 Get Predefined Reasons

This use case is implemented by one operation (Operation Id **`findPredefinedReasonCodes`**). It returns the list of predefined reason codes and their configuration known to the NAMS:

```
GET /api/alert/external/v2/predefinition/reason
```

*Code Block 55 Get predefined reason.*

Use this operation to get all the reason codes known to the NAMS, and to see what codes the client software is allowed to use. For example, consider the following response:


```
[
  {
    "code": "RN-006",
    "text": "Local system error intended to be fixed by NMVO (excluded for
status “Closed”)",
    "audience": "NATIONAL",
    "usable": true,
    "excludedStatuses": [
      "CLOSED"
    ]
  },
  {
    "code": "RO-001",
    "text": "Issue intended to be fixed by OBP (excluded for status “Closed”)",
    "audience": "OBP",
    "usable": false,
    "excludedStatuses": [
      "CLOSED"
    ]
  },
  {
    "code": "RC-001",
    "text": null,
    "audience": "ALL",
    "usable": true,
    "excludedStatuses": []
  },
  {
    "code": "RN-001",
    "text": "No technical reason for alert identified by any party",
    "audience": "NATIONAL",
    "usable": true,
    "excludedStatuses": []
  },
  {
    "code": "RN-002",
    "text": "Removed reason code.",
    "audience": "NATIONAL",
    "usable": false,
    "excludedStatuses": []
  }
]
```


*Code Block 56 Response predefined reason.*

Like predefined message codes, the predefined reason codes have **`code`**, **`text`**, **`audience`** and **`usable`** attributes with identical semantics. Additionally, the list of alert target statuses a particular code is *not* allowed to be used for, is specified by the **`excludedStatuses`** array. In this example, an attempt to use **`RN-006`** for a status transition with target status **`CLOSED`** would be rejected with HTTP 400.

### 6.3.15 Attachment Upload – P217 Get Alert Attachment Upload Link

This use case is implemented by one operation (Operation Id **`generateAlertAttachmentUploadUri`**). It returns a URL that points to a web page optimized for mobile clients where files can be uploaded: that are then attached to the alert referenced by the **`alertId`** path variable:

```
GET /api/attachment/external/v1/attach/{alertId}/upload-uri
```

*Code Block 57 Get alert attachment upload link.*

The alert for which to generate the uploade URL is specified using the **`alertId`** path variable, like so:

```
GET /api/attachment/external/v1/attach/XX-c12e536d-968e-41fc-b119-
092a75ca8256/upload-uri
```

*Code Block 58 Get alert attachment upload link with alertId path variable.*

Responses look like this and only contain a **`uri`** attribute with the generated URL as its value:

```
{
 "uri": "https://nams-<stage>-
<country>.nmvs.eu/attach/app/upload/aHR0cHM6Ly9uY..."
}
```

*Code Block 59 Response get alert attachment upload link.*

Navigating to this URL with a web browser will open the *Mobile Upload Page*, which looks like this:

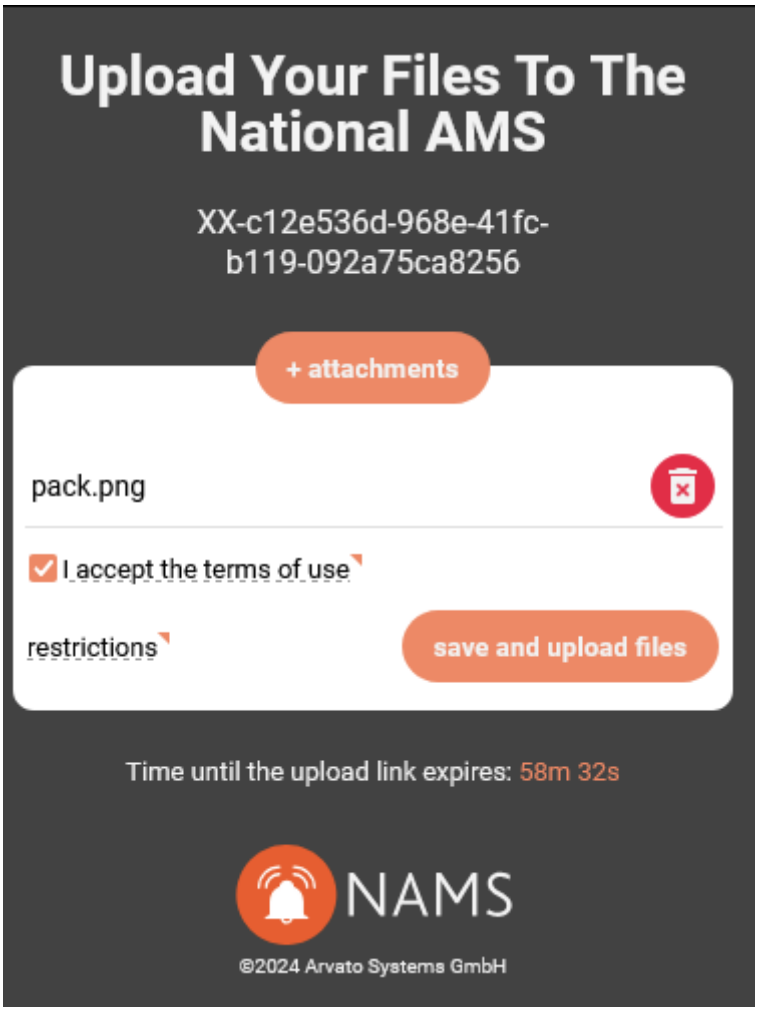


*Figure 3 Screenshot of Mobile Upload Page*

The Mobile Upload Page allows users to attach files to the alert identified by the alert id the URL was generated for, *without* authentication/authorization. A use case for this is to allow users to easily attach files (such as photos) from their mobile phone, without having to configure their credentials and client certificates.

The generated URL can only be used to upload files to the corresponding alert and can only be used once, i.e. it becomes invalid after a successful upload. Additionally, the URL is valid only for a specific amount of time (one hour by default, but this is configurable and may differ between NAMS instances).

**The *Mobile Upload Page* is an *optional* feature. Customers can request it to be disabled for their NAMS instance. If that is the case, calling this API operation will fail with HTTP 404, regardless of whether the referenced alert id exists or not.**

**This API operation is subject to alert-level authorization. Performing it for alerts the requesting client is not authorized for, will result in HTTP 404 errors.**

### 6.3.16 Configuration – P410 Delegate Access

This use case is implemented by three operations, allowing users to configure which other users are allowed to manage their alerts on their behalf. We use the terms *grantor* and *grantee*: The *grantor* is the user that grants access to their alerts to another user, the *grantee*.

#### 6.3.16.1 Adding Grantees

The following request (Operation Id **`addAlertAccessGrantee`**) adds another user – specified with the **`grantee`** query parameter – to the list of users that are allowed to manage the grantor's alerts:

```
PUT /api/config/external/v1/delegated-access/alert?grantee={grantee}
```

*Code Block 60 Grant another user the permission to manage the calling user's alerts.*

The following restrictions apply to this operation, violating these will be rejected with HTTP 400:

1. At most two grantees can be specified for a single grantor.
2. The grantor cannot add themselves as the grantee.
3. The grantee must be an existing user known to the associated NMVS instance.
4. Grantor and grantee must belong *to the same client*.

This operation is idempotent, adding the same user again is allowed but doesn't have an effect.

Examples

For the following examples, assume that the calling user is named *PHARM-ABC/USER1*. Also note the URL-encoded query parameters.

```
PUT /api/config/external/v1/delegated-access/alert?grantee=PHARM-ABC%2FADMIN
```

*Code Block 61 Valid, assuming the grantee exists in the NMVS.*

```
PUT /api/config/external/v1/delegated-access/alert?grantee=PHARM-ABC%2FSOME-
UNKNOWN-USER
```

*Code Block 62 Invalid, assuming the grantee doesn't exist in the NMVS.*

```
PUT /api/config/external/v1/delegated-access/alert?grantee=PHARM-DEF%2FUSER
```

*Code Block 63 Invalid, the grantee does not belong to the same client.*

```
PUT /api/config/external/v1/delegated-access/alert?grantee=PHARM-ABC%2FUSER2

PUT /api/config/external/v1/delegated-access/alert?grantee=PHARM-ABC%2FUSER3

PUT /api/config/external/v1/delegated-access/alert?grantee=PHARM-ABC%2FUSER4
```

*Code Block 64 Assuming all grantees exist, the third request will fail because the limit of at most two grantees has been exceeded.*

#### 6.3.16.2 Removing Grantees

The following request (Operation Id **`removeAlertAccessGrantee`**) removes a user specified with the **`grantee`** query parameter from the list of users that are allowed to manage the grantor's alerts:

```
DELETE /api/config/external/v1/delegated-access/alert?grantee={grantee}
```

*Code Block 65 Revoke the permission to manage the calling user's alerts from another user.*

It is allowed to call this operation with grantees that aren't on the calling user's list. In those cases, the requests don't have any effect.

#### 6.3.16.3 Retrieving Grantors and Grantees

With the following request (Operation Id **`getOwnAlertAccessGrants`**), a user can get a list of *grants*, i.e. where they are either the grantor or the grantee:

```
GET /api/config/external/v1/delegated-access/alert
```

*Code Block 66 Retrieve list of grants, either given to or received from other users.*

Example

Assume that three users have set up the following grants:

| Grantor | Grantee(s) |
|---|---|
| PHARM-ABC/USER1 | PHARM-ABC/USER2 |
| PHARM-ABC/USER2 | PHARM-ABC/USER1<br>PHARM-ABC/USER3 |
| PHARM-ABC/USER3 | PHARM-ABC/USER1 |

The following example response may be returned when the operation is performed by **PHARM-ABC/USER1**:


```
[
  {
    "createdBy": "PHARM-ABC/USER1",
    "createdAt": "2023-05-15T15:27:39.805Z",
    "modifiedBy": "PHARM-ABC/USER1",
    "modifiedAt": "2023-05-15T15:27:39.805Z",
    "grantorUsername": "PHARM-ABC/USER1",
    "granteeUsername": "PHARM-ABC/USER2"
  },
  {
    "createdBy": "PHARM-ABC/USER2",
    "createdAt": "2023-05-25T20:46:19.595Z",
    "modifiedBy": "PHARM-ABC/USER2",
    "modifiedAt": "2023-05-25T20:46:19.595Z",
    "grantorUsername": "PHARM-ABC/USER2",
    "granteeUsername": "PHARM-ABC/USER1"
  },
  {
    "createdBy": "PHARM-ABC/USER3",
    "createdAt": "2023-05-25T15:26:38.114Z",
    "modifiedBy": "PHARM-ABC/USER3",
    "modifiedAt": "2023-05-25T15:26:38.114Z",
    "grantorUsername": "PHARM-ABC/USER3",
    "granteeUsername": "PHARM-ABC/USER1"
  }
]
```


*Code Block 67 Example Grants Response*

We see that the response contains one item for each grant, where the **PHARM-ABC/USER1** is either the grantor or a grantee.

> These API operations are only available to users with organisation type PHARMACY, CONCENTRATOR, WHOLESALER and HOSPITAL. Requests by users of other organisation types will be rejected with HTTP 403.

### 6.3.17 Configuration - Terms of Use

*Terms of Use* ("terms" hereafter) are an optional feature allowing administrators to define the terms and conditions that users must accept in order to be allowed to use the NAMS. The terms themselves can be defined with basic text styling (bold, italics, underline, etc.) and structuring (headings, paragraphs, lists, etc). The formatted text is stored in HTML format.

The terms have an *effective date*, defining the point in time at which users will be required to accept them before they can (continue) using the system. Administrators can specify if accepting the terms is required for specific users, or globally for all users. For example, they may globally require confirmation, but exclude specific technical users from having to confirm the terms. Or, they may configure terms for informational purposes only without forcing anyone to explicitly confirm them.

API calls by users that are required to accept terms but have not done so in time will be rejected with HTTP 424.

Retrieving and confirming the terms is realized with two uses cases that are documented in the following sections.

#### 6.3.17.1 Configuration - P401 Get Terms of Use for NAMS

This use case is implemented by three operations, allowing users to retrieve current and future versions, and to quickly check if new terms are to be confirmed.

Operation Id `findAll`: Retrieving All Versions

With the following request, both the current and future version of the terms (if any) can be retrieved:

```
GET /api/config/external/v1/terms-of-use
```

*Code Block 68 Retrieve all versions of the Terms of Use*

The following is an example response indicating that one version is currently in effect which the calling user is required to confirm, and another version will become effective later which does not need to be confirmed by the calling user (of course assuming the time at which the request was performed is between March 1st and June 10th 2023):

```
[
  {
    "htmlContent": "<h1>Current Version</h1><p>Lorem ipsum...</p>",
    "validFrom": "2023-03-01T00:00:00Z",
    "inEffect": true,
    "confirmationRequired": true
  },
  {
    "htmlContent": "<h1>Future Version</h1><p>Lorem ipsum...</p>",
    "validFrom": "2023-06-10T00:00:00Z",
    "inEffect": false,
    "confirmationRequired": false
  }
]
```

*Code Block 69 Example Terms of Use Response*

The **`confirmationRequired`** attribute represents the *configuration* of the terms, and is **`true`** if the administrator requires confirmation *and* the calling user is *not* exempt from confirming them. The value is **`false`** if the administrator either does not require confirmation, or the calling user is exempt from confirming them.

Note that **`htmlContent`** is nullable. If it is **`null`**, **`validFrom`** specifies when the previous version *goes out of effect*. For example, the following response indicates that the current version will go out of effect on June 10th 2023:

```
[
  {
    "htmlContent": "<h1>Current Version</h1><p>Lorem ipsum...</p>",
    "validFrom": "2023-03-01T00:00:00Z",
    "inEffect": true,
    "confirmationRequired": true
  },
  {
    "htmlContent": null,
    "validFrom": "2023-06-10T00:00:00Z",
    "inEffect": false,
    "confirmationRequired": true
  }
]
```

*Code Block 70 Example Terms of Use Response with one version going out of effect in the future.*

Note that if **`htmlContent`** is **`null`** (to indicate the date at which the terms go out of effect), **`confirmationRequired`** has no meaning.

If no terms have been configured, an empty JSON array is returned.

### Operation Id `findCurrent`: Retrieving the Current Version

With the following request, the *currently effective* terms can be retrieved:

```
GET /api/config/external/v1/terms-of-use/current
```

*Code Block 71 Retrieve the currently effective Terms of Use.*

If terms are currently in effect, a response like this is returned:

```
{
  "htmlContent": "<h1>Current Version</h1><p>Lorem ipsum...</p>",
  "validFrom": "2023-03-01T00:00:00Z",
  "inEffect": true,
  "confirmationRequired": true
}
```

*Code Block 72 Example Response with Current Terms of Use.*

If no terms are currently in effect (either because none have been configured, or because the configured effective date has not been reached), an HTTP 404 is returned.

### Operation Id `confirmationRequired`: Checking If Confirmation of Terms of Use is Required

With the following request, one can check if terms are in effect that the calling user has not yet confirmed:

```
GET /api/config/external/v1/terms-of-use/confirmation-required
```

*Code Block 73 Check if terms must be confirmed.*

The following response is returned, if terms are in effect *and* the current user has not confirmed them *and* the current user is required to confirm them in order to be able to perform API requests:

```
{
  "value": true
}
```

*Code Block 74 Example Response indicating that terms must be confirmed by the calling user.*

If **`true`** is returned, the calling user *is required* to confirm the current terms (see P402). Until this is done, API calls (except those to retrieve/confirm the terms) will be rejected with HTTP 424.

Similarly, if either no terms are currently in effect, or the calling user has already confirmed them, or the calling user does not need to confirm terms, the following is returned:

```
{
  "value": false
}
```

*Code Block 75 Example Response indicating that terms need not be confirmed by the calling user.*

**The response payload of this operation is significantly smaller compared those returned by the operations used to retrieve the terms. It is thus strongly recommended to use the `confirmationRequired` operation in use cases where clients want to check if new terms must be confirmed (such as in polling scenarios).**

#### 6.3.17.2 Configuration - P402 Accept Terms of Use for NAMS

This use case is implemented by one operation (Operation Id **`confirm`**), allowing the user to confirm the *currently effective* terms of use. The request looks like this:

```
PUT /api/config/external/v1/terms-of-use/confirm
```

*Code Block 76 Confirms the current terms.*

If terms are currently in effect, the operation returns a bodiless HTTP 200. If no terms are currently in effect, the operation returns an HTTP 404.

## 6.4 Bulk Processing

In addition to the *synchronous* operations that operate on individual alerts, the NAMS API offers *bulk* operations. These operate *asynchronously* on multiple alerts at once and are more resource-efficient compared to issuing many individual requests.

The bulk processing approach implemented by NAMS is described in the following sections.

### 6.4.1 Job Lifecycle

Bulk requests are processed asynchronously. When receiving such a request, NAMS creates a *job*, associates it with a unique *job id* and stores it in a queue for later processing. Then, the API operation completes immediately with HTTP 202 (Accepted), returning the generated job id in its response.

When first enqueued, a job's status is **`QUEUED`**. When actual processing starts, its status changes to **`RUNNING`**. Upon completion, the status finally becomes either **`SUCCESSFUL`** (at least one alert could be processed successfully) or **`FAILED`** (no alert was processed successfully, or a more general error happened while processing the job). The following diagram describes these status transitions:

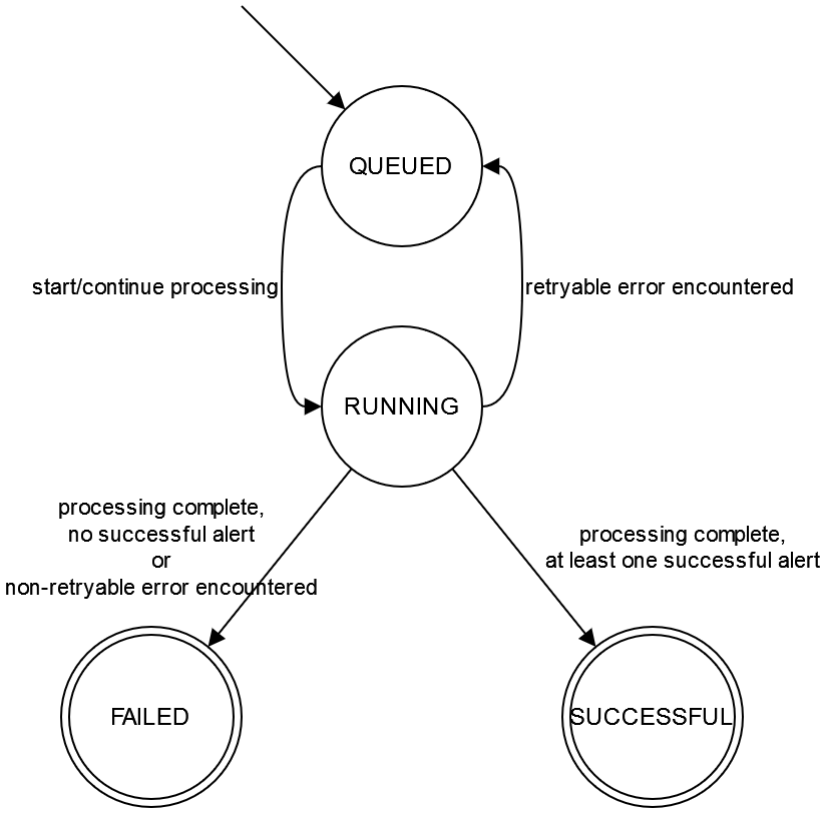


*Figure 4 Job status transitions.*

### 6.4.2 Retrieving Results

When accepting a bulk request, NAMS generates a unique id and associates it with the created job instance. That *job id* is used by the client to retrieve the processing results. The following diagram illustrates that:

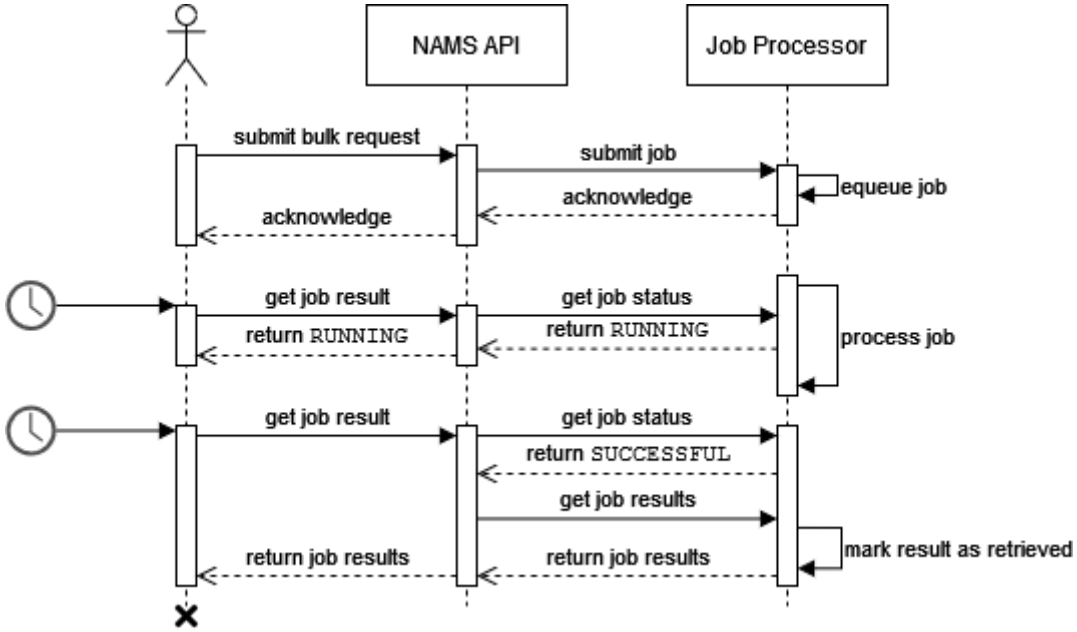


*Figure 5 Job processing and result retrieval.*

To get the job results, the client must call the **getBulkJobStatus** operation with the job id (see P221) and check the response's **status** attribute: If it is **QUEUED** or **RUNNING**, the clients should wait a few seconds and try again. As an example, consider a client requesting an investigation status change for multiple alerts:

1. Send a PUT request to */api/alert/external/v2/alert/investigation-status* with body

```
{
  "investigationStatusChange": {
    "targetStatus": "INVESTIGATION_PENDING"
  },
  "alertIds": [ "XX-fffac61d-ee5e-4874-a191-3e8bfc109c06", "XX-63a06175-
   ab81-44e6-bfad-4ed28b2a994b", "XX-b036363b-12fb-4dc0-b538-
   f2a72db18b35" ]
}
```

2. The API responds with HTTP 202 and a body like this:

```
{
  "jobId": "55a31177-a667-4d89-8071-d64c4d9b868f",
  "type": "CHANGE_ALERT_INVESTIGATION_STATUS",
  "status": "QUEUED",
  "submittedAt": "2024-08-22T09:20:29.960459Z",
  "startedAt": null,
  "completedAt": null,
  "resultAvailableUntil": null
}
```

This indicates that a job instance has been created and enqueued, its id is **55a31177-a667-4d89-8071-d64c4d9b868f**.

3. Wait a few seconds and send a GET to */api/alert/external/v2/alert/bulk/55a31177-a667-4d89-8071-d64c4d9b868f/job-status*.

4. The API responds with HTTP 200 and a body like this:

```
{
  "jobId": "55a31177-a667-4d89-8071-d64c4d9b868f",
  "type": "CHANGE_ALERT_INVESTIGATION_STATUS",
  "status": "RUNNING",
  "submittedAt": "2024-08-22T09:20:29.960459Z",
  "startedAt": "2024-08-22T09:20:30.621462Z",
  "completedAt": null,
  "resultAvailableUntil": null,
  "numTotal": 3,
  "numProcessed": 1,
  "items": []
}
```

This indicates that processing has started and is in progress. Note that individual item results are only returned for completed jobs, so **items** is empty.

5. Because the status is **RUNNING**, go back to step 3 and try again, until the status is neither **RUNNING** nor **QUEUED**. A response for a successfully completed job may look like this:


```
{
  "jobId": "55a31177-a667-4d89-8071-d64c4d9b868f",
  "type": "CHANGE_ALERT_INVESTIGATION_STATUS",
  "status": "SUCCESSFUL",
  "submittedAt": "2024-08-22T09:20:29.960459Z",
  "startedAt": "2024-08-22T09:20:30.621462Z",
  "completedAt": "2024-08-22T09:20:32.490215Z",
  "resultAvailableUntil": "2024-08-24T09:25:40.346447Z",
  "numTotal": 3,
  "numProcessed": 3,
  "items": [
    {
      "alertId": "XX-fffac61d-ee5e-4874-a191-3e8bfc109c06",
      "status": "FAILED",
      "failureReason": "Alert with id 'XX-fffac61d-ee5e-4874-a191-
   3e8bfc109c06' not found."
    },
    {
      "alertId": "XX-63a06175-ab81-44e6-bfad-4ed28b2a994b",
      "status": "FAILED",
      "failureReason": "Alert investigation already in requested state."
    },
    {
      "alertId": "XX-b036363b-12fb-4dc0-b538-f2a72db18b35",
      "status": "SUCCESS",
      "failureReason": null
    }
  ]
}
```


This indicates that two of the alerts could not be processed, but the third one was transitioned successfully. Since at least one alert was successfully processed, the overall job status is **SUCCESSFUL** as well. This particular result will be retained *at least* until the time specified by the **resultAvailableUntil** attribute.

### 6.4.3 Result Availability

Results of completed (successful or failed) jobs will eventually be deleted to conserve resources. The time at which that happens depends on whether the result has been retrieved by the client or not, as the following table shows:

| Result retrieved using `getBulkJobStatus` operation? | Earliest Deletion | Remarks |
|---|---|---|
| no | Time of completion plus 720 hours (30 days). | Once a job is complete, clients have to retrieve the results within 720 hours. |
| yes | Time of retrieval plus 48 hours. | Once the results have been retrieved for the first time, clients can retrieve the results again within the following 48 hours. |

*Table 15 Job result retention periods.*

Note that job results become *eligible* to be deleted once their retention period ends. Actual deletion may happen a short time later. This means that it is possible for `resultAvailableUntil` to be in the past.

> A job result's retention period will *only* be shortened when the same client that submitted the job explicitly retrieves it using the `getBulkJobStatus` operation. Using `getBulkJobs` to get a list of job statuses will not mark a result as retrieved, neither will a result retrieval performed by a different client.

# List of Tables

# List of Figures