nevisFIDO

Reference Guide

nevisFIDO

Reference Guide

Software Version: 1.15.4.4, Date: 03.08.2022

Version 1.15.4.4

Table of Contents

1. About This Document

This content is part of the nevisFIDO technical documentation. The technical documentation provides all the information required for the installation, configuration, administration and operation of the NEVIS Mobile Authentication component nevisFIDO.

For more high-level information regarding concepts and integration, please visit NEVIS Mobile Authentication - Concept and Integration Guide.

1.1. Document Conventions

Below, find an overview of the NEVIS specific styles used in the nevisFIDO documentation.

Code Block

Used to mark code snippets and code extracts.

Example:

```
<init-param>
    <param-name>MaxVirtualSessionsPerClient</param-name>
    <param-value>1</param-value>
</init-param>
```

Emphasis

Used within running text passages to mark technical elements/components in italics. This includes:

- Single code expression (if it appears within a text sentence).
- User interface elements, such as the names of views, fields, tabs, buttons, and so on.
- Names of AuthStates (nevisAuth).
- Names of filters and servlets (nevisProxy).
- URIs/paths to directories (if they appear within a text sentence).

Example:
In addition to the parameters available from the *AuthHttpClient*, the *TokenAuthHttpClient* evaluates parameters starting with *token*.

Info

Used to mark text intended as information worth pointing out.

Example:

Personalize your copy of Microsoft Office by filling in your name and internal phone number.

Note

Used to mark text intended as an important note that should draw explicit attention, but is not so urgent as a warning.

Example:

Do not delete the section break or the footer will disappear.

Warning

Used to mark text intended as a warning.

Example:

	*Do not log out during processing, otherwise you will lose all content.*

	*Fading out* *Content written in grey refers to a function, attribute, parameter, or other technical element that is fading out. Do not use such a function for new setups. In existing setups, switch to an alternative solution if possible. A Fading out text element is preceded by a note.*

	*Deprecated* *Content written in red refers to a function, attribute, parameter, or other technical element that is deprecated. A deprecated element becomes invalid or obsolete in future release versions. Do not use it anymore. A Deprecated text element is preceded by a warning.*

2. Installation

This chapter describes how to get a nevisFIDO component up and running.

2.1. System Environment

nevisFIDO supports the following system environment:

Table 1. System environment

Operating system	Red Hat Enterprise Linux version 7 (RHEL7)
Third party components	Java Runtime Environment 8
Hardware	It is recommended using a minimal hardware configuration of • Multicore CPU 2,90 GHz • 4 GB RAM • 20 GB HDD

nevisFIDO {project-version} requires a JDK 8 environment.

2.2. Server RPM Installation

The nevisFIDO server is packaged in the *RPM* format. It can be installed with the RPM Package Manager.

```
rpm -i nevisfido-<version>.noarch.rpm
```

The previous command will:

- Create the directory `/opt/nevisfido`, containing binaries, documentation and Configuration templates.
- Add a *systemd* unit template.
- Add a nevisFIDO Administrative Command-Line Interface on the path.

2.3. Client RPM Installation

The nevisFIDO client libraries are packaged in the *RPM* format. They can be installed with the RPM Package Manager.

```
rpm -i nevisfidocl-<version>.noarch.rpm
```

The previous command will:

- Create the directory `/opt/nevisfidocl/nevisauth/lib`, containing the nevisAuth AuthStates.
- Create the directory `/opt/nevisfidocl/nevislogrend/lib` with the JavaScript Login Application.

2.4. Environment Configuration

As the first priority, nevisFIDO uses the Java installation defined in the file `env.conf` using the configuration property `JAVA_HOME`. If the `JAVA_HOME` property is not defined in the file `env.conf`, the Java version as defined in the `PATH` environment variable is used.

To define the usage of a specific Java installation, we recommend setting the configuration property `JAVA_HOME` in the file `env.conf`:

Example:

```
JAVA_HOME=/etc/alternatives/jre_1.8.0
```

2.5. Creating an Instance

An instance is one representation of the nevisFIDO component. All configurations, log and other files related to the instance are located in the directory `/var/opt/nevisfido/<instance>`.

`nevisfido` *is the Administrative Command-Line Interface.*

```
nevisfido create <instance>
```

The previous command will:

- Create the directory `/var/opt/nevisfido/<instance>`.
- Copy configuration templates from the directory `/opt/nevisfido/template/conf` into the instance directory.
- Enable the *systemd* unit for this instance.

2.6. Starting the Instance

```
nevisfido start <instance>
```

The previous command will start nevisFIDO via *systemd* using the configuration of this instance.

When creating a nevisFIDO instance, you copy configuration templates into the instance directory (see also Creating an Instance). These default templates contain a basic configuration. Because nevisFIDO requires a sophisticated integration with various components, you must change this default configuration to represent your actual system and use cases.

You find more information on configuring nevisFIDO in Configuration.

3. Instance Management

3.1. Administrative Command-Line Interface

The administrative nevisFIDO command-line interface can be used to create and manage instances of nevisFIDO.

It consists of the following commands:

Command	Description
`nevisfido start <instance>`	Starts a nevisFIDO server instance.
`nevisfido stop <instance>`	Stops a nevisFIDO server instance.
`nevisfido restart <instance>`	Restarts a nevisFIDO server instance.
`nevisfido status <instance>`	Displays the status of a nevisFIDO server instance.
`nevisfido list`	Lists all nevisFIDO server instances.
`nevisfido create <instance>`	Creates a nevisFIDO server instance.
`nevisfido remove <instance>`	Removes a nevisFIDO server instance.

4. Configuration

This section describes the required nevisFIDO configuration files and options.

4.1. Application Configuration

The main nevisFIDO configuration is specified in a file in standard YAML format. You find the file in the instance directory: `/var/opt/nevisfido/<instance>/conf/nevisfido.yml`.

It is possible to replace the configuration by supplying a custom `--config` location in the `RUN_ARGS` environment variable in the `env.conf` file of the instance.

Some property values are expressions that will be replaced. The next table shows the available syntax:

Table 2. Expression syntax

${exec:command}	server.host: ${exec:hostname -f}	Executes the given command and uses its output as the value.
${env:variablename}	server.host: ${env:HOSTNAME}	Uses the value of the specified environment variable.

4.1.1. Server Configuration

The server configuration defines the configuration of the main web server.

server
: The root node of the server configuration.

`port`
: Web application port. If not specified, the default value is 8080.

`host`
: Network address to which the server must bind. By default, it binds to all available interfaces (that is, "0.0.0.0").

`protocol`
: Either `http` or `https`. It is advisable to always run nevisFIDO in `https` mode.

`connection-timeout`
: The time after which idle connections will be terminated. The default value is 30 seconds. If no time unit is provided, seconds will be used.

tls
: Either "http" or "https". It is advisable to enable TLS, by configuring the value "https".

`keystore`
: Path to the keystore that holds the TLS certificate (typically a PKCS12 file) of nevisFIDO.

`keystore-passphrase`
: Password used to access the keystore and the key. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`keystore-type`
: Defines the type of keystore. It is recommended using a "pkcs12" type of keystore. If not specified, the system will use the default keystore type of the Java Virtual Machine that runs nevisFIDO.

`key-alias`
: Alias that identifies the key in the keystore.

`require-client-auth`
: `true` if client authentication (2-way TLS) is required for the TLS connection, `false` if no client authentication

is required.

`truststore`
: Path to the truststore that holds the TLS certificates (typically a PKCS12 file) that nevisFIDO trusts. These are the certificates presented by the client when doing client authentication.

`truststore-passphrase`
: Password used to access the truststore contents. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`truststore-type`
: Defines the type of truststore. It is recommended using a "pkcs12" type of keystore. If not specified, the system will use the default keystore type of the Java Virtual Machine that runs nevisFIDO.

`supported- protocols`
: Provides a list of protocols that are accepted by the client when trying to initiate a connection with TLS. This attribute must be provided as an array. By default only `TLSv1.2` is supported.

`cipher-suites`
: Provides a list of ciphers that are accepted by the client when trying to initiate a connection with TLS. This attribute must be provided as an array. The default cipher suites are:
`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`, `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`, `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`, `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384`.

Server configuration example

```
server:
    port: 9443
    host: localhost
    protocol: https
    connection-timeout: 30s
    tls:
      keystore: conf/nevisfido-server-keystore.p12
      keystore-passphrase: password
      keystore-type: pkcs12
      key-alias: nevisfido
      supported-protocols:
        - TLSv1.2
      cipher-suites:
        - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
        - TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
        - TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
        - TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        - TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

4.1.2. Management Configuration

Configuration of liveness and readiness probes.

The configuration described in this section are experimental and can change in future releases.

management
: The root node of the management configuration.

server
: Main node of the management server configuration properties.

`port`
: Port where the management server listens. Currently this includes the readiness probe.

healthchecks
: Main node for health probe configurations.

`enabled`
: Enables liveness and readiness probes.

Management configuration example

```
management:
  server:
    port: 9452
  healthchecks:
    enabled: false
```

4.1.3. FIDO UAF Configuration

All UAF related configuration is located in the `fido-uaf` configuration section. This includes the UAF specific configuration options and files.

fido-uaf
: The root node of the FIDO UAF application configuration. It defines the application ID associated with this nevisFIDO server as well as the application's *trusted facets*. See FIDO AppID and Facet Specification for more information.

`app-id`
: The application ID.

`facets`
: The facets associated with the application ID. This is an array of strings.
Note that changing the contents of the facets configuration in the `nevisfido.yml` file are automatically taken into account without requiring a server restart.
The default metadata contains the definitions of the NEVIS Mobile Authentication authenticators.

metadata
: The configuration of the metadata that nevisFIDO uses to process the requests.

`path`
: The path (file or directory) containing the metadata. If the path is a directory, nevisFIDO reads all the files in the directory and aggregates its content. This allows you to update the metadata by adding/removing/updating single files. nevisFIDO expects that each of the files contains one JSON object. The JSON object is either an object defining a single Metadata Statement or a JSON array with several metadata statements.

`polling-period`
: Determines how often nevisFIDO checks whether metadata content has changed. For example, if `polling-period` is set to "120s", then nevisFIDO checks the metadata content every two minutes for changes. The default value is 5 seconds. If no time unit is provided, seconds will be used.

policy
: The configuration of the policy that nevisFIDO uses to process requests. Currently nevisFIDO only supports one policy for all requests.
The default nevisFIDO policy accepts the NEVIS Mobile Authentication authenticators.

- In case of a *registration request* from a user that already has registered other credentials, nevisFIDO adds one non-acceptable match criteria to each of the already registered credentials in the policy. This is to prevent the user from reregistering an already registered credential.

- In case of a *step-up authentication*, nevisFIDO validates the user against the policy. If at least one credential is registered for the provided username, nevisFIDO returns a policy that only contains the user's AAID and KeyIDs. If no valid credentials are found, the policy is sent as defined in the configuration file.

  `path`

  The path to the file containing the policy. nevisFIDO expects that the file contains one JSON object. The JSON object is a single Policy.

timeout

The configuration of the client timeouts.

`registration`

Defines the maximum time duration between the generation of the `RegistrationRequest` by nevisFIDO and the `RegistrationResponse` by the FIDO UAF client. If the client has not sent the response after this time, a client timeout occurs. The default value is 5 minutes. If no time unit is provided, seconds will be used.

`authentication`

Defines the maximum time duration between the generation of the `AuthenticationRequest` by nevisFIDO and the `AuthenticationResponse` by the FIDO UAF client. If the client has not sent the response after this time, a client timeout occurs. The default value is 2 minutes. If no time unit is provided, seconds will be used.

`token-registration`

Defines the maximum time a client has to redeem a registration token after the generation of the token by nevisFIDO. Once the token is redeemed, the value of the `registration` timeout applies: from the moment of the redemption of the token the client has a maximum time (the `registration` timeout) to send a `RegistrationResponse` to nevisFIDO). The default value is 5 minutes. If no time unit is provided, seconds will be used.

`token-authentication`

Defines the maximum time a client has to redeem an authentication token after the generation of the token by nevisFIDO. Once the token is redeemed, the value of the `authentication` timeout applies: from the moment of the redemption of the token the client has a maximum time (the `authentication` timeout) to send an `AuthenticationResponse` to nevisFIDO). The default value is 2 minutes. If no time unit is provided, seconds will be used.

`token-deregistration`

Defines the maximum time a client has to redeem a deregistration token after the generation of the token by nevisFIDO. The default value is 2 minutes. If no time unit is provided, seconds will be used.

transaction-confirmation

The configuration of the transaction confirmation.

`max-text-length`

The maximum length of the transaction content when the content type of the transaction is `text-plain`. For example, if the transaction is of type text-plain, the value of `max-text-length` is 1000 and the content of the transaction in base64 URL encoded format has more than 1000 characters, the transaction will result in an error. The default value is 200 (as defined in the FIDO UAF specification). The minimum value is 200, the maximum value is 2000. See Transaction Confirmation for more details.

FIDO UAF configuration example

```
fido-uaf:
    app-id: "https://www.siven.ch/appID"
    facets:
        - "https://register.siven.ch"
        - "android:apk-key-hash:324234234"
    metadata:
        path: conf/metadata
    policy:
        path: conf/policy/policy.json
    timeout:
        registration: 30s
        authentication: 30s
        token-registration: 30s
        token-authentication: 30s
        token-deregistration: 30s
```

4.1.4. Credential Repository Configuration

The credential repository holds the registered credentials. nevisFIDO needs access to this repository to be able to perform the following actions:

- Adding credentials (during the registration operation).
- Reading credentials (notably during the authentication operation).
- Removing credentials (during the deregistration operation).

You specify the configuration of the credential repository with the `credential-repository` attribute:

credential-repository
: The root node of the credential repository configuration.

`type`
: Defines the type of the credential repository.

Currently, you can configure two types of credential repositories, the *in-memory* and the *nevisIDM* types of credential repository.

4.1.4.1. In-Memory Credential Repository

In case of an *in-memory* type of credential repository, the registered credentials are stored in the memory. The registered credentials will be lost when you stop nevisFIDO. This type of credential repository is therefore mainly used for testing purposes.

To define an in-memory type of credential repository, set the value of the `type` attribute to "in-memory".
The in-memory repository does not have additional configuration options.

In-memory credential repository configuration example

```
credential-repository:
    type: in-memory
```

4.1.4.2. nevisIDM Credential Repository

In case of a *nevisIDM* type of credential repository, the FIDO UAF credentials are stored in nevisIDM.

If you select this type of repository, nevisFIDO needs key material to connect to nevisIDM. The connection is based on TLS Client Authentication. This means that both parties require a correctly set up private key and public

certificate to be able to trust each other. The public certificate of nevisIDM has to be imported into nevisFIDO's truststore and vice versa.

nevisFIDO has 60 seconds to establish connections to and execute requests with nevisIDM before a timeout will occur.

To define a nevisIDM type of credential repository, set the value of the `type` attribute to "nevisidm".

The nevisIDM credential repository has the following additional configuration attributes:

`administration-url`
: Defines the URL pointing to the nevisIDM `AdminService`. For example: `https://nevisidm.siven.ch:8443/nevisidmcc/services/v1/AdminService`

`keystore`
: Defines the path to the keystore holding the nevisFIDO client certificate. nevisIDM must have a technical user whose certificate credential contains the public key of this nevisFIDO client certificate.

`keystore-passphrase`
: Sets the password of the keystore. It is assumed that this keystore password can be used to access the key. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`keystore-type`
: Defines the type of keystore. A "pkcs12" type of keystore is recommended. (Using the legacy Java key-/truststore "jks" is possible but discouraged.) If you do not specify this attribute, the system will use the default keystore type of the Java Virtual Machine that runs nevisFIDO.

`truststore`
: Path to the truststore containing the nevisIDM public key.

`truststore-passphrase`
: Defines the password used to access the truststore. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`truststore-type`
: Defines the type of truststore. A "pkcs12" truststore type is recommended. (Using the legacy Java key-/truststore "jks" is possible but discouraged.) If you do not specify this attribute, the system will use the default truststore type of the Java Virtual Machine that runs nevisFIDO.

`admin-service-version`
: Defines the version of the Admin Service SOAP interface to be used to communicate with nevisIDM. The default value is "v1_42".

`client-id`
: Specifies the ID of the nevisIDM client that contains the relevant data. Either this parameter or the `client-name` parameter is required if nevisIDM is set up in multi-client mode. If both `client-id` and `client-name` are available, then the system will use `client-id`. It is not necessary to provide this parameter if the multi-client mode is not enabled in nevisIDM.

`client-name`
: Specifies the name of the nevisIDM client that contains the relevant data. Either this parameter or the `client-id` parameter is required if nevisIDM is set up in multi-client mode. If both `client-id` and `client-name` are available, then the system will use `client-id`. It is not necessary to provide this parameter if the multi-client mode is not enabled in nevisIDM.

`username-mapper`
: Specifies how the `username` provided by the HTTP clients in the context of the `GetUAFRequest` (for registration, authentication and deregistration) is mapped to a user in nevisIDM.

  `attributes`
  : Defines the attributes that the server will use to retrieve the user entry in nevisIDM. The algorithm goes through the provided attributes in order and returns the first user whose attribute value is as `username` by the

client. If several entries have the `username` value in the attribute, those entries will be discarded. This is why the mapping only works if the attribute values are unique. The allowed values are `loginId`, `email` and `extId`.
If not defined, the `extId` attribute will be used as value of the attribute. Note that `extId` is guaranteed to be unique, but it might requiring some mapping in the client front to be used, because in general the end-user does not know it.

For example, if the value of `attributes` is the list [`loginId`, `email`], and the value of the `username` in the `GetUAFRequest` is `jeff@company.com`, the server will try to retrieve first the user whose login ID is `jeff@company.com`, and if the user is not found, it will search for a user whose email is `jeff@company.com`.

The username mapper is experimental and there are no guarantees that it will be kept in the future.

Configuration example:

nevisIDM credential repository configuration example

```
credential-repository:
    type: nevisidm
    administration-url: https://nevisauth-fido-test1.zh.nevis-
security.com:8443/nevisidmcc/services/v1/AdminService
    keystore: /var/opt/nevisfido/default/conf/keystore.p12
    keystore-passphrase: password
    keystore-type: pkcs12
    truststore: /var/opt/nevisfido/default/conf/truststore.p12
    truststore-passphrase: password
    truststore-type: pkcs12
    admin-service-version: v1_42
    client-id: 100
    client-name: nevis
    username-mapper:
        attributes:
            - loginId
            - email
```

4.1.5. Dispatch Target Repository Configuration

nevisFIDO needs to access a repository containing the dispatch targets to be able to cover the out-of-band-case (where a token is sent to a device described by the dispatch target). Currently two types of dispatch target repositories can be configured, the *in-memory* and the *nevisIDM* types of dispatch target repository.

dispatch-target-repository

The root node of the dispatch target repository configuration.

`type`

The type of the dispatch target repository

4.1.5.1. In-memory Dispatch Target Repository

In case of an *in-memory* type of dispatch target repository, the dispatch targets are stored in the memory. The dispatch targets will be lost when you stop nevisFIDO. This type of dispatch target repository is therefore mainly used for testing purposes.

To define an in-memory type of dispatch target repository, set the value of the `type` attribute to "in-memory". The in-memory repository does not have additional configuration attributes.

In-memory dispatch target repository configuration example

```
dispatch-target-repository:
  type: in-memory
```

4.1.5.2. nevisIDM Dispatch Target Repository

In case of a *nevisIDM* type of dispatch target repository, the FIDO UAF dispatch targets are stored in nevisIDM.

If you select this type of repository, nevisFIDO needs key material to connect to nevisIDM. The connection is based on TLS Client Authentication. This means that both parties require a correctly set up private key and public certificate to be able to trust each other. The public certificate of nevisIDM has to be imported into nevisFIDO's truststore and vice versa.

nevisFIDO has 60 seconds to establish connections to and execute requests with nevisIDM before a timeout will occur.

To define a nevisIDM type of dispatch target repository, set the value of the `type` attribute to "nevisidm".

The nevisIDM dispatch target repository has the following additional configuration attributes:

`administration-url`
: Defines the URL pointing to the nevisIDM `AdminService`. For example: `https://nevisidm.siven.ch:8443/nevisidmcc/services/v1/AdminService`

`keystore`
: Defines the path to the keystore holding the nevisFIDO client certificate. nevisIDM must have a technical user whose certificate dispatch target contains the public key of this nevisFIDO client certificate.

`keystore-passphrase`
: Sets the password of the keystore. It is assumed that this keystore password can be used to access the key. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`keystore-type`
: Defines the type of keystore. A "pkcs12" type of keystore is recommended. (Using the legacy Java key-/truststore "jks" is possible but discouraged.) If you do not specify this attribute, the system will use the default keystore type of the Java Virtual Machine that runs nevisFIDO.

`truststore`
: Path to the truststore containing the nevisIDM public key.

`truststore-passphrase`
: Defines the password used to access the truststore. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`truststore-type`
: Defines the type of truststore. A "pkcs12" truststore type is recommended. (Using the legacy Java key-/truststore "jks" is possible but discouraged.) If you do not specify this attribute, the system will use the default truststore type of the Java Virtual Machine that runs nevisFIDO.

`admin-service-version`
: Defines the version of the Admin Service SOAP interface to be used to communicate with nevisIDM. The default value is "v1_42".

`client-id`
: Specifies the ID of the nevisIDM client that contains the relevant data. Either this parameter or the `client-name` parameter is required if nevisIDM is set up in multi-client mode. If both `client-id` and `client-name` are available, then the system will use `client-id`. It is not necessary to provide this parameter if the multi-client mode is not enabled in nevisIDM.

`client-name`
: Specifies the name of the nevisIDM client that contains the relevant data. Either this parameter or the `client-id`

parameter is required if nevisIDM is set up in multi-client mode. If both `client-id` and `client-name` are available, then the system will use `client-id`. It is not necessary to provide this parameter if the multi-client mode is not enabled in nevisIDM.

`username-mapper`

Specifies how the `username` provided by the HTTP clients in the create and query dispatch target requests is mapped to a user in nevisIDM.

`attributes`

Defines the attributes that the server will use to retrieve the user entry in nevisIDM. The algorithm goes through the provided attributes in order and returns the first user whose attribute value is as `username` by the client. If several entries have the `username` value in the attribute, those entries will be discarded. This is why the mapping only works if the attribute values are unique. The allowed values are `loginId`, `email` and `extId`.
If not defined, the `extId` attribute will be used as value of the attribute. Note that `extId` is guaranteed to be unique, but it might requiring some mapping in the client front to be used, because in general the end-user does not know it.

For example, if the value of `attributes` is the list [`loginId`, `email`], and the value of the `username` in the `GetUAFRequest` is `jeff@company.com`, the server will try to retrieve first the user whose login ID is `jeff@company.com`, and if the user is not found, it will search for a user whose email is `jeff@company.com`.

The username mapper is experimental and there are no guarantees that it will be kept in the future.

Configuration example:

nevisIDM dispatch target repository configuration example

```
dispatch-target-repository:
  type: nevisidm
  administration-url: https://nevisauth-fido-test1.zh.nevis-
security.com:8443/nevisidmcc/services/v1/AdminService
  keystore: /var/opt/nevisfido/default/conf/keystore.p12
  keystore-passphrase: password
  keystore-type: pkcs12
  truststore: /var/opt/nevisfido/default/conf/truststore.p12
  truststore-passphrase: password
  truststore-type: pkcs12
  admin-service-version: v1_42
  client-id: 100
  client-name: nevis
  username-mapper:
    attributes:
      - loginId
      - email
```

4.1.6. Session Repository Configuration

Sessions managed by nevisFIDO are stored either in the memory or in a (MariaDB) SQL database.

You specify the configuration of the session storage with the `session-repository` attribute:

session-repository

The root node of the session repository configuration.

`type`

The type of the session repository. The default value is "in-memory".

4.1.6.1. In-Memory Session Repository

The in-memory session repository is selected by setting "in-memory" as the repository `type`. This type of session repository has no other configuration options.

In-memory session repository configuration example

```
session-repository:
    type: in-memory
```

The in-memory session repository is the default. It will also be selected if the above configuration is omitted.

4.1.6.2. SQL Session Repository

The SQL session repository is selected by configuring "sql" as the repository `type`. nevisFIDO comes bundled with MariaDB client capability. MariaDB is the officially supported SQL storage backend.

The SQL session repository needs a JDBC URL for locating the database, and a user and password for authentication. nevisFIDO will automatically generate the required schema and tables in the specified database. It is recommended creating a dedicated MariaDB database to store the nevisFIDO session information. The required attributes are:

`jdbc-url`
: The JDBC URL to the MariaDB database.

`user`
: The user name needed to access the database.

`password`
: The password needed to access the database. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`schema-user`
: The user name of the user who creates the schema and the tables in the database. If you do not provide a schema user here, the system will use the user specified in the attribute `user` to create the schema. However, it is recommended having different users for creating the schema and for accessing the data.
NOTE: If you do not set the `schema-user` attribute, do not set the `schema-user-password` attribute either. Otherwise the system will throw an error. This is because the system takes the `user` as fallback if no `schema-user` is provided. If in this case both the password attributes `schema-user-password` and `password` are set, there are two passwords available for the `user` attribute, which is ambiguous.

`schema-user-password`
: The password of the `schema-user`, that is, the user who creates the schema and the tables in the database. If you do not provide the schema user password, the system takes the password specified in the attribute `password` as fallback. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`automatic-db-schema-setup`
: If set to `true`, the schema and table used to store the session data will be created automatically by nevisFIDO. Set this property to `false` if the database must be not be set-up automatically. If not specified, the default value is `true`.

`max-connection-lifetime`
: Defines the maximum time that a database connection remains in the connection pool. Although having a high value improves the performance, it should be lower than the connection idle timeout of the database (parameter *wait_timeout* in MariaDB). If not specified, the default value is 30 minutes. If no time unit is provided, seconds will be used.

SQL session repository configuration example

```
session-repository:
    type: sql
    jdbc-url: 'jdbc:mariadb://localhost:3306/db'
    user: data-user
    password: secret
    schema-user: schema-user
    schema-user-password: schema-secret
    automatic-db-schema-setup: true
    session-lifetime: 25m
```

SQL session repository user creation

To be able to create the tables and define the schema, the schema user must have `CREATE` privileges in the database. To be able to read, update and remove the session information, the user must have `SELECT`, `INSERT`, `UPDATE` and `DELETE` privileges in the database.

The following sample configuration uses SQL commands to create two separate users and grant them the required privileges. In the sample, the database where you store the session information is called `nevisfido`. The database and nevisFIDO are located on the same machine. The created users match the users from the previous sample configuration (SQL session repository).

```
CREATE USER 'data-user'@'localhost' IDENTIFIED BY 'secret';
CREATE USER 'schema-user'@'localhost' IDENTIFIED BY 'schema-secret';
GRANT SELECT, INSERT, UPDATE, DELETE ON nevisfido.* TO 'data-user'@'localhost';
GRANT CREATE ON nevisfido.* TO 'schema-user'@'localhost';
FLUSH PRIVILEGES;
```

4.1.6.2.1. Session Reaping

When the SQL backend session repository is in use, a session reaping mechanism cleans up old sessions in the database to prevent the table from growing unlimitedly in size.

Sessions are reaped after *three times the timeout of the client session*. For example, if the default client timeout for a registration operation is 30 seconds, the registration session will be reaped after 90 seconds.

For client session timeout values, refer to FIDO UAF Configuration.

4.1.6.3. Resilient SQL Session Repository

This chapter describes how to create a fault tolerant setup for database- or network outages between nevisFIDO and MariaDB. (At least one database node must be available to prevent application failure.)

4.1.6.3.1. Use cases

Failure

On the primary DB node, failure connections will move to the secondary DB node when a 30 second timeout expires or immediately in case of an incoming request to the DB. You may experience increased response time for the duration of the switch.

Recovery

Recovery connections will move back to the primary DB node once the connection maximum lifetime expires. The connection maximum lifetime is 30 minutes.

4.1.6.3.2. Implementation overview

Regular clustering solutions provide all fault tolerant features themselves. The connecting application is not aware of the resilient setup.

The suggested solution takes a different approach. In this setup, the connecting application becomes part of the resilient setup in terms of configuration.

There are two key features to achieve resilience:

1.) Connectivity (MariaDB JDBC driver) Configure a JDBC url where you define the DB nodes in priority order:
`jdbc:mariadb:sequential://host-db1:3306,host-db2:3306/nevisfido`

2.) Data consistency (MariaDB replication)
Configure Master-to-Master replication.

Replication is done by the database, the application and the JDBC driver are not aware of it.

4.1.6.3.3. Overview of database users

The replicated session store is managed by several database users in order to separate concerns. The creation of the users is explained below.

Username	Purpose	Required permissions
replication_user	Account used by the slave node to log in to the master node.	REPLICATION SLAVE to be able to replicate data.
binarylog_user	Account used for binary log management.	SUPER to be able to purge the binary logs.

4.1.6.3.4. Step-by-step setup of the replicated session store

This chapters assumes that the SQL session repository user creation is completed and the tables used by nevisFIDO are created.

1.) Creation of the replication user:

```
CREATE USER IF NOT EXISTS replication_user IDENTIFIED BY 'replicationpassword';
GRANT REPLICATION SLAVE ON *.* TO replication_user;
```

2.) Creation of the binary logs user:

```
CREATE USER IF NOT EXISTS binarylog_user IDENTIFIED BY 'binarylogspassword';
GRANT SUPER ON *.* TO binarylog_user;
```

3.) Configuration of MariaDB service.
To configure the MariaDB service, add the following entries to the file */etc/my.cnf* as super user. The two configuration files (*host-db1* and *host-db2*) differ at some points. The different lines are marked with (*).

3.1) Configure the MariaDB service on *host-db1*:

```
[mariadb]
# Enabling binary log
log-bin
# The ID of this master (*)
server_id=1
# The ID of the replication stream created by this master (*)
gtid-domain-id=1
# The basename and format of the binary log
log-basename=mariadbmaster
binlog-format=MIXED
# Setting which tables are replicated
replicate_wild_do_table="nevisfido.%"
# Avoiding collisions of primary IDs for tables where the primary ID is auto-
incremented
# Auto-increment value
auto_increment_increment=2
# Auto-increment offset (*)
auto_increment_offset=1
# Suppressing duplicated keys errors for multi-master setup
slave_exec_mode=IDEMPOTENT
# Ignoring some data definition language errors
slave-skip-errors=1452, 1062
# Suppressing binary logs after a delay regardless of the replication status
expire_logs_days=1
# Maximum number of connections
max_connections=1000
# Size of each of the binary log files (default: 1GB)
max_binlog_size=500M
# Enabling writing to the DB in parallel threads for the replication
slave-parallel-threads=10
# enabling semi-synchronous replication
rpl_semi_sync_master_enabled=ON
rpl_semi_sync_slave_enabled=ON
# change to READ COMMITTED
transaction-isolation=READ-COMMITTED
```

3.2) Configure the MariaDB service on *host-db2*:

```
[mariadb]
# Enabling binary log
log-bin
# The ID of this master (*)
server_id=2
# The ID of the replication stream created by this master (*)
gtid-domain-id=2
# The basename and format of the binary log
log-basename=mariadbmaster
binlog-format=MIXED
# Setting which tables are replicated
replicate_wild_do_table="nevisfido.%"
# Avoiding collisions of primary IDs for tables where the primary ID is auto-
incremented
# Auto-increment value
auto_increment_increment=2
# Auto-increment offset (*)
auto_increment_offset=2
# Suppressing duplicated keys errors for multi-master setup
slave_exec_mode=IDEMPOTENT
# Ignoring some data definition language errors
slave-skip-errors=1452, 1062
# Suppressing binary logs after a delay regardless of the replication status
expire_logs_days=1
# Maximum number of connections
max_connections=1000
# Size of each of the binary log files (default: 1GB)
max_binlog_size=500M
# Enabling writing to the DB in parallel threads for the replication
slave-parallel-threads=10
# enabling semi-synchronous replication
rpl_semi_sync_master_enabled=ON
rpl_semi_sync_slave_enabled=ON
# change to READ COMMITTED
transaction-isolation=READ-COMMITTED
```

3.3) Restart the MariaDB servers on both hosts:

```
sudo service mariadb restart
```

4.1.6.3.5. Semi-synchronous replication

By default, MariaDB uses asynchronous replication. In order to reach more consistency, it is recommended using semi-synchronous replication.

The database configurations previously shown enable semi-synchronous replication with the following lines:

```
rpl_semi_sync_master_enabled=ON
rpl_semi_sync_slave_enabled=ON
```

MariaDB versions before 10.3.3 require the installation of plug-ins for semi-synchronous replication and are not supported.

4.1.6.3.6. Replication start

In order to start the replication, log in as root into your MariaDB client and run the following commands.

1.) On *host-db1* (master is *host-db2*):

```
CHANGE MASTER TO
  MASTER_HOST='host-db2',
  MASTER_USER='replication_user',
  MASTER_PASSWORD='replicationpassword',
  MASTER_PORT=3306,
  MASTER_USE_GTID=current_pos,
  MASTER_CONNECT_RETRY=10;
```

2.) On *host-db2* (master is *host-db1*):

```
CHANGE MASTER TO
  MASTER_HOST='host-db1',
  MASTER_USER='replication_user',
  MASTER_PASSWORD='replicationpassword',
  MASTER_PORT=3306,
  MASTER_USE_GTID=current_pos,
  MASTER_CONNECT_RETRY=10;
```

3.) On *host-db1*:

```
START SLAVE;
```

4.) On *host-db2*:

```
START SLAVE;
```

4.1.6.3.7. Additional setup

Purging the binary logs

With the provided configuration (*expire_logs_days=1* in the MariaDB settings), the system will automatically remove binary logs that are older than one day, even if the logs were not copied by the slave. This prevents the disk of the master node from being filled up in case the slave is down for a long time.

The automatic binary log removal takes place when

- the master DB node starts,
- the logs are flushed (nevisFIDO does not use this feature),
- the binary log rotates, or
- the binary logs are purged manually (see below).

This means that binary logs older than one day can exist, if none of the listed actions occurred recently.

Complementary to this expiration feature, MariaDB provides the possibility to manually purge the binary logs. The purge action removes all binary logs that were already copied by the slave. This allows a safe removal of the binary logs on a regular basis.

The nevisProxy package is delivered with an adaptable purging script, which is located at:
/opt/nevisproxy/sql/mariadb/purgebinarylogs.sh

In order to use this script,

- copy the script to a location of your choice, and
- adapt it to your configuration.

The script takes care of both DB nodes, so that it only needs to be configured once.

If you use different database server nodes for nevisProxy and nevisAuth, you have to set them up separately.

You can schedule the script to run for example once per hour, with a cron job:

```
0 * * * * /var/opt/nevisproxy/instance/conf/purgebinarylogs.sh # Absolute path
of your adapted script
```

Size the binary logs

The provided configuration (*max_binlog_size=500M* in the MariaDB settings) allows you to configure the maximal size of the binary log files before rotating. The smaller the size, the more often rotations occur, which will slow down replication. The advantage is a more efficient purge process. The bigger the size, the less often rotations occur, but the disk may be filled with old logs.

According to our experiences, a size less than 8K does stop replication completely under heavy load, because the slave keeps rotating the logs.

Troubleshooting

Usually the slave stops replication if an error occurs. You can check the state of the slave with the following SQL command:

```
show slave status\G
```

Showing the slave status requires the REPLICATION CLIENT grant.

If replication has stopped, usually the error that caused it will be displayed. First you should try to fix the error. If this is not possible, you can do a "forced" restart of the slave like this:

- On the master call (in order to display the current state of the master):

```
MariaDB [replicated_session_store]> show master status\G
*************************** 1. row ***************************
            File: mariadbmaster-bin.000131
        Position: 194630804
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

- On the slave (using the values returned by the call *show master status\G* on the master):

```
STOP SLAVE;
CHANGE MASTER TO
  MASTER_LOG_FILE='mariadbmaster-bin.000131',
  MASTER_LOG_POS=194630804;
START SLAVE;
```

In this way, the system will restart the slave, without replicating to the slave all sessions that occurred from the

moment the replication has stopped until now.

4.1.7. Authorization

nevisFIDO provides several authorization validation mechanisms for each operation, in order to meet the operations' different security requirements. For example, the deregistration operation usually requires authorization, to ensure that only known clients can deregister a FIDO UAF credential. To trigger authentication, however, authorization is usually not required.

The Modify Dispatch Target service uses JWS to verify whether a request is sent by a trusted client. Therefore, the authorization validation configuration does not apply to this service.

Configuration example:

```
authorization:

    # The username will be retrieved from the SecToken for the registration
requests
    registration:
        type: sectoken
        truststore: /var/opt/nevisfido/default/conf/truststore-with-nevisauth-
signature-keys.p12
        truststore-passphrase: password
        truststore-type: pkcs12
        username-attribute-names:
            - userid

    # The username will be retrieved from the Context for the authentication
requests
    authentication:
        type: none

    # The username will be retrieved from the SecToken for the deregistration
requests
    deregistration:
        type: sectoken
        truststore: /var/opt/nevisfido/default/conf/truststore-with-nevisauth-
signature-keys.p12
        truststore-passphrase: password
        truststore-type: pkcs12
        username-attribute-names:
            - userid
```

Currently, you can configure two authorization validation modes: Authorization validation with SecToken, or no authorization validation.

4.1.7.1. SecToken Authorization

The SecToken authorization mode expects that the username is provided in the HTTP headers, as part of the `SecToken`. The `SecToken` is validated by nevisFIDO. For the authorization to be successful, the username in the `SecToken` must match the username in the request (which is provided, for example, in the context of the `GetUAFRequest`).

Assuming that the username is provided in the `SecToken` for registration operations, the process is as follows:

1. The FIDO UAF client tries to register a credential.
2. nevisProxy detects that the FIDO UAF client is not authenticated. It redirects the client to nevisAuth for

authentication.
3. nevisAuth generates a `SecToken` upon successful authentication of the client.
4. The authenticated FIDO UAF client will now be able to access the registration endpoint of nevisFIDO. nevisProxy will include the `SecToken` generated in the previous step in the `Basic` authorization header.
5. nevisFIDO validates the `SecToken`, by verifying that the token was signed by nevisAuth and is not expired.
6. nevisFIDO also checks whether the `SecToken` contains the username provided in the request (for example, in the context of the `GetUAFRequest` or in the `username` attribute of the "Create dispatch target" request).

The `Basic` authorization header has the following value:

```
Basic <base64Value>
```

where `<base64Value>` is a string encoded with base64 and with the following format:

```
<username>:<sectoken>
```

Note that nevisFIDO does *not* take into account the first part of the base64 value (`username`), but retrieves the username from the `SecToken`. The reason for this is that the username used to log in may differ from the username required in nevisFIDO. The content of the `SecToken` is configurable, thus allowing for flexibility in scenarios where one user can have several user identifiers.

However, it is still recommended that you include a `username` in the `Basic` authorization header. This is because nevisProxy must provide two values to meet the definition of a correct `Basic` header.

See nevisProxy Configuration for configuration details.

The use of a `SecToken` implies a preceding validation of the user identity by nevisAuth. So for security reasons, we recommend the usage of this type of authorization for non-authentication operations (registration, deregistration, creation and deletion dispatch targets).

To enable a SecToken authorization for a specific operation, you must set the value of the `type` attribute to `sectoken`.

The SecToken authorization has the following additional configuration attributes:

`truststore`
: Defines the path to the truststore with the public key that is used by nevisAuth to sign the `SecToken`.

`truststore-passphrase`
: The password needed to access the truststore. Use the mechanisms described in Application Configuration to avoid providing a plaintext password.

`truststore-type`
: The type of the truststore. The recommended type is "pkcs12". (Using the legacy Java key-/truststore "jks" is possible but discouraged.) If this attribute is not set, the system will use the default truststore type of the Java Virtual Machine running nevisFIDO.

`username-attribute-names`
: Defines the attributes that the server will use to retrieve the username. If any of the `SecToken` attributes listed in this property contains the expected username, then the request will be authorized. By default, the value is `userId`.
Typically, the authorized user and the user associated with the nevisFIDO operation are the same. For example, the authorized user is also the user whose FIDO credentials must be deleted. The authorized user is normally the user defined by the `userId` attribute of the `SecToken`.
But in some cases, the authenticated user must perform an operation on behalf of another user. For example, an administrator must delete the FIDO credentials of a given user whose user device was lost or stolen. In this case, the username associated with the operation is not the user defined in the `SecToken` (that is, the `userId` attribute of the `SecToken` does not contain the username of the FIDO credentials to be deleted).

To be able to cover both use cases, the configuration attribute `username-attribute-names` can take several values.

There is no username provided in the delete dispatch target request, so the SecToken authorization works in a different way as for the other endpoints: the element that is used to validate that the request is authorized is the user identifier used by the repository (`extId` when the user is stored in nevisIDM) of the user owning the dispatch target to be deleted. This is why the value of the `username-attribute-names` attribute of the SecToken configuration should contain in most cases the value `userId`; with this configuration the userId in the SecToken is compared to the user identifier (as stored by the repository) of the owner of the dispatch target.

Configuration example:

```
authorization:
    registration:
        type: sectoken
        truststore: /var/opt/nevisfido/default/conf/truststore-with-nevisauth-
signature-keys.p12
        truststore-passphrase: password
        truststore-type: pkcs12
        username-attribute-names:
            - userid
```

4.1.7.2. No Authorization Validation

*It is **not** recommended using the "no authorization" validation mode for non-authentication operations, such as registration, deregistration as well as creating and deleting dispatch targets. This is because the "no authorization" validation mode is not secure. It allows everyone to perform an operation and thus makes impersonation possible. For example, the malicious client "Mallory" could send a registration request providing Alice's name. This would generate credentials in Mallory's FIDO device that would be associated with Alice, allowing Mallory to authenticate as Alice.*

To not perform authorization validation for a specific operation, set the value of the `type` attribute to `none`.

Configuration example:

```
authorization:
    authentication:
        type: none
```

4.1.8. Dispatchers Configuration

nevisFIDO can generate and dispatch tokens through one of the configured dispatchers (see Dispatch Token Service). The HTTP client specifies the dispatcher to be used in the HTTP request to generate the token.

The dispatching mechanism is used in the out-of-band case. In this case, one device triggers the FIDO operation by asking to generate a token. Then the token is dispatched to another device through a dispatcher. This FIDO UAF enabled device redeems the token and proceeds with the FIDO UAF operation. Usually, the device that triggers the operation is a desktop computer whereas the device receiving the token is a mobile device with FIDO UAF capabilities.

A typical example of a dispatcher is the FCM (Firebase Cloud Messaging) Dispatcher, which allows to dispatch tokens to a mobile device. The FCM dispatcher is provided by nevisFIDO.

nevisFIDO allows to configure several dispatchers. The HTTP client specifies the dispatcher to be used by providing the value of the `type` attribute. Currently only one dispatcher for a given type can be configured (that is, you cannot configure two FCM dispatchers).

Configuration example:

```
dispatchers:
    - type: logger
```

Expression resolution in dispatcher configuration has a limitation, which is only relevant for developers implementing custom dispatchers. Expressions resulting in integer or Boolean values require special escaping to result in strings. For example, assuming that the value of the environment variable BOOLEAN_VALUE is "true":

```
dispatchers:
    - type: logger
      value: ${ENV:BOOLEAN_VALUE}
```

`value` is boolean "true".

```
dispatchers:
    - type: logger
      value: "${ENV:BOOLEAN_VALUE}"
```

`value` is boolean "true".

```
dispatchers:
    - type: logger
      value: '"${ENV:BOOLEAN_VALUE}"'
```

`value` is string "true".

4.1.9. Application Configuration Example

The following example shows the full default configuration provided by nevisFIDO:

Default configuration file

```
# The default nevisFIDO configuration permits to have a running nevisFIDO
instance with minimal
# (if any) changes.  Checking that the specified server port and server host
properties are valid
# is enough to start the nevisFIDO server and do basic testing.
#
# Note that this default configuration is only recommended for testing.  The
configuration to use in
# production requires additional configured elements, such as certificates and
a running nevisIDM
# instance.  Examples of the recommended configuration are included in
comments. See the
# configuration blocks for details.

# The server configuration contains the main web server configuration.
server:
    port: 9080
    host: localhost
    protocol: http
    connection-timeout: 30s
    # Example of enabled TLS configuration, which is recommended. First, enable
the HTTPS protocol:
```

```
    # protocol: https
        # Then configure TLS properties:
        # tls:
            # All these properties must be provided, each helping to point to
the server's private key.
            # keystore: conf/nevisfido-server-keystore.p12
            # keystore-passphrase: password
            # keystore-type: pkcs12
            # key-alias: nevisfido
            # The following two properties are optional, below the default
values are included.
            # supported-protocols:
            #   - TLSv1.2
            # cipher-suites:
            #   - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
            #   - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
            #   - TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
            #   - TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
            #   - TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
            #   - TLS_DHE_RSA_WITH_AES_256_GCM_SHA384

# Management configuration for readiness and liveness endpoints
management:
    server:
        port: 9089
    healthchecks:
        enabled: false

# Specifies where the credentials are stored.  Currently two types of
credential repositories
# are supported: nevisIDM and in-memory.  Note that the in-memory repository
has no persistence and
# should only be used for testing purposes.
credential-repository:
    type: in-memory # Allowed values: in-memory, nevisidm
    # Example of nevisidm credential repository:
    # type: nevisidm
    # administration-url:
https://localhost:8443/nevisidmcc/services/v1/AdminService
    # keystore: conf/nevisfido-client-keystore.p12
    # keystore-passphrase: password
    # keystore-type: pkcs12
    # truststore: conf/nevisfido-truststore.p12
    # truststore-passphrase: password
    # truststore-type: pkcs12
    # admin-service-version: v1_42
    ## The name of the client required to access the data in nevisIDM. Only for
multi-client mode.
    # client-name: the-client-name
    # client-id: 100
    ## Specifies how the `username` provided by the HTTP clients in the request
is
    # mapped to a user in nevisIDM.
    # username-mapper:
    #   attributes:
    #     - loginId
```

```
# Specifies where the dispatch targets are stored.  Currently two types of
dispatch target
# repositories are supported: nevisIDM and in-memory.  Note that the in-memory
repository has no
# persistence and should only be used for testing purposes.
dispatch-target-repository:
    type: in-memory # Allowed values: in-memory, nevisidm
    # Example of nevisidm dispatch target repository:
    # type: nevisidm
    # administration-url:
https://localhost:8443/nevisidmcc/services/v1/AdminService
    # keystore: conf/nevisfido-client-keystore.p12
    # keystore-passphrase: password
    # keystore-type: pkcs12
    # truststore: conf/nevisfido-truststore.p12
    # truststore-passphrase: password
    # truststore-type: pkcs12
    # admin-service-version: v1_42
    ## The name of the client required to access the data in nevisIDM. Only for
multi-client mode.
    # client-name: the-client-name
    # client-id: 100
    ## Specifies how the `username` provided by the HTTP clients in the request
is
    # mapped to a user in nevisIDM.
    # username-mapper:
    #   attributes:
    #     - loginId

# Configuration for the session repository (FIDO UAF sessions and token
sessions).
session-repository:
    type: in-memory  # Allowed values: in-memory, sql
    # jdbc-url: 'jdbc:mariadb://localhost:3306/db'
    # user: data-user
    # password: secret
    # schema-user: schema-user
    # schema-user-password: schema-password
    # automatic-db-schema-setup: true
    # max-connection-lifetime: 30m

# Defines the authorization to be used when handling GetUAFRequest requests.
Currently two types of
# authorization are supported:
# sectoken: extracts the username from a SecToken in the header and compares it
with
#           the one in the GetUAFRequest.
# none: do not perform any validation (all requests are authorized)
# Note that for each operation an authorization mode must be defined.
authorization:

    # No authorization will be performed for the registration requests.  Note
that this
    # is not the recommended configuration.  The SecToken authorization should
be used instead.
```

```
    registration:
        type: none
        # Example of SecToken authorization configuration:
        # type: sectoken
        # truststore: conf/nevisfido-truststore.p12
        # truststore-passphrase: password
        # truststore-type: pkcs12
        # username-attribute-names:
        #   - userid

    # No authorization validation will be performed for the authentication
requests
    authentication: # No authorization validation will be performed for the
authentication requests
        type: none # Type of the authorization validation. Allowed values:
none, sectoken

    # No authorization will be performed for the deregistration requests.  Note
that this
    # is not the recommended configuration.  The SecToken authorization should
be used instead.
    deregistration:
        type: none

    # No authorization will be performed for the creation of dispatch targets.
Note that this
    # is not the recommended configuration.  The SecToken authorization should
be used instead.
    create-dispatch-target:
        type: none

    # No authorization will be performed for the deletion of dispatch targets.
Note that this
    # is not the recommended configuration.  The SecToken authorization should
be used instead.
    # There is no username provided in the delete dispatch target request, so
the SecToken
    # authorization works in a different way as for the other endpoints: the
element that is used to
    # validate that the request is authorized is the user identifier used by
the repository ('extId'
    # when the user is stored in nevisIDM) of the user owning the dispatch
target to be deleted.
    # This is why the value of the 'username-attribute-names' attribute of the
SecToken
    # configuration should contain in most cases the value 'userId'; with this
configuration the
    # 'userId' in the SecToken is compared to the user identifier (as stored by
the repository) of
    # the owner of the dispatch target.
    delete-dispatch-target:
        type: none

    # No authorization will be performed for querying the dispatch targets.
    query-dispatch-target:
        type: none
```

```
fido-uaf:
    app-id: "https://www.siven.ch/appID"
    facets:
        - "https://register.siven.ch"
        - "https://fido.siven.ch"
        - "http://www.siven.ch"
        - "http://www.muvonda.com"
        - "https://www.siven.ch:444"
        - "android:apk-key-hash:<sha1_hash-of-apk-signing-cert>"
        - "ios:bundle-id:<ios-bundle-id-of-app>"
    metadata:
        path: conf/metadata
        # The metadata statements are dynamically reloaded with the period
provided here
        polling-period: 5s
    policy:
        path: conf/policy/policy.json
        # The policies are dynamically reloaded with the period provided here
        polling-period: 5s
    # client timeouts.  If no unit is provided, seconds is assumed
    timeout:
        # The maximum time that a FIDO UAF client has to send a
RegistrationResponse since nevisFIDO
        # generated a RegistrationRequest
        registration: 5m
        # The maximum time that a FIDO UAF client has to send an
AuthenticationResponse since nevisFIDO
        # generated an AuthenticationRequest
        authentication: 2m
        # The maximum time that a client has to redeem a registration token
since the token was
        # generated by nevisFIDO.
        token-registration: 5m
        # The maximum time that a client has to redeem an authentication token
since the token was
        # generated by nevisFIDO.
        token-authentication: 2m
        # The maximum time that a client has to redeem a deregistration token
since the token was
        # generated by nevisFIDO.
        token-deregistration: 2m
    transaction-confirmation:
        # The maximum length of the transaction content when the content type
of the transaction is
        # text-plain.
        max-text-length: 200

# The list of dispatchers that can be used to dispatch tokens.  The HTTP client
can specify a
# dispatcher to be used to dispatch a created token in the HTTP request by
providing the value
# of the type attribute. Currently only one dispatcher per type can be
configured.
dispatchers:
    - type: png-qr-code
#    - type: firebase-cloud-messaging
```

```
#      dry-run: false
#      service-account-json: conf/service-account.json
#
```

4.2. Metadata Configuration Example

nevisFIDO is supplied with authenticator metadata via JSON file(s). Each configuration contains one JSON object. This is either an object defining a single Metadata Statement or a JSON array with metadata statements.

A minimal metadata configuration **must** include the following attributes:

- `aaid`
- `description`
- `authenticatorVersion`
- `upv`
- `assertionScheme`
- `authenticationAlgorithm`
- `publicKeyAlgAndEncoding`
- `attestationTypes`
- `userVerificationDetails`
- `keyProtection`
- `matcherProtection`
- `attachmentHint`
- `isSecondFactorOnly`
- `tcDisplay`
- `attestationRootCertificates`

The possible configuration options are not explained in detail here. For more information, refer to the Metadata Statement specification. A helpful utility for exploring the official metadata repository is the FIDO Metadata Browser.

The following example shows a complete metadata configuration for one authenticator:

Metadata configuration example

```
[ {
  "aaid" : "0013#0001",
  "assertionScheme" : "UAFV1TLV",
  "attestationRootCertificates" : [ "MIIDAT..." ],
  "attestationTypes" : [ 15879, 15880 ],
  "description" : "ETRI SW Authenticator for SECP256R1_ECDSA_SHA256_Raw",
  "icon" : "...",
  "tcDisplayContentType" : "image/png",
  "tcDisplayPNGCharacteristics" : [ {
    "bitDepth" : 16,
    "colorType" : 2,
    "compression" : 0,
    "filter" : 0,
    "height" : 240,
    "interlace" : 0,
    "width" : 320
  } ],
  "upv" : [ {
    "major" : 1,
    "minor" : 0
  } ],
  "userVerificationDetails" : [ [ {
    "userVerification" : 4
  } ] ],
  "attachmentHint" : 1,
  "authenticationAlgorithm" : 1,
  "authenticatorVersion" : 1,
  "isSecondFactorOnly" : false,
  "keyProtection" : 1,
  "matcherProtection" : 1,
  "publicKeyAlgAndEncoding" : 256,
  "tcDisplay" : 3
} ]
```

4.3. Policy Configuration Examples

nevisFIDO expects that the policy configuration file contains one JSON object, representing a single Policy.

The possible configuration options are not explained in detail here. For more information, refer to the Policy and Match Criteria specifications.

The following sample code shows a policy that accepts all authenticators from the vendor with ID "1234":

Policy configuration example for authenticators from vendor 1234

```
{
  "accepted":
  [
    [{ "vendorID": ["1234"], "authenticationAlgorithms": [1, 2, 5, 6],
"assertionSchemes": ["UAFV1TLV"]}]
  ]
}
```

4.4. Logging Configuration

nevisFIDO uses the Logback framework to log messages. The logback configuration file is located in `/var/opt/nevisfido/<instance>/conf/logback.xml`. nevisFIDO will not start if the logback configuration file does not exist or cannot be read.

You can replace the logback configuration by specifying a custom `--log-config` location in the `RUN_ARGS` environment variable in the instance's `env.conf` file.

For more information regarding the logback configuration, refer to the official Logback framework documentation.

The following example shows the logback configuration that comes with the nevisFIDO package:

Logback configuration example

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false" scan="true" scanPeriod="30 seconds">

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${pattern:-%d{ISO8601} %-4relative [%thread] %-5level
%logger{35} %msg%n}</pattern>
        </encoder>
    </appender>

    <appender name="FILE" class=
"ch.qos.logback.core.rolling.RollingFileAppender">
        <file>/var/opt/nevisfido/default/log/nevisfido.log</file>
        <append>true</append>

        <rollingPolicy class=
"ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>/var/opt/nevisfido/default/log/nevisfido-
%i.log</fileNamePattern>
            <minIndex>1</minIndex>
            <maxIndex>10</maxIndex>
        </rollingPolicy>
        <triggeringPolicy class=
"ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
            <maxFileSize>100MB</maxFileSize>
        </triggeringPolicy>
        <encoder>
            <pattern>${pattern:-%d{ISO8601} %-4relative [%thread] %-5level
%logger{35} %msg%n}</pattern>
        </encoder>
    </appender>

    <logger name="org.springframework.web.filter.CommonsRequestLoggingFilter"
level="DEBUG" additivity="false">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </logger>
    <logger name="ch.nevis.auth.fido.uaf" level="DEBUG" additivity="false">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </logger>
    <logger name="jcan.Op" level="INFO" additivity="false">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </logger>
    <root level="INFO">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

5. Management

The management feature described in this section are experimental and can change in future releases.

nevisFIDO provides two management endpoints that you can use to detect the current state of the instance. These can for example be used to integrate with Kubernetes container probes. [1]

5.1. Liveness Endpoint

The liveness endpoint informs whether the server is up and running. Note that the server can be up and running but not ready to properly handle requests (for instance because of a misconfiguration). The endpoint is accessible via the management port. You configure this port with the `management.server.port` property as described in the server configuration properties. Currently only HTTP is supported.

The following table describes the HTTP API of the liveness endpoint:

Default URL	http://hostname:9089/nevisfido/liveness
Request HTTP method	GET
Response content-type	application/vnd.spring-boot.actuator.v2+json
Response body if nevisFIDO is alive	{ "status" : "UP" }
HTTP status code if nevisFIDO is alive	200 (OK)

5.2. Readiness Endpoint

The readiness endpoint informs whether the server is ready to handle requests. The endpoint is accessible via the management port. You configure this port with the `management.server.port` property as described in the server configuration properties. Currently only HTTP is supported.

The following table describes the HTTP API of the readiness endpoint:

Default URL	http://hostname:9089/nevisfido/health
Request HTTP method	GET
Response content-type	application/vnd.spring-boot.actuator.v2+json
Response body if nevisFIDO is ready	{ "status" : "UP" }
HTTP status code if nevisFIDO is ready	200 (OK)
Response body if nevisFIDO is not ready	{ "status" : "DOWN" }
HTTP status code if nevisFIDO is not ready	503 (Service Unavailable)

6. nevisAuth AuthStates

nevisFIDO provides its own AuthStates, which you can configure and use in nevisAuth.

6.1. Installation

The nevisFIDO AuthStates are installed using the client RPM. After installing the RPM, the AuthStates and the required dependencies are located under `/opt/nevisfidocl/nevisauth/lib`.

6.2. Configuration

When you configure a nevisFIDO AuthState, pay attention to the following:

- If the `classPath` attribute is specified in the `AuthEngine` configuration, it **must** contain the directory with the nevisFIDO AuthStates directory (as in the following sample code):

```
<AuthEngine name="AuthEngine"
    classPath=
"/var/opt/nevisauth/default/plugin:/opt/nevisauth/plugin:/opt/nevisfidocl/nevis
auth/lib"
    classLoadStrategy="PARENT_FIRST"
    useLiteralDictionary="true"
    addAutheLevelToSecRoles="true"
    compatLevel="none"
    inputLanguageCookie="LANG"
>
```

- When you create a FidoUafAuthState, specify the following attributes as follows:
  - `fidoUafServerUrl` : The base URL of the nevisFIDO (server) instance.
  - `fidoUafUsername` : The username of the user in the nevisFIDO (server) instance.

  See also the next sample code:

```
<AuthState name="FidoUafAuthState"
           class="ch.nevis.auth.fido.uaf.authstate.FidoUafAuthState"
           final="false"
           resumeState="false">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="error" next="AuthError"/>
    <ResultCond name="failed" next="AuthError"/>
    <property name="fidoUafUsername" value="${sess:username}" />
    <property name="fidoUafServerUrl" value="https://siven.ch:8443/nevisfido"
/>
    <property name="trustStoreRef" value="DefaultKeyStore" />
</AuthState>
```

- If you employ a JSON based client, use a *DirectResponseState* AuthState as the "AuthDone" AuthState in order to make integration easier. For example, the "AuthDone" AuthState in the sample below will return a successful HTTP response with `{ "message" : "successful authentication" }` as body:

```
<AuthState name="AuthDone"
        class="ch.nevis.esauth.auth.states.directResponse.DirectResponseState"
        final="true"
        resumeState="false">
    <Response value="AUTH_DONE"/>
        <property name="contentType" value="application/json"/>
        <property name="content" value="{&amp;quot;message&amp;quot;:&amp;quot;successful
authentication&amp;quot;}"/>
        <property name="statusCode" value="200" />
</AuthState>
```

- To use a *FidoUafAuthState* AuthState, configure the entry point of the related domain accordingly:

```
<Domain name="FIDO_UAF_AUTHENTICATION" default="true"
    reauthInterval="0"
    inactiveInterval="1800">
    <Entry method="authenticate" state="FidoUafAuthState"/>
</Domain>
```

- Restart the nevisAuth instance.

6.2.1. General Considerations

The FIDO UAF *AuthStates* have been designed to interact with clients that are capable of handling JSON. They can be configured to take information from HTTP requests containing JSON as payload; the configuration allows to define the attribute names of the JSON payload (see details below).

If the *AuthStates* are configured to take JSON as input, it is **required** that the `Content-Type` header of the incoming HTTP request is set to `application/json;charset=UTF-8`.

The protocol between the HTTP client and nevisAuth is not exactly the same as the one interacting directly with nevisFIDO (this protocol is specified in the HTTP API). For example when using the FIDO UAF *AuthStates*, nevisAuth is the one generating the `GetUAFRequest` that triggers the authentication process and sends it to nevisFIDO. Depending on the use case and on how nevisAuth is configured, the information required to generate this `GetUAFRequest` (the `username` and the `transactions`) may come from the HTTP client or not.

6.2.2. FidoUafAuthState

The *FidoUafAuthState* AuthState manages the UAF authentication in a nevisFIDO server.

This section refers to protocol messages defined in the FIDO UAF HTTP Transport Specification.

The UAF authentication process including nevisAuth and the *FidoUafAuthState* is as follows:

1. The `FidoUafAuthState` sends a `GetUAFRequest` to the nevisFIDO server via the `fidoUafUsername` attribute. The value of this attribute (that is, the username) can be retrieved in different ways:
   - By authenticating the user through an AuthState that is configured before the `FidoUafAuthState`. The username value can be a variable expression using `notes`.
   - The HTTP client provides the FIDO UAF username directly in the incoming request as JSON or FORM parameters. The username value can be a variable expression using `inargs`.

   In the context of transaction confirmation, the `FidoUafAuthState` can be configured with a list of FIDO UAF `transaction`s sent to the nevisFIDO server as part of the `GetUAFRequest`. See the FIDO UAF Protocol Specification for more information.

2. The nevisFIDO server returns a `ReturnUafRequest` to nevisAuth, which forwards this response to the client. If the `GetUAFRequest` includes `transaction`s, the `ReturnUafRequest` also contains these `transaction`s (see Authentication Request Service for more information).
3. The client processes the `ReturnUafRequest` and authenticates the user on the FIDO authenticator. The `ReturnUafRequest` contains an `AuthenticationRequest` with an `OperationHeader` including an `Extension`. This `Extension` provides a FIDO UAF session ID, like the one in the following code snippet:

   ```
   { "id" : "ch.nevis.auth.fido.uaf.sessionId",
     "value" : "asdetwdIDsdfsewSAdsds09823423sdfsd9ds" }
   ```

4. The client sends a `SendUafResponse` directly to the nevisFIDO server.
5. The client queries nevisAuth. It depends on the configuration of the `fidoUafSessionId` property in the `FidoUafAuthState` whether the client must provide this property. By default, the client is required to send the property in a POST with a payload like this:

```
{ "fidoUafSessionId" : "asdetwdIDsdfsewSAdsds09823423sdfsd9ds" }
```

6. The `FidoUafAuthState` processes the authentication status query, by checking in the nevisFIDO server whether the user was actually authenticated. If so, the `FidoUafAuthState` will have the transition `ok`. nevisAuth will send a response to the client including a payload like this:

```
{ "status" : "succeeded" }
```

7. The client must send another request to nevisAuth in order to continue with the authentication process.

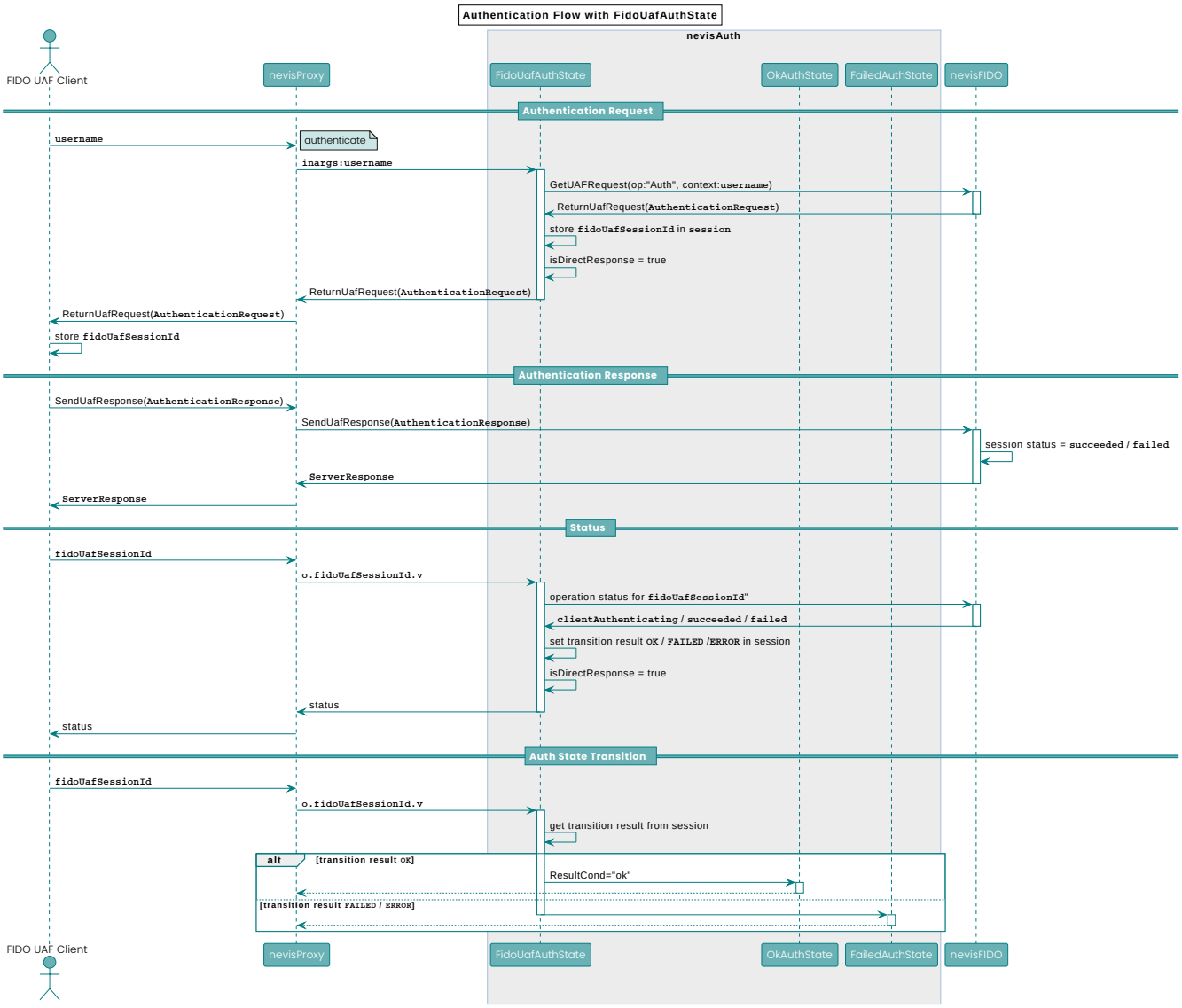


Figure 1. FidoUafAuthState example flow

To allow chained UAF AuthStates, nevisAuth will remove the stored FIDO UAF session ID every time an AuthState transition occurs in the `FidoUafAuthState`. As a consequence, the HTTP client will receive the `authenticated` status only once for a given session ID. The next queries with the same session ID will return an `unknown` status. This is to prevent the following undesired situation: Suppose a given configuration consists of the two chained UAF AuthStates AuthState1 and AuthState2. If the user authenticates in AuthState1, the AuthEngine will make AuthState2 the current AuthState. If the client sends an authentication status request with the session ID used in AuthState1, and the session has not been cleaned up, then the user will be considered authenticated in AuthState2.

6.2.2.1. Restarting the FIDO UAF Authentication

To be able to retrieve the authentication status, nevisAuth requires the FIDO UAF session ID. If the `fidoUafSessionId`

property is not provided in the nevisAuth configuration, it must be provided as part of the request (using the `fidoUafSessionId` JSON attribute).

In any case where a FIDO UAF session ID cannot be associated with the client, the client will initiate an authentication flow, even though an authentication was already triggered. If you configure `fidoUafSessionId` in nevisAuth, the client may lose this ability to control redundant authentications.

The main reasons for requiring the session ID are:

Security

Requiring the session ID adds a security layer to what already is provided by nevisAuth and nevisProxy. Stealing the cookie is not enough to hijack the authentication session. An attacker must also have access to the session ID returned in a previous request by nevisAuth. That is, by stealing the cookie, the attacker can prevent the user from being authenticated by restarting the FIDO UAF authentication (see the point below). However, without the session ID the attacker will not be able to authenticate.

Convenience

Requiring the session ID allows a client application to restart the FIDO UAF authentication at any point in time. This can help to improve the user experience.

By default, sending a request without a FIDO UAF session ID is enough to restart the FIDO UAF authentication. This can be useful in different scenarios, for instance:

- The application requiring authentication is executed in the browser in a laptop whereas the FIDO UAF authentication is done in a mobile device (out-of-band scenario). Suppose the browser polls nevisAuth for the authentication status, but the authentication is still going on after a certain time. Then the application can ask the user to trigger a new FIDO UAF authentication by confirming a message. Note that the browser application could also decide to restart the *whole* authentication, by removing the nevisAuth cookie. However, this could be annoying for the end user.
- The FIDO UAF client is the one interacting with nevisAuth. The FIDO UAF client detects an error in the interaction with the user. Instead of restarting the entire authentication, the FIDO UAF client can decide to restart *only* the FIDO UAF authentication.
- An example using this behaviour is described in Authentication Retry / Fallback Example.

6.2.2.2. FidoUafAuthState Properties

Topic	Description
Class	`ch.nevis.auth.fido.uaf.authstate.FidoUafAuthState`
Logging	`FidoUaf`
Auditing	none
Marker	none

Topic	Description

Properties (generic)	**fidoUafUsername** (string, optional)
	Username of the user in the nevisFIDO server.
	Examples:
	`${inargs:username}`
	`${inargs:o.username.v}`: Provides the username in the request parameter `username`.
	`{ "username" : "jsmith" }`: Provides the username with a JSON object.
	
	fidoUafServerUrl (string, required)
	Base URL of the nevisFIDO server. You **cannot** use variable expressions to specify this value.
	Example: `https://siven.ch:8443/nevisfido`. With this configuration the FIDO UAF AuthState will send the `GetUAFRequest` to `https://siven.ch:8443/nevisfido/uaf/1.1/request/authentication`.
	
	fidoUafSessionId (string, optional)
	The session ID required to query nevisFIDO about the status of an ongoing authentication.
	Default value: `${inargs:o.fidoUafSessionId.v}`.
	With the default value, the client is expected to maintain and send the FIDO UAF session ID to nevisAuth using a POST with the following payload:
	`{ "fidoUafSessionId" : "the_session_ID" }`
	
	fidoUafTransactions (string, optional)
	The transactions that can be provided to nevisFIDO in the `GetUAFRequest`. The string represents a JSON array with the transactions as defined in the FIDO UAF Protocol Specification.
	Example:
	`{ "contentType": "text/plain", "content": "VGhlIHRyYW5zYWN0aW9uIGNvbmZpcm1hdGlvbiB0ZXh0" }`
	
	trustStoreRef (string, optional)
	The keystore reference that must be used as truststore to connect to nevisFIDO. The truststore must contain the public key of the nevisFIDO server certificate. The referenced keystore must be defined inside a `KeyStore` element of the *SessionCoordinator* section of the nevisAuth configuration.
	To skip certificate validation (including hostname validation), use the keyword `all`.
	You **cannot** use variable expressions to specify this value.
	Default Value: `DefaultKeyStore`
	
	keyStoreRef (string, optional)
	The keystore reference that must be used as keystore to connect to nevisFIDO using client authentication (2-way TLS). The keystore must contain the private key that will be used as credentials of nevisAuth when connecting to nevisFIDO. The referenced keystore must be defined inside a `KeyStore` element of the *SessionCoordinator* section of the nevisAuth configuration.
	You **cannot** use variable expressions to specify this value.
	Default Value: `DefaultKeyStore`
	
	keyObjectRef (string, optional)
	Reference to the key object in the `KeyStore`, referenced by `keyStoreRef`. This specifies the private key that will be used as credentials of nevisAuth when connecting to nevisFIDO using client authentication. The referenced key object reference must be defined inside a `KeyObject` element of the `KeyStore` in the nevisAuth configuration referenced by `keyStoreRef`.
	You **cannot** use variable expressions to specify this value.
	Currently only PKCS12 and JKS keystores are supported.

Methods	process

Topic	Description
Input	*FIDO UAF session ID* The `FidoUafAuthState` queries the nevisFIDO server to check whether the session with the provided ID is authenticated. The FIDO UAF session ID can be provided using a POST with the following payload: `{ "fidoUafSessionId" : "the_session_ID" }` **NOTE**: This is the equivalent of sending the session ID in the `o.fidoUafSessionId.v` attribute of the `inargs`.
Transitions	`error`: If an error occured in the communication with nevisFIDO. `failed`: If nevisFIDO reports a failed authentication (in response to the client's query for the authentication status). `ok`: If nevisAuth detects a successfully authenticated user in nevisFIDO. In this case, nevisFIDO responses positively to the client's query for the authentication status.

Output	`ReturnUafRequest`:
	If the `FidoUafAuthState` generates a `GetUAFRequest`, it will forward the response from the nevisFIDO server, a `ReturnUafRequest` object, to the client. The client must process this `ReturnUafRequest`, authenticate the user and eventually send a `SendUafResponse` with the `AuthenticationResponse` to the nevisFIDO server.
	
	Payload with authentication status:
	If the `FidoUafAuthState` queried nevisFIDO regarding the authentication status of the session, it will return a JSON object describing the status. Example:
	
	`{`
	`"status" : "succeeded",`
	`"timestamp" : "2018-12-14T20:37:48.556Z",`
	`"uafStatusCode" : 1200,`
	`"userId" : "09890989009",`
	`"authenticators" : [ {`
	`"aaid" : "ABBA#0001"`
	`} ]`
	`}`

Errors	
Notes	

Example	The AuthState in the following example integrates FIDO UAF authentication in nevisAuth using the nevisFIDO server located in `siven.ch` with port `8443`. The FIDO UAF username is retrieved from the `username` parameter of the `notes`.
	
	`<AuthState name="FidoUafAuthState"`
	`class="ch.nevis.auth.fido.uaf.authstate.FidoUafAuthState" resumeState="false">`
	`<ResultCond name="ok" next="AuthDone"/>`
	`<ResultCond name="error" next="AuthError"/>`
	`<ResultCond name="failed" next="AuthError"/>`
	`<property name="fidoUafUsername" value="${sess:username}" />`
	`<property name="fidoUafServerUrl" value="https://siven.ch:8443/nevisfido" />`
	`<property name="trustStoreRef" value="DefaultKeyStore" />`
	`</AuthState>`

The `FidoUafAuthState` does not display a GUI. This means that defining GUI elements or setting the property `final` to "true" will have no effect.

6.2.2.3. Request and Response Examples

6.2.2.3.1. Request Body Providing Username

It is assumed that the `fidoUafUsername` attribute value is `${inargs:o.username.v}` and the `transactions` attribute is `${inargs:o.transaction.v}`.

the transactions must be provided as an stringified JSON array.

```
{
  "username" : "jeff",
  "transaction" : "[{\"contentType\":\"text/plain\",\"content\":
\"Q29uZmlybSB5b3VyIHB1cmNoYXNlIGZvciBhIHZhbHVlIG9mIENIRjIwMC4\"}]"
}
```

6.2.2.3.2. Response Containing AuthRequest

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:21 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 813

{
  "lifetimeMillis" : 120000,
  "uafRequest" : "[{\"header\":{\"serverData\":\"YxHI-
E0SLDf7uH_GvQB10Nb50oB8f8GBiBFO5KiyK0lZfGmoz_QH_jueHmMdBOSF2XIq5Be-
UcQAEJ4O33XK9Q\",\"upv\":{\"major\":1,\"minor\":1},\"op\":\"Auth\",\"appID\":\"
https://www.siven.ch/appID\",\"exts\":[{\"id\":\"ch.nevis.auth.fido.uaf.session
id\",\"data\":\"a0b84187-28e5-4672-92f7-
f4cbbb598568\",\"fail_if_unknown\":false}]},\"challenge\":\"QRh3CZwG8t7xGKBetB7
PmpX8rFr9AgThKbWR58pSy6nFWh9-
AH1yYokrjZlSURqPGQ0Kd6DU16JUAAu5c1bVRA\",\"policy\":{\"accepted\":[[{\"userVeri
fication\":1023,\"authenticationAlgorithms\":[1,2,3,4,5,6,7,8,9],\"assertionSch
emes\":[\"UAFV1TLV\"]}]]},\"transaction\":[{\"contentType\":\"text/plain\",\"co
ntent\":\"Q29uZmlybSB5b3VyIHB1cmNoYXNlIGZvciBhIHZhbHVlIG9mIENIRjIwMC4\"}]}]",
  "statusCode" : 1200,
  "op" : "Auth"
}
```

6.2.2.3.3. Status Request Body

It is assumed that the `fidoUafSessionId` attribute value is `${inargs:o.fidoUafSessionId.v}`.

```
{
  "fidoUafSessionId" : "1c8a5b00-165c-4a63-ae13-2e03fb7f57ce"
}
```

6.2.2.3.4. Status Response

```
HTTP/1.1 200 OK
Date: Mon, 04 Feb 2019 12:42:29 GMT
Content-Type: application/json;charset=utf-8
Transfer-Encoding: chunked
Content-Length: 415

{
  "status" : "succeeded",
  "timestamp" : "2019-02-04T12:44:34.980Z",
  "uafStatusCode" : 1200,
  "asmStatusCode" : 0,
  "clientErrorCode" : 0,
  "userId" : "123122233",
  "authenticators" : [ {
    "aaid" : "ABBA#0001"
  } ]
}
```

6.2.3. OutOfBandFidoUafAuthState

The *OutOfBandFidoUafAuthState* AuthState manages the UAF authentication in a nevisFIDO server by invoking the Dispatch Token Service. The typical use case is that a client wants to access an application using a desktop, whereas the authentication is done with a mobile device. In this case, a token is dispatched to the mobile device, where the user authenticates.

The UAF authentication process including nevisAuth and the *OutOfBandFidoUafAuthState* is as follows:

1. The `OutOfBandFidoUafAuthState` sends a dispatch target query to retrieve the dispatch targets defined for the user. The `fidoUafUsername` attribute is used to generate this request. The value of this attribute (that is, the username) can be retrieved in different ways:
   - By authenticating the user through an AuthState that is configured before the `OutOfBandFidoUafAuthState`. The username value can be a variable expression using `notes`.
   - The HTTP client provides the FIDO UAF username directly in the incoming request as JSON or FORM parameters. The username value can be a variable expression using `inargs`.
2. The nevisFIDO server returns the list of dispatch targets to nevisAuth, which forwards the dispatch targets to the client. If no dispatch targets are found, nevisAuth returns an empty list.
3. The client selects one of the dispatch targets, if required in interaction with the user. Subsequently, the client sends a request with the dispatch target ID and the username to nevisAuth.
4. The `OutOfBandFidoUafAuthState` sends a dispatch token request to the nevisFIDO server with the provided dispatch target ID. nevisFIDO uses the Dispatch Token Service for this.
   In the context of transaction confirmation, the `OutOfBandFidoUafAuthState` can be configured with a list of FIDO UAF `Transaction`s. This list of `Transaction`s will be sent to the nevisFIDO server as part of the `GetUAFRequest` inside the dispatch token request. See the FIDO UAF Protocol Specification for more information.
5. The nevisFIDO server returns a dispatch token response to nevisAuth, which forwards the response to the client. The dispatch token response contains a FIDO UAF session ID. See Example Response Using FCM Dispatcher for a response example.
6. The client queries nevisAuth. It depends on the configuration of the `fidoUafSessionId` property in the `OutOfBandFidoUafAuthState` whether the client must provide this property. By default, the client is required to send the `fidoUafSessionId` property in a POST with a payload like this:

   ```
   { "fidoUafSessionId" : "asdetwdIDsdfsewSAdsds09823423sdfsd9ds" }
   ```

7. The `OutOfBandFidoUafAuthState` processes the authentication status query, by asking the nevisFIDO server whether the user was actually authenticated. If so, the `OutOfBandFidoUafAuthState` has a transition `ok`.

nevisAuth sends a response to the client, including a payload like this:

```
{
  "status" : "succeeded",
  "timestamp" : "2018-12-14T20:37:48.556Z",
  "tokenInformation" : {
    "tokenResult" : "tokenRedeemed",
    "dispatcherInformation" : {
      "name" : "firebase-cloud-messaging",
      "response" : "successful dispatch"
    }
  },
  "uafStatusCode" : 1200,
  "userId" : "123122233",
  "authenticators" : [ {
    "aaid" : "ABBA#0001"
  } ]
}
```

8. The client must send another request to nevisAuth to continue with the authentication process.

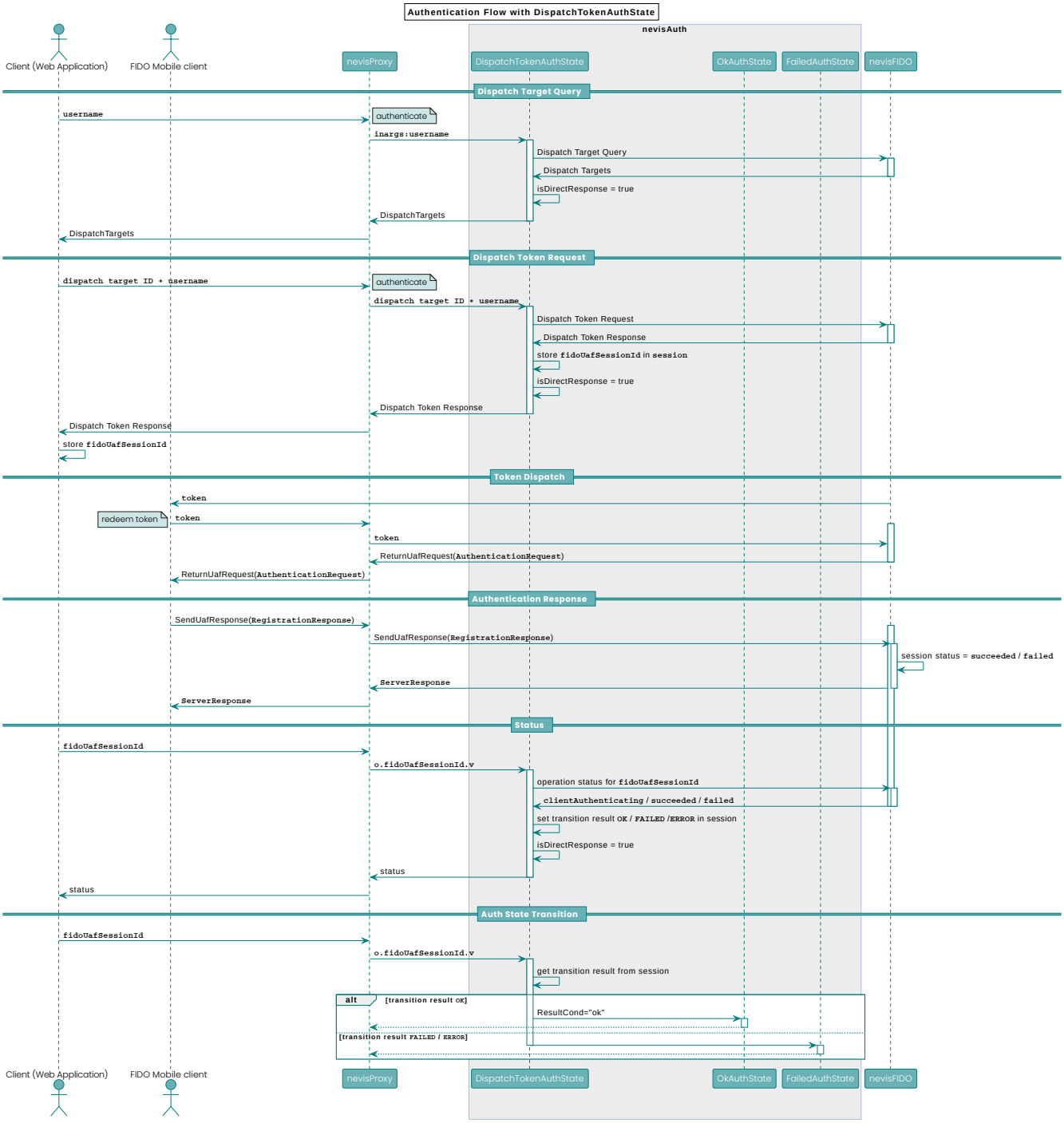


Figure 2. OutOfBandFidoUafAuthState example flow

To allow chained UAF AuthStates, nevisAuth will remove the stored FIDO UAF session ID every time an AuthState transition occurs in the `OutOfBandFidoUafAuthState`. As a consequence, the HTTP client will receive the `authenticated` status only once for a given session ID. The next queries with the same session ID will return an `unknown` status.
This is to prevent the following undesired situation: Suppose a given configuration consists of the two chained UAF AuthStates AuthState1 and AuthState2. If the user authenticates in AuthState1, the AuthEngine will make AuthState2 the current AuthState. If the client sends an authentication status request with the session ID used in AuthState1, and the session has not been cleaned up, then the user will be considered authenticated in AuthState2.

6.2.3.1. Restarting the FIDO UAF Authentication

To be able to retrieve the authentication status, nevisAuth requires the FIDO UAF session ID. If the `fidoUafSessionId` property is not provided in the nevisAuth configuration, it must be provided as part of the request (using the `fidoUafSessionId` JSON attribute).

In any case where a FIDO UAF session ID cannot be associated with the client, the client will initiate an authentication flow, even though an authentication was already triggered. If you configure `fidoUafSessionId` in nevisAuth, the client may lose this ability to control redundant authentications.

The main reasons for requiring the session ID are:

Security

Requiring the session ID adds a security layer to what already is provided by nevisAuth and nevisProxy. Stealing the cookie is not enough to hijack the authentication session. An attacker must also have access to the session ID returned in a previous request by nevisAuth. That is, by stealing the cookie, the attacker can prevent the user from being authenticated by restarting the FIDO UAF authentication (see the point below). However, without the session ID the attacker will not be able to authenticate.

Convenience

Requiring the session ID allows a client application to restart the FIDO UAF authentication at any point in time. This can help to improve the user experience.

By default, sending a request without a FIDO UAF session ID is enough to restart the FIDO UAF authentication (see step 1 in the previous section). This can be useful in different scenarios, for instance:

- The application requiring authentication is executed in the browser in a laptop whereas the FIDO UAF authentication is done in a mobile device (out-of-band scenario). Suppose the browser polls nevisAuth for the authentication status, but the authentication is still going on after a certain time. Then the application can ask the user to trigger a new FIDO UAF authentication by confirming a message. Note that the browser application could also decide to restart the *entire* authentication, by removing the nevisAuth cookie. However, this could be annoying for the end user.
- The FIDO UAF client is the one interacting with nevisAuth. The FIDO UAF client detects an error in the interaction with the user. Instead of restarting the entire authentication, the FIDO UAF client can decide to restart *only* the FIDO UAF authentication.

6.2.3.2. Channel Linking

In the out-of-band case the end-user owns two devices: one device (typically a desktop computer or laptop) that tries to access a protected system or to perform a protected operation, such a bank transaction. Another device (typically a mobile device) which is in charge of performing the authentication (or part of it) required to access the protected resource. The channel linking refers to a mechanism that allows to the end-user to verify that the authentication requested in the mobile device corresponds to the operation triggered by the device trying to access the protected system.

Note: the Dispatch Token Service service can be used to implement custom channel linking mechanism by sending the required data to the user's mobile device. The `dispatchInformation` attribute of this AuthState can be used to provide information to the dispatch token service.

The channel linking can be configured using the `channelLinkingMode` AuthState attribute.

6.2.3.2.1. Visual String Channel Linking

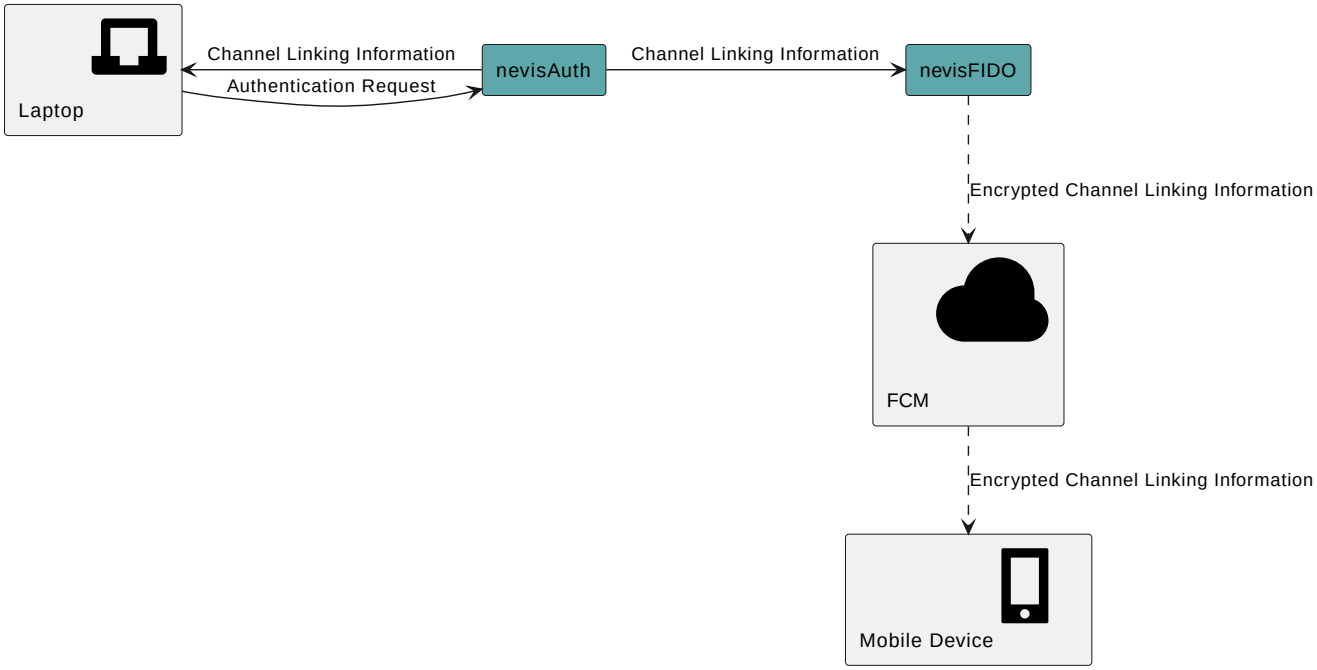
Currently, the only channel linking supported out-of-the box by NEVIS Mobile Authentication is the visual string channel linking. With this channel linking, when the client requiring authentication is redirected to nevisAuth and the `OutOfBandFidoUafAuthState` is invoked, the AuthState will generate a random string of two hexadecimal characters. This String will be sent encrypted to the user's mobile device (the one performing FIDO UAF authentication) and the `OutOfBandFidoUafAuthState` will also include the generated random String in the HTTP response that is being to the user's laptop.

When the mobile device receives message (typically with a push notification when using the FCM (Firebase Cloud Messaging) Dispatcher, it will decrypt it and present the generated random String to the user. The random String will also be presented in the device (laptop) trying to access the protected resource. The user will be asked to verify that both strings match before proceeding with the authentication.

To enable the Visual String Channel Linking, set `visualString` as the value for the `channelLinkingMode`

configuration attribute.

The following diagram shows the process when using the FCM dispatcher:



When using the QR Code Dispatcher, there is no added value to use the visual String linking, because the QR code scan is implicitly linking the device triggering the operation (laptop) with the authenticating device (the mobile).

The same thing applies to the Link Dispatcher, which is used when both the device requiring authentication and the device containing the authenticating application are the same.

6.2.3.2.2. Channel Linking Payload

The format of the channel linking information sent to both user devices (typically laptop and mobile device) is described in the table below:

Table 3. Channel Linking Payload Dictionary

Attribute	Type	Description	Optional
`channelLinking.mode`	`String`	The channel linking verification mode. Currently only `visualString` is supported.	false
`channelLinking.verificationContent`	`String`	The data used to do the channel linking.	false

Example:

```
{
  "channelLinking" : {
    "mode" : "visualString",
    "content" : "BC"
  }
}
```

See the Encrypted Push Message for an example on how the channel linking information is transmitted inside the push notification message to the end-users mobile device.

See the Response with the Dispatch Response for an example on how the channel linking information is sent back to the end-users laptop in the HTTP response.

6.2.3.3. Security Considerations

The `OutOfBandFidoUafAuthState` is able to query dispatch targets for a non-authenticated user. This means that dispatch target information will be exposed to non-authenticated users. This is possible even if you configured the dispatch target query such that it requires SecToken authorization.

If you consider exposing information to non-authenticated users as a concern, you must not use the `OutOfBandFidoUafAuthState` as a first factor authentication. That is, the `OutOfBandFidoUafAuthState` should not be the first AuthState in the authentication flow.
Instead, you could use the `OutOfBandFidoUafAuthState` as the second factor authentication. In this case, the username is provided by the first factor authentication, typically a username/password-based AuthState.

The `OutOfBandFidoUafAuthState` AuthState requires the use of a dispatch target, even though some nevisFIDO dispatchers do not require it. The reason behind is security: when a dispatch target is used, the contents that are dispatched are encrypted, which avoids a number of attacks (for instance QRLJacking attacks using the QR Code Dispatcher).

6.2.3.4. OutOfBandFidoUafAuthState Properties

Topic	Description
Class	`ch.nevis.auth.fido.uaf.authstate.OutOfBandFidoUafAuthState`
Logging	`FidoUaf`
Auditing	none
Marker	none

Topic	Description

Properties (generic)	**fidoUafUsername** (string, required)
	Username of the user in the nevisFIDO server.
	Examples:
	`${inargs:username}`
	`${inargs:o.username.v}` : Provides the username in the request parameter `username`.
	`{ "username" : "jsmith" }` : Provides the username in a JSON object.
	NOTE: The `fidoUafUsername` property value might be required several times when contacting the nevisFIDO server (see the `Input` and `Output` rows below for details). If the username is provided by a previous AuthState and not in the HTTP request, do not use `notes` to store and pass the `fidoUafUsername` property value. This is because the lifetime of `notes` is limited to the current request. In this case, use another mechanism, such as the session with the `sess` keyword.
	
	fidoUafServerUrl (string, required)
	Base URL of the nevisFIDO server. You **cannot** use variable expressions to specify this value.
	Example: `https://siven.ch:8443/nevisfido`. With this configuration, the FIDO UAF AuthState will send a dispatch target request to `https://siven.ch:8443/nevisfido/token/dispatch/authentication`.
	
	dispatchTargetId (string, optional)
	The identifier of the dispatch target to which nevisFIDO must send the token.
	Default value: `${inargs:o.dispatchTargetId.v}`. With this value, the client must send the dispatch target ID using a POST with the following payload:
	`{ "dispatchTargetId" : "the_dispatch_target_ID" }`
	
	dispatcher (string, optional)
	The identifier of the dispatcher that must be used by nevisFIDO to send the token. If no dispatcher is provided, then the dispatcher associated with the dispatch target specified with the attribute `dispatchTargetId` will be used. The supported dispatchers are: `firebase-cloud-messaging`, `png-qr-code` and `link`.
	Default value: `${inargs:o.dispatcher.v}`. With this value, the client can send the dispatcher using a POST with the following payload:
	`{ "dispatcher" : "png-qr-code" }`
	If you want to systematically use the dispatcher associated with the dispatch target ID, you can provide an empty String as value:
	`<property name="dispatcher" value=""/>`
	
	dispatchInformation (string, optional)
	The dispatch information that will be send to the nevisFIDO dispatcher. See Dispatch Token Request Format for details. *Default value*: `${inargs:o.dispatchInformation.v}`. With this value, the client must send the dispatch information using a POST with the following payload:
	`{ "dispatchInformation" : { "some": "information for the dispatcher" }`
	
	channelLinkingMode (string, optional)
	The channel linking verification mode. Currently only two modes are supported: `none`: no channel verification information will be generated. `visualString`: a random two digit hexadecimal String will be used to perform visual channel linking. See Channel Linking for details. *Default Value*: `none`
	
	this is a convenience attribute to generate the channel linking information, send it along the contents of the `dispatchInformation` attribute and return the generated information in the response. The information of the `dispatchInformation` attribute will take precedence over the contents of this attribute.
	
	fidoUafSessionId (string, optional)
	The session ID required to query nevisFIDO for the status of an ongoing authentication.
	Default value: `${inargs:o.fidoUafSessionId.v}`.
	When this parameter is not set, the client must maintain and send the FIDO UAF session ID to nevisAuth using a POST with the following payload:
	`{ "fidoUafSessionId" : "the_session_ID" }`

Topic	Description
Methods	process

Input	*Username*:
	If only the username is provided as input, the `OutOfBandFidoUafAuthState` will send a dispatch target query to nevisFIDO to retrieve the available dispatch targets for this user.
	
	Username and dispatch target ID:
	If the dispatch target ID and the username are included in the input (and no FIDO UAF session ID is provided), the `OutOfBandFidoUafAuthState` will send a dispatch token request to the nevisFIDO server to trigger an authentication. The dispatch target ID and username can be provided using a POST with the following payload:
	`{ "dispatchTargetId" : "the_dispatch_target_ID", "username" : "the_user_name" }`
	
	FIDO UAF session ID:
	If the FIDO UAF session ID is provided as input, the `OutOfBandFidoUafAuthState` will query the nevisFIDO server to check whether the session with the provided ID is authenticated. The FIDO UAF session ID can be provided using a POST with the following payload:
	`{ "fidoUafSessionId" : "the_session_ID" }`
	NOTE: This is the equivalent of sending the session ID in the `o.fidoUafSessionId.v` attribute of the `inargs`.

Transitions	`error`: If an error occured in the communication with nevisFIDO. `failed`: If nevisFIDO reports a failed authentication (in response to the client's query for the authentication status). `dispatchFailed`: If nevisFIDO reports that the token dispatch failed. `ok`: If nevisAuth detects a successfully authenticated user in nevisFIDO. In this case, nevisFIDO responses positively to the client's query for the authentication status.
Output	*Dispatch targets*: If the `OutOfBandFidoUafAuthState` generated a query dispatch target request, the output is a list of dispatch target objects (coming from the nevisFIDO server in response to the request). The client must identify the correct dispatch target in this list, that is, the target to dispatch the token to. See an example here. *Dispatch token response*: If the `OutOfBandFidoUafAuthState` generated a dispatch token request, the output is the dispatch token response coming from the nevisFIDO server. If channel linking has been configured, then the channel linking information will also be included. The client must retrieve the session ID from the dispatch token response. The session ID is needed to poll the authentication status. See an example here. *Authentication status*: If the `OutOfBandFidoUafAuthState` queried nevisFIDO for the authentication status of the session, the output is a JSON object describing the status. See an example here.
Errors	
Notes	

Topic	Description

Example	The AuthState in the following example integrates FIDO UAF authentication in nevisAuth using the nevisFIDO server located in `siven.ch` with port `8443`. The FIDO UAF username is retrieved from the `username` parameter of the `notes`.
	
	`<AuthState name="OutOfBandFidoUafAuthState"`
	`class="ch.nevis.auth.fido.uaf.authstate.OutOfBandFidoUafAuthState"`
	`resumeState="false">`
	`<ResultCond name="ok" next="AuthDone"/>`
	`<ResultCond name="error" next="AuthError"/>`
	`<ResultCond name="failed" next="AuthError"/>`
	`<property name="fidoUafUsername" value="${sess:username}" />`
	`<property name="fidoUafServerUrl" value="https://siven.ch:8443/nevisfido" />`
	`<property name="trustStoreRef" value="DefaultKeyStore" />`
	`</AuthState>`

The `OutOfBandFidoUafAuthState` does not display a GUI. This means that defining GUI elements or setting the property `final` to "true" will have no effect.

6.2.3.5. Request and Response Examples

6.2.3.5.1. Request Body Providing Username

It is assumed that the `fidoUafUsername` attribute value is `${inargs:o.username.v}`.

```
{
  "username" : "jeff"
}
```

6.2.3.5.2. Response Containing Dispatch Targets for User

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:24 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 117

{
  "dispatchTargets" : [ {
    "id" : "42950470-0209-47f6-85de-ae7ab2826bfd",
    "name" : "My Mobile Phone"
  } ]
}
```

6.2.3.5.3. Request Body Providing dispatchTargetId

It is assumed that the `fidoUafUsername` attribute value is `${inargs:o.username.v}` and the `dispatchTargetId` attribute value is `${inargs:o.dispatchTargetId.v}`.

```
{
  "username" : "jeff",
  "dispatchTargetId" : "054a0272-0822-4dac-9c7b-42a27174d287"
}
```

6.2.3.5.4. Response with the Dispatch Response

```
{
  "token" : "e65e7b0f-947e-41bc-a26a-042314d08583",
  "sessionId" : "2708df2f-5b95-49a2-8aa9-0cb5248a7153",
  "dispatcherInformation" : {
    "name" : "firebase-cloud-messaging",
    "response" : 0
  },
  "dispatchResult" : "dispatched",
  "channelLinking" : {
    "mode" : "visualString",
    "content" : "AB"
  }
}
```

6.2.3.5.5. Status Request Body

It is assumed that the `fidoUafSessionId` attribute value is `${inargs:o.fidoUafSessionId.v}`.

```
{
  "fidoUafSessionId" : "1c8a5b00-165c-4a63-ae13-2e03fb7f57ce"
}
```

6.2.3.5.6. Status Response Example

```
{
  "status" : "succeeded",
  "timestamp" : "2022-08-03T13:13:14.691Z",
  "tokenInformation" : {
    "tokenResult" : "tokenRedeemed",
    "dispatcherInformation" : {
      "name" : "firebase-cloud-messaging",
      "response" : "successful dispatch"
    }
  },
  "uafStatusCode" : 1200,
  "asmStatusCode" : 0,
  "clientErrorCode" : 0,
  "userId" : "userId",
  "authenticators" : [ {
    "aaid" : "ABBA#0001"
  } ]
}
```

6.2.3.6. Authentication Retry / Fallback Example

During the out-of-band authentication, there are a number of issues that can occur before the user can proceed with the authentication in the mobile device. For instance, if the FCM (Firebase Cloud Messaging) Dispatcher is being used, some of these potential problems are: the message could not be dispatched because there is a temporary network error between nevisFIDO and the Firebase infrastructure, the message could be displayed but did not reach the authenticating mobile device, the user ignored inadvertently the push message that reached the mobile device, etc.

When one of these situations occur, a new authentication can be triggered as described in Restarting the FIDO UAF Authentication. For example, if the first dispatcher that was used was the FCM dispatcher, the FCM dispatcher can be used again, or it can be chosen to use another dispatcher, like the QR Code Dispatcher as an alternative/fallback; the QR code dispatcher might be considered less user-friendly because it requires more interaction from the user (the user must point the mobile device to do the QR code scan), but it does not have some of the pitfalls of the FCM dispatcher, like the network connectivity issues mentioned above.

In this chapter, the client (typically in a laptop or desktop) that interacts with NEVIS (and in particular with nevisAuth), will be referred as HTTP client.

6.2.3.6.1. Identifying if an Authentication Restart is Needed

It is the responsibility of the HTTP client communicating with nevisAuth to decide whether another FIDO UAF authentication should be triggered (using the initial dispatcher or a new one). The criteria to be followed (time to wait before triggering the new authentication, whether the new authentication triggering requires the user explicit consent or not, etc.) depends on the desired user interaction. This section just hints which attributes in the responses returned by the `OutOfBandFidoUafAuthState` can be used to determine to start a new authentication or not.

There are two main reasons to initiate a new FIDO UAF authentication:

- The dispatch failed.
  This situation can be identified by the HTTP client by checking the value of the dispatchResult attribute of the dispatch response: if the value is not `dispatched`, then dispatch failed.

To be able to retry out-of-band FIDO UAF authentication after a dispatch failure, the transition `dispatchFailed` *must either not be set to continue to another* `AuthState` *(default behaviour), or the* `dispatchFailed` *transition must be set to continue to another* `OutOfBandFidoUafAuthState`.

- The dispatch worked (i.e. nevisFIDO could send the message to the Firebase infrastructure), but the user did not receive the push message or did not manage to proceed with the authentication (for instance, because the push message notification was discarded). After a certain time, the HTTP client could decide (after asking directly the end-user or not) to trigger a new FIDO UAF authentication.
  The situation can be identified by the HTTP client by checking that the value of the status attribute of the status response is `tokenCreated`.

If the message in the initial authentication could be dispatched, the new FIDO UAF authentication must be triggered **before** *the initial authentication fails. By failure it is meant either reaching the authentication time-out in nevisFIDO, or the user providing bad credentials in the device. Once the FIDO UAF authentication fails, the* `OutOfBandFidoUafAuthstate` *will go to the next AuthState configured for the transition* `error`.

6.2.3.6.2. Restarting Authentication

To restart the authentication, all the HTTP client must do is to send a new POST request without the session ID on it. In the example below the following `OutOfBandFidoUafAuthState` configuration is assumed. In the example the `username` is retrieved from the session (set by a previous AuthState, this can be used in a scenario with FIDO UAF as a second factor authentication). The `dispatchTargetId` and `dispatcher` values allow the HTTP client to provide them using a JSON payload:

```
<AuthState name="OutOfBandFidoUafAuthState" class=
"ch.nevis.auth.fido.uaf.authstate.OutOfBandFidoUafAuthState" final="false">
   <ResultCond name="ok" next="AuthDone" />
   <ResultCond name="error" next="AuthError" />
   <ResultCond name="failed" next="AuthError" />
   <property name="fidoUafUsername" value="${sess:username}" />
   <property name="fidoUafServerUrl" value="https://siven.ch:9443/nevisfido" />
   <property name="trustStoreRef" value="TrustStoreForNevisFido" />
   <property name="keyStoreRef" value="KeyStoreForNevisFido" />
   <property name="keyObjectRef" value="ClientKeyForNevisFido" />
   <property name="dispatchTargetId" value="${inargs:o.dispatchTargetId.v}" />
   <property name="dispatcher" value="${inargs:o.dispatcher.v}" />
</AuthState>
```

The HTTP client can decide to start a new FIDO UAF authentication, but instead of using FCM as dispatcher, it prefers to receive a QR code in PNG format to transfer the required information to the mobile device (i.e. use the QR Code Dispatcher). This can be done by sending a POST HTTP request with the following payload:

```
{
  "dispatcher" : "png-qr-code",
  "dispatchTargetId" : "054a0272-0822-4dac-9c7b-42a27174d287"
}
```

The response will look like the following:

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "c9fa045e-2d2b-47d8-83de-6af50d938e7f",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "png-qr-code",
    "response" : "<QR Code as PNG encoded using base64 URL>"
  }
}
```

The client can then simply read `dispatcherInformation.response` attribute, which contains the QR code in PNG format, and render the QR code, so that the end-user can scan it in the mobile device associated with dispatch target `054a0272-0822-4dac-9c7b-42a27174d287` (only the mobile device containing the dispatch target can decrypt the contents of the QR code).

Once the QR code scanned, FIDO UAF authentication will proceed in the mobile device.

7. nevisProxy Configuration

This section guides you through the steps necessary to set up nevisProxy. Because these steps are different for each operation, the configuration is described on a per-operation basis and illustrated by code snippets.

You configure the registration and deregistration endpoints in the same way. To avoid duplication and for the sake of clarity, this chapter does not include the snippets for deregistration. Please use the registration configuration snippets to configure the deregistration endpoints (and adapt the endpoints).

7.1. Configuration Snippets For Registration

It is recommended only allowing authenticated users to perform registration. The following configuration snippet shows how to protect the nevisFIDO registration endpoint.

This is how it works. First the client application tries to perform a registration through the URL `https://<hostname>/fidouaf/registration/`. nevisProxy redirects the client to the `FIDO_UAF_REGISTRATION` realm/domain in nevisAuth, in order for the client to authenticate, for instance by providing an user ID and password. Redirecting and authenticating the client is achieved by defining an `IdentityCreationFilter`.
Once the client is successfully authenticated, nevisAuth generates a `SecToken` with information regarding the authenticated user. nevisAuth the sends the SecToken to nevisProxy, which must forward it to nevisFIDO (see Authorization for details). In order for nevisProxy to send the `SecToken` to nevisFIDO, you must configure a `DelegationFilter`.

```
    <!-- Authentication filter for FIDO UAF Registration -->
    <filter>
        <filter-name>FidoUafRegistrationFilter</filter-name>
        <filter-class>
ch::nevis::isiweb4::filter::auth::IdentityCreationFilter</filter-class>
        <init-param>
            <param-name>AuthenticationServlet</param-name>
            <param-value>NevisAuthConnector</param-value>
            <description>The configured name of the authentication
servlet</description>
        </init-param>
        <init-param>
            <param-name>LoginRendererServlet</param-name>
            <param-value>BuiltinLoginRenderer</param-value>
            <description>The configured name of the login renderer
servlet</description>
        </init-param>
        <init-param>
            <param-name>Realm</param-name>
            <param-value>FIDO_UAF_REGISTRATION</param-value>
            <description>The realm of the registration</description>
        </init-param>
        <init-param>
            <param-name>InactiveInterval</param-name>
            <param-value>7200</param-value>
            <description>
                The maximum interval between two request associated to the same
session
                (if deleted or 0, value is taken from nevisAuth 'Domain'
element)
            </description>
        </init-param>
        <init-param>
            <param-name>EntryPointID</param-name>
            <param-value>localhost</param-value>
            <description>The entry point id (will be part of the
sectoken)</description>
        </init-param>
        <init-param>
            <param-name>StoreInterceptedRequest</param-name>
            <param-value>false</param-value>
        </init-param>
```

```
        <init-param>
            <param-name>InterceptionRedirect</param-name>
            <param-value>never</param-value>
        </init-param>
    </filter>

    <!-- Delegation filter for nevisFIDO.  The role of this filter is to
transmit the SecToken to nevisFIDO. -->
    <filter>
        <filter-name>FidoUafRegistrationDelegationFilter</filter-name>
        <filter-
class>::ch::nevis::isiweb4::filter::delegation::DelegationFilter</filter-class>
        <init-param>
            <param-name>DelegateBasicAuth</param-name>
            <!-- The value of the first element here is not relevant.  However
the SecToken must
                 be included as the second element.  nevisFIDO will ignore the
value of
                 the first parameter and will extract the username from the
SecToken.  -->
            <param-value>
                AUTH:user.auth.UserId
                AUTH:user.auth.SecToken
            </param-value>
        </init-param>
    </filter>

    <!-- URL Mapping with the endpoint to be protected. -->
    <filter-mapping>
        <filter-name>FidoUafRegistrationFilter</filter-name>
        <url-pattern>/fidouaf/registration/*</url-pattern>
    </filter-mapping>

    <!-- Mapping to use the delegation filter with the connector for
registration. -->
    <filter-mapping>
        <filter-name>FidoUafRegistrationDelegationFilter</filter-name>
        <servlet-name>FidoUafRegistrationConnector</servlet-name>
    </filter-mapping>

    <!-- nevisFIDO registration endpoint.  It is protected with an
IdentityFilter. It describes the
         nevisFIDO server host, port and URL. -->
    <servlet>
        <servlet-name>FidoUafRegistrationConnector</servlet-name>
        <servlet-
class>ch::nevis::isiweb4::servlet::connector::http::HttpsConnectorServlet</serv
let-class>
        <init-param>
            <param-name>InetAddress</param-name>
            <param-value>localhost:9443</param-value>
        </init-param>
        <init-param>
            <param-name>MappingType</param-name>
            <param-value>pathinfo</param-value>
        </init-param>
```

```
        <init-param>
            <param-name>URIPrefix</param-name>
            <param-value>/nevisfido/uaf/1.1/request/registration</param-value>
        </init-param>
        <init-param>
            <param-name>AutoRewrite</param-name>
            <param-value>off</param-value>
        </init-param>
        <init-param>
            <param-name>Transport.SSLCACertificateFile</param-name>
            <param-value>/var/opt/neviscerts/X509-nevisfido-server.cer</param-
value>
        </init-param>
    </servlet>

    <!-- The URL mapping for the nevisFIDO registration as application. -->
    <servlet-mapping>
        <servlet-name>FidoUafRegistrationConnector</servlet-name>
        <url-pattern>/fidouaf/registration/*</url-pattern>
    </servlet-mapping>
```

7.2. Configuration Snippets For Authentication

The following snippet shows a sample nevisProxy configuration that allows you to protect an application through authentication.

This is how it works. The application is accessible through the URL `https://<hostname>/exampleapplication/`. The `IdentityCreationFilter` redirects non-authenticated clients that want to access the application to nevisAuth, to the realm/domain `FIDO_UAF_AUTHENTICATION`. Here, the clients have to authenticate using FIDO UAF. See nevisAuth AuthStates for details on how to configure nevisAuth for authentication with FIDO UAF.

```
    <!-- Authentication filter to protect the application. -->
    <filter>
        <filter-name>AuthenticationFilter</filter-name>
        <filter-class>
ch::nevis::isiweb4::filter::auth::IdentityCreationFilter</filter-class>
        <init-param>
            <param-name>AuthenticationServlet</param-name>
            <param-value>NevisAuthConnector</param-value>
            <description>The configured name of the authentication
servlet</description>
        </init-param>
        <init-param>
            <param-name>LoginRendererServlet</param-name>
            <param-value>BuiltinLoginRenderer</param-value>
            <description>The configured name of the login renderer
servlet</description>
        </init-param>
        <init-param>
            <param-name>Realm</param-name>
            <param-value>FIDO_UAF_AUTHENTICATION</param-value>
            <description>The realm of the authentication</description>
        </init-param>
        <init-param>
            <param-name>InactiveInterval</param-name>
```

```
            <param-value>7200</param-value>
            <description>
                The maximum interval between two request associated to the same
session
                (if deleted or 0, value is taken from nevisAuth 'Domain'
element)
            </description>
        </init-param>
        <init-param>
            <param-name>EntryPointID</param-name>
            <param-value>localhost</param-value>
            <description>The entry point id (will be part of the
sectoken)</description>
        </init-param>
        <!-- The following is required, so that nevisAuth retrieves the body of
the POST operations
             sent to nevisProxy. -->
        <init-param>
            <param-name>InterceptionRedirect</param-name>
            <param-value>never</param-value>
        </init-param>
    </filter>

    <!-- Configuration describing the nevisAuth instance to be used to perform
the authentication. -->
    <servlet>
        <servlet-name>NevisAuthConnector</servlet-name>
        <servlet-
class>ch::nevis::isiweb4::servlet::connector::soap::esauth4::Esauth4ConnectorSe
rvlet</servlet-class>
        <init-param>
            <param-name>TargetURI</param-name>
            <param-value>/nevisauth/services/AuthenticationService</param-
value>
        </init-param>
        <init-param>
            <param-name>Encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>Transport.InetAddress</param-name>
            <param-value>localhost:8991</param-value>
        </init-param>
        <init-param>
            <param-name>Transport.ConnectTimeout</param-name>
            <param-value>45000</param-value>
            <description>
                msec, nevisAuth startup in tomcat5 takes some time and
                listener is open too early
            </description>
        </init-param>
        <init-param>
            <param-name>Transport.RequestTimeout</param-name>
            <param-value>90000</param-value>
            <description>
                msec, 1/3 of this timeout is used to poll nevisAuth for
```

```
                terminated session.
            </description>
        </init-param>
        <init-param>
            <param-name>Transport.SSLClientCertificateFile</param-name>
            <param-value>/var/opt/keybox/default/node_keystore.pem</param-
value>
        </init-param>
        <init-param>
            <param-name>Transport.SSLCACertificateFile</param-name>
            <param-value>/var/opt/keybox/default/truststore.pem</param-value>
        </init-param>
        <init-param>
            <param-name>Transport.SSLCheckPeerHostname</param-name>
            <param-value>false</param-value>
        </init-param>
    </servlet>

    <!-- URL Mapping for the application to be protected. -->
    <filter-mapping>
        <filter-name>AuthenticationFilter</filter-name>
        <url-pattern>/exampleapplication/*</url-pattern>
    </filter-mapping>
```

8. nevisIDM Configuration

nevisIDM requires special configuration to be able to support dispatch targets. This section guides you through the steps necessary to make nevisIDM work with nevisFIDO.

You need to adjust the following nevisIDM configuration:

- Add custom properties to the nevisIDM database (only if the nevisIDM version is previous to 2.75.0).
- Configure client TLS.
- Customize the *standalone.xml* of nevisIDM in order to configure client TLS.

8.1. Custom Properties

If you are using db created by nevisIDM 2.75.0 or later, all the required schema definitions for the dispatch targets are provided out-of-the-box by nevisIDM, and you can skip this section.

nevisFIDO manages persisted dispatch targets by using nevisIDM custom credential objects.
The following SQL script sets up the custom properties used by nevisFIDO:

nevisIDM custom properties SQL script

```
-- Generic Credential policy is required in order to be able to create Generic
Credentials.
https://clientconnect.adnovum.net/nevisdoc/display/NEVISIDMDOC06/Generic+creden
tial
INSERT INTO TIDMA_POLICY_CONFIGURATION
    (POLICY_CONFIGURATION_ID, CTL_TCN, CTL_CRE_UID, CTL_CRE_DAT, CTL_MOD_UID,
CTL_MOD_DAT, POLICY_TYPE, DESCRIPTION, NAME, EXTID, DEFAULT_POLICY, CLIENT_ID)
VALUES
    (901, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'GenericCredentialPolicy', 'Default policy for generic credentials', 'Default
Generic Credential Policy', '901', 1, 100);
commit;

-- Add Dispatcher target property definitions for Generic Credential type
-- Note: using bootstrap user, the fields will be editable on nevisIdm UI,
because the root user has permission "PropertyAttributeAccessOverride"
https://clientconnect.adnovum.net/nevisdoc/display/NEVISIDMDOC06/Database+table
s+related+to+properties
INSERT INTO TIDMA_PROPERTY
    (PROPERTY_ID, CTL_TCN, CTL_CRE_UID, CTL_CRE_DAT, CTL_MOD_UID, CTL_MOD_DAT,
NAME, DESCRIPTION, TYPE, SCOPE, ENCRYPTED, PROPAGATED, MANDATORY_ON_GUI,
STR_MAX_LEN, STR_REGEX, ACCESS_CREATE, ACCESS_MODIFY, UNIQUENESS_SCOPE,
GUI_PRECEDENCE, DISPLAYNAME_DICT_ENTRY_ID, APPLICATION_ID, CLIENT_ID)
VALUES
    (901, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_name', 'Human readable name of the device', 2, 21, 0, 0, 1, 1000,
NULL, 'rw', 'rw', NULL, 0, NULL, NULL, NULL),
    (902, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_app_id', 'The appId of the application where the device is registered
', 2, 21, 0, 0, 1, 100, NULL,'rw', 'rw', NULL, 1, NULL, NULL, NULL),
    (903, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_target', 'The target identifier of the channel', 2, 21, 0, 0, 1, 4096,
NULL, 'rw', 'rw', NULL, 2, NULL, NULL, NULL),
    (904, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_dispatcher', 'The name of the dispatcher', 2, 21, 0, 0, 1, 100, NULL,
'rw', 'rw', NULL, 3, NULL, NULL, NULL),
    (905, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_encryption_key', 'Encryption key used for encrypting the channel data
', 2, 21, 0, 0, 1, 10000, NULL, 'rw', 'rw', NULL, 4, NULL, NULL, NULL),
    (906, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_signature_key', 'The signature key used for signing dispatch channel
information', 2, 21, 0, 0, 1, 10000, NULL, 'rw', 'rw', NULL, 5, NULL, NULL,
NULL),
    (907, 0, 'Default/bootstrap', SYSDATE, 'Default/bootstrap', SYSDATE,
'fidouaf_device_id', 'The device identifier', 2, 21, 0, 0, 1, 10000, NULL, 'rw
', 'rw', NULL, 6, NULL, NULL, NULL);
commit;
```

The script must be applied to the nevisIDM database:

```
mysql -u root --password="" nevisidm < nevisfido-customprops.sql
```

8.2. Client TLS Configuration (Certificates)

Because nevisAuth uses client TLS to communicate with nevisIDM, you need to configure the required certificate data and the CertLoginModule of nevisIDM. See below how to proceed.

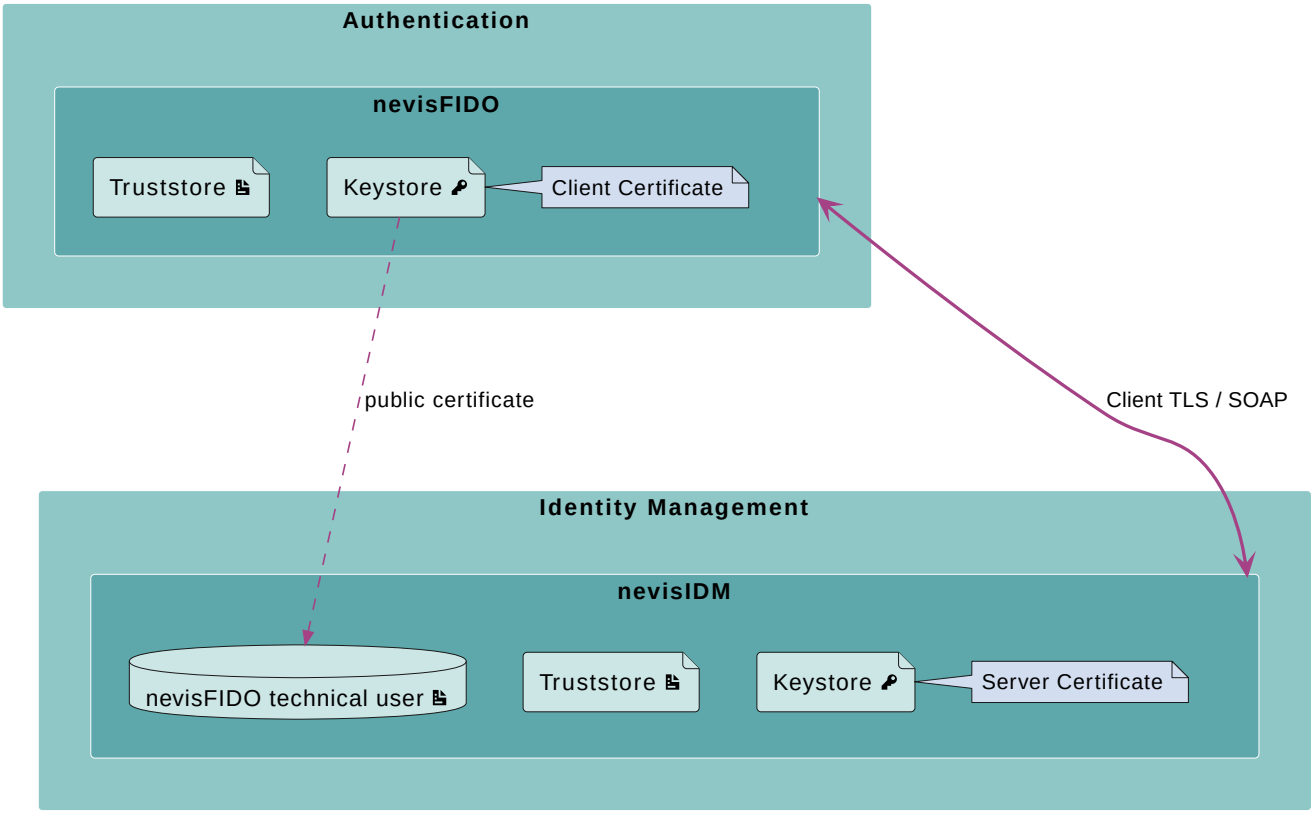


Figure 3. Client TLS overview

Using the PKCS12 truststore for nevisIDM does not work together with WildFly as container. Therefore, it is suggested using JKS for both the nevisIDM key- and truststore. In nevisIDM standalone deployment, PKCS12 can be used with no limitations.

1. Ensure that the `nevisidm.administration-url` property in the *nevisfido.yml* refers to the nevisIDM administration context root. See also the nevisIDM reference guide, chapter "Integration > Configuring certificate login (2-way TLS) for accessing nevisIDM web services".
   - In case of a nevisIDM WildFly deployment, the context root is `nevisidmcc` (for instance `https://<hostname>:8443/nevisidmcc/services/v1/AdminService`).
   - In case of a nevisIDM standalone deployment, the context root is `nevisidm` (for instance `https://<hostname>:8443/nevisidm/services/v1/AdminService`).
2. Create the self-signed certificates and the keystores. All of the following commands create keystores protected with the password `password`.

   Client (nevisFIDO) certificate

   ```
   keytool -genkeypair -keyalg RSA -alias nevisfido -keystore nevisfido-
   keystore.p12 -storetype pkcs12 -storepass password -keypass password
   -validity 360 -keysize 2048 -dname "cn=nevisfido,ou=auth,dc=nevis-
   security,dc=com" --noprompt
   ```

   Server (nevisIDM) certificate (a PKCS12 keystore can be used in case of a nevisIDM standalone deployment)

   ```
   keytool -genkeypair -keyalg RSA -alias nevisidm -keystore nevisidm-
   keystore.jks -storetype jks -storepass password -keypass password -validity
   360 -keysize 2048 -dname "cn=siven.ch,ou=auth,dc=nevis-security,dc=com"
   --noprompt
   ```

The hostname is relevant: The hostname used in the certificate DN is the one used by the hostname verifiers when establishing the HTTPS connection.

3. Create the truststores. The easy way is to use the nevisFIDO keystore as the truststore for nevisIDM and vice versa. However, the clean/safe way is to define a truststore that only contains the public key. For this, use the following commands:

   Truststore configuration (a PKCS12 keystore can be used with a nevisIDM standalone deployment)

   ```
   keytool -export -alias nevisfido -keystore nevisfido-keystore.p12 -storetype
   pkcs12 -storepass password -rfc -file X509_nevisfido.cer

   keytool -importcert -alias nevisfido -file X509_nevisfido.cer -keystore
   nevisidm-truststore.jks -storetype jks -storepass password --noprompt

   keytool -export -alias nevisidm -keystore nevisidm-keystore.jks -storetype
   jks -storepass password -rfc -file X509_nevisidm.cer

   keytool -importcert -alias nevisidm -file X509_nevisidm.cer -keystore
   nevisfido-truststore.p12 -storetype pkcs12 -storepass password --noprompt
   ```

4. Configure the nevisIDM Java Virtual Machine arguments (only in the context of a nevisIDM Wildfly deployment). Add the following elements to `/var/opt/nevisidm/nevisidm/conf/vmargs.conf`:

   Additional nevisIDM vmargs

   ```
   -Dch.adnovum.nevisidm.web.servlets.ForwardServlet.Enabled=true
   -Dch.adnovum.nevisidm.web.servlets.ForwardServlet.ForwardPath=/services/
   ```

   See also the nevisIDM reference guide, chapter "Integration > Configuring certificate login (2-way TLS) for accessing nevisIDM web services".

   Do not add the Java Virtual Machine arguments above in case of a nevisIDM standalone deployment.

5. Update the nevisIDM configuration:
   - In case of nevisIDM WildFly deployment, update the file `/var/opt/adnwildfly/instances/nevisidm/standalone/configuration/standalone.xml`.
     Copy the previously created nevisIDM keystore and truststore to the nevisIDM machine. You will have to update the paths of the keystores and truststores referenced in the configuration (look for `nevisidm-keystore.jks` and `nevisidm-truststore.jks`).
     For more information, refer to nevisIDM WildFly standalone.xml.
   - In case of nevisIDM standalone deployment, update the file `/var/opt/nevisidm/nevisidm/conf/nevisidm-prod.properties`.
     Copy the previously created nevisIDM keystore and truststore to the nevisIDM machine. You will have to update the paths of the keystores and truststores referenced in the configuration (look for `nevisidm-keystore.p12` and `nevisidm-truststore.p12`).
     For more information, refer to nevisIDM Standalone nevisidm-prop.properties.
6. Restart nevisIDM:

   ```
   nevisidm restart
   ```

7. Add the public certificate of nevisFIDO to the `nevisfido` user in nevisIDM. Proceed as follows:
   a. Go to the nevisIDM administration UI. In the case of server `siven.ch`, the URL of the administration UI is: `https://siven.ch/nevisidm/admin`.
   b. The initial credentials to log in to the nevisIDM admin UI are id=`bootstrap` / password=`generated`. You will be prompted to reset the password. By convention, reset it to `Generated1!`.

c. In the nevisIDM administration UI, search for the user `nevisfido`.
d. Add a credential of type "certificate" to the user `nevisfido`. Uncheck the *Create ticket for upload* checkbox. The certificate to be added is the public key of the nevisFIDO certificate. Print the public key on the shell and copy & paste it to the nevisIDM administration UI. To print the certificate, you can run one the following command:

```
keytool -export -alias nevisfido -keystore nevisfido-keystore.p12
-storetype pkcs12 -storepass password -rfc
```

or

```
keytool -export -alias nevisfido -keystore nevisidm-truststore.jks
-storetype jks -storepass password -rfc
```

Some notes to the last step:

- *The browser will complain about the presented certificate (which is self-signed). Accept the certificate anyway.*
- *If you performed the above procedure on your pc before, there will already be a certificate in your browser. Remove this certificate from your browser. This is necessary to prevent you from being blocked by the browser: Because the previously generated certificate does not match the new certificate, the browser will not allow you to carry on with the connection.*

8.3. nevisIDM WildFly standalone.xml

In case of a nevisIDM WildFly deployment, you need to adjust the *standalone.xml* file of nevisIDM. Pay special attention to the following sections:

8.3.1. HTTPS Security Realm

Add the keystore and truststore configurations to the *HttpsRealm* security realm:

```
<security-realm name="HttpsRealm">
    <server-identities>
        <ssl>
            <keystore path="/var/opt/certs/nevisidm-keystore.jks" keystore-
password="password" provider="jks" />
        </ssl>
    </server-identities>
    <authentication>
        <truststore path="/var/opt/certs/nevisidm-truststore.jks" keystore-
password="password" provider="jks" />
        <local default-user="$local" skip-group-loading="true"/>
        <properties path="mgmt-users.properties" relative-to=
"jboss.server.config.dir"/>
    </authentication>
</security-realm>
```

8.3.2. Client Certificate Security Domain

Configure the `NevisIDMClientCertDomain` security domain:

```
<security-domain name="NevisIDMClientCertDomain" cache-type="default">
    <authentication>
        <login-module code="ch.adnovum.nevisidm.jaas.CertLoginModule" flag=
"required" />
    </authentication>
```

8.3.3. Server Configuration

Add an HTTPS listener to the configured security realm:

```
<server name="default-server">
    ...
    <https-listener name="cc-ssl" socket-binding="https" security-realm=
"HttpsRealm" verify-client="required"/>
    ...
</server>
```

8.3.4. Interface Configuration

Adjust the interface configuration:

```
<interfaces>
    <interface name="management">
        <inet-address value="0.0.0.0"/>
    </interface>
    <interface name="public">
        <inet-address value="0.0.0.0"/>
    </interface>
</interfaces>
```

8.3.5. Socket Binding Configuration

Update the socket binding configuration:

```
<socket-binding-group name="standard-sockets" default-interface="public" port-
offset="${jboss.socket.binding.port-offset:0}">
    ....
    <socket-binding name="https" port="8443" fixed-port="true"/>
</socket-binding-group>
```

8.3.6. Deployment Configuration

Add the required deployment configuration:

```
<deployments>
    <deployment name="nevisidm-application-2.64.1.0-mysql.wildfly10.ear"
runtime-name="nevisidm-application-2.64.1.0-mysql.wildfly10.ear">
       <content sha1="cfccdfe81db5928424d02a71367c5456c2ccbf81"/>
    </deployment>
</deployments>
```

8.3.7. Complete standalone.xml Example

The following code sample shows a complete *standalone.xml* configuration to be used with nevisIDM WildFly deployment:

```
<?xml version='1.0' encoding='UTF-8'?>

<server xmlns="urn:jboss:domain:4.2">

    <extensions>
        <extension module="org.jboss.as.connector"/>
        <extension module="org.jboss.as.deployment-scanner"/>
        <extension module="org.jboss.as.ee"/>
        <extension module="org.jboss.as.jmx"/>
        <extension module="org.jboss.as.logging"/>
        <extension module="org.jboss.as.naming"/>
        <extension module="org.jboss.as.mail"/>
        <extension module="org.jboss.as.remoting"/>
        <extension module="org.jboss.as.security"/>
        <extension module="org.jboss.as.transactions"/>
        <extension module="org.jboss.as.webservices"/>
        <extension module="org.wildfly.extension.io"/>
        <extension module="org.wildfly.extension.undertow"/>
        <extension module="org.wildfly.extension.messaging-activemq"/>
        <extension module="org.wildfly.extension.bean-validation"/>
    </extensions>


    <management>
        <security-realms>
            <security-realm name="ManagementRealm">
                <authentication>
                    <local default-user="$local" skip-group-loading="true"/>
                    <properties path="mgmt-users.properties" relative-to=
"jboss.server.config.dir"/>
                </authentication>
                <authorization map-groups-to-roles="false">
                    <properties path="mgmt-groups.properties" relative-to=
"jboss.server.config.dir"/>
                </authorization>
            </security-realm>
            <security-realm name="ApplicationRealm">
                <authentication>
                    <local default-user="$local" allowed-users="*" skip-group-
loading="true"/>
                    <properties path="application-users.properties" relative-
to="jboss.server.config.dir"/>
                </authentication>
                <authorization>
                    <properties path="application-roles.properties" relative-
to="jboss.server.config.dir"/>
                </authorization>
            </security-realm>
            <security-realm name="HttpsRealm">
                <server-identities>
```

```
                    <ssl>
                        <keystore path="/var/opt/certs/nevisidm-keystore.jks"
keystore-password="password" provider="jks" />
                    </ssl>
                </server-identities>
                <authentication>
                    <truststore path="/var/opt/certs/nevisidm-truststore.jks"
keystore-password="password" provider="jks" />
                    <local default-user="$local" skip-group-loading="true"/>
                    <properties path="mgmt-users.properties" relative-to=
"jboss.server.config.dir"/>
                </authentication>
            </security-realm>
        </security-realms>
        <audit-log>
            <formatters>
                <json-formatter name="json-formatter"/>
            </formatters>
            <handlers>
                <file-handler name="file" formatter="json-formatter" path=
"audit-log.log" relative-to="jboss.server.data.dir"/>
            </handlers>
            <logger log-boot="true" log-read-only="false" enabled="false">
                <handlers>
                    <handler name="file"/>
                </handlers>
            </logger>
        </audit-log>
        <management-interfaces>
            <http-interface security-realm="ManagementRealm" http-upgrade-
enabled="true">
                <socket-binding http="management-http"/>
            </http-interface>
        </management-interfaces>
        <access-control provider="simple">
            <role-mapping>
                <role name="SuperUser">
                    <include>
                        <user name="$local"/>
                    </include>
                </role>
            </role-mapping>
        </access-control>
    </management>

    <profile>
        <subsystem xmlns="urn:jboss:domain:logging:3.0">
            <add-logging-api-dependencies value="false"/>
            <console-handler name="CONSOLE">
                <level name="INFO"/>
                <formatter>
                    <named-formatter name="COLOR-PATTERN"/>
                </formatter>
            </console-handler>
            <periodic-rotating-file-handler name="FILE" autoflush="true">
                <formatter>
```

```
                    <named-formatter name="PATTERN"/>
                </formatter>
                <file relative-to="jboss.server.log.dir" path="server.log"/>
                <suffix value=".yyyy-MM-dd"/>
                <append value="true"/>
            </periodic-rotating-file-handler>
            <logger category="io.undertow">
                <level name="WARN"/>
            </logger>
            <logger category="com.arjuna">
                <level name="WARN"/>
            </logger>
            <logger category="org.apache.tomcat.util.modeler">
                <level name="WARN"/>
            </logger>
            <logger category="org.jboss.as.config">
                <level name="INFO"/>
            </logger>
            <logger category="sun.rmi">
                <level name="WARN"/>
            </logger>
            <logger category="ch.nevis.idm">
                <level name="INFO"/>
            </logger>
            <logger category="ch.adnovum.nevisidm">
                <level name="ERROR"/>
            </logger>
            <logger category="ch.nevis.ninja">
                <level name="INFO"/>
            </logger>
            <root-logger>
                <level name="INFO"/>
                <handlers>
                    <handler name="CONSOLE"/>
                    <handler name="FILE"/>
                </handlers>
            </root-logger>
            <formatter name="PATTERN">
                <pattern-formatter pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p
[%c] (%t) %s%E%n"/>
            </formatter>
            <formatter name="COLOR-PATTERN">
                <pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c]
(%t) %s%E%n"/>
            </formatter>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:deployment-scanner:2.0">
            <deployment-scanner path="deployments" relative-to=
"jboss.server.base.dir" scan-interval="5000" runtime-failure-causes-rollback=
"${jboss.deployment.scanner.rollback.on.failure:false}"/>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:ee:4.0">
            <spec-descriptor-property-replacement>false</spec-descriptor-
property-replacement>
            <concurrent>
                <context-services>
```

```
                    <context-service name="default" jndi-name=
"java:jboss/ee/concurrency/context/default" use-transaction-setup-provider=
"true"/>
                </context-services>
                <managed-executor-services>
                    <managed-executor-service name="default" jndi-name=
"java:jboss/ee/concurrency/executor/default" context-service="default" hung-
task-threshold="60000" core-threads="5" max-threads="25" keepalive-time="5000
"/>
                </managed-executor-services>
                <managed-scheduled-executor-services>
                    <managed-scheduled-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/scheduler/default" context-service="default"
hung-task-threshold="60000" core-threads="2" keepalive-time="3000"/>
                </managed-scheduled-executor-services>
            </concurrent>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:bean-validation:1.0"/>
        <subsystem xmlns="urn:jboss:domain:jmx:1.3">
            <expose-resolved-model/>
            <expose-expression-model/>
            <remoting-connector/>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:io:1.1">
            <worker name="default"/>
            <buffer-pool name="default"/>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:jca:4.0">
            <archive-validation enabled="true" fail-on-error="true" fail-on-
warn="false"/>
            <bean-validation enabled="true"/>
            <default-workmanager>
                <short-running-threads>
                    <core-threads count="50"/>
                    <queue-length count="50"/>
                    <max-threads count="50"/>
                    <keepalive-time time="10" unit="seconds"/>
                </short-running-threads>
                <long-running-threads>
                    <core-threads count="50"/>
                    <queue-length count="50"/>
                    <max-threads count="50"/>
                    <keepalive-time time="10" unit="seconds"/>
                </long-running-threads>
            </default-workmanager>
            <cached-connection-manager/>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:mail:2.0"/>
        <subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
            <server name="default">
                <security-setting name="#">
                    <role name="consumer" consume="true"/>
                    <role name="producer" send="true"/>
                </security-setting>
                <address-setting name="#" dead-letter-address="jms.queue.DLQ"
expiry-address="jms.queue.ExpiryQueue" max-size-bytes="10485760" page-size-
```

```
bytes="2097152" message-counter-history-day-limit="10"/>
                <remote-connector name="netty" socket-binding="msg"/>
                <in-vm-connector name="in-vm" server-id="0"/>
                <remote-acceptor name="netty" socket-binding="msg"/>
                <in-vm-acceptor name="in-vm" server-id="0"/>
                <jms-queue name="ExpiryQueue" entries=
"java:/jms/queue/ExpiryQueue"/>
                <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
                <jms-queue name="Provisioning" entries=
"java:/jms/queue/Provisioning java:/jboss/exported/jms/queue/Provisioning"/>
                <connection-factory name="InVmConnectionFactory" entries=
"java:/ConnectionFactory" connectors="in-vm"/>
            </server>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:naming:2.0">
            <remote-naming/>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:remoting:3.0"/>
        <subsystem xmlns="urn:jboss:domain:resource-adapters:4.0"/>
        <subsystem xmlns="urn:jboss:domain:security:1.2">
            <security-domains>
                <security-domain name="other" cache-type="default">
                    <authentication>
                        <login-module code="Remoting" flag="optional">
                            <module-option name="password-stacking" value=
"useFirstPass"/>
                        </login-module>
                        <login-module code="RealmDirect" flag="required">
                            <module-option name="password-stacking" value=
"useFirstPass"/>
                        </login-module>
                    </authentication>
                </security-domain>
                <security-domain name="NevisSecTokenDomain" cache-type="
default">
                    <authentication>
                        <login-module code=
"ch.nevis.ninja.jboss.auth.NinjaJbossLoginModuleImpl" flag="sufficient" module
="ch.nevis.ninja">
                            <module-option name="NevisSignerCertificate" value
="/var/opt/neviskeybox/default/nevis/truststore.jks"/>
                            <module-option name="UserGetter" value=
"AttributeUserGetter(source=loginId)"/>
                            <module-option name="AdjustIdentity" value="true"/>
                            <module-option name="LogDebug" value="false"/>
                            <module-option name="RoleGetters" value=
"ch.nevis.ninja.commons.mapping.StaticRoleGetter,
ch.nevis.ninja.commons.mapping.TokenRoleGetter"/>
                        </login-module>
                        <login-module code=
"ch.adnovum.nevisidm.jaas.BasicAuthLoginModule" flag="required"/>
                    </authentication>
                </security-domain>
                <security-domain name="NevisIDMClientCertDomain" cache-type=
"default">
                    <authentication>
                        <login-module code=
```

```
"ch.adnovum.nevisidm.jaas.CertLoginModule" flag="required">
                        </login-module>
                    </authentication>
                </security-domain>
            </security-domains>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:transactions:3.0">
            <core-environment>
                <process-id>
                    <uuid/>
                </process-id>
            </core-environment>
            <recovery-environment socket-binding="txn-recovery-environment"
status-socket-binding="txn-status-manager"/>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:undertow:3.1">
            <buffer-cache name="default"/>
            <server name="default-server">
                <http-listener name="default" socket-binding="http" redirect-
socket="http"/>
                <https-listener name="cc-ssl" socket-binding="https" security-
realm="HttpsRealm" verify-client="required"/>
                <host name="default-host" alias="localhost"/>
            </server>
            <servlet-container name="default">
                <jsp-config/>
            </servlet-container>
        </subsystem>
        <subsystem xmlns="urn:jboss:domain:webservices:2.0">
            <modify-wsdl-address>true</modify-wsdl-address>
            <wsdl-host>${jboss.bind.address:localhost}</wsdl-host>
            <wsdl-port>${jboss.management.http.port:1}</wsdl-port>
            <endpoint-config name="Standard-Endpoint-Config"/>
            <endpoint-config name="Recording-Endpoint-Config">
                <pre-handler-chain name="recording-handlers" protocol-bindings
="##SOAP11_HTTP ##SOAP11_HTTP_MTOM ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
                    <handler name="RecordingHandler" class=
"org.jboss.ws.common.invocation.RecordingServerHandler"/>
                </pre-handler-chain>
            </endpoint-config>
            <client-config name="Standard-Client-Config"/>
        </subsystem>
    </profile>

    <interfaces>
        <interface name="management">
            <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
        </interface>
        <interface name="public">
            <inet-address value="0.0.0.0"/>
        </interface>
    </interfaces>

    <socket-binding-group name="standard-sockets" default-interface="public"
port-offset="${jboss.socket.binding.port-offset:0}">
        <socket-binding name="management-http" interface="management" port=
```

```
"${jboss.management.http.port:1}"/>
        <socket-binding name="http" port="8989" fixed-port="true"/>
        <socket-binding name="txn-recovery-environment" port="2"/>
        <socket-binding name="txn-status-manager" port="3"/>
        <socket-binding name="msg" port="4"/>
        <socket-binding name="https" port="8443" fixed-port="true"/>
    </socket-binding-group>

    <deployments>
        <deployment name="nevisidm-application-2.64.1.0-mysql.wildfly10.ear"
runtime-name="nevisidm-application-2.64.1.0-mysql.wildfly10.ear">
            <content sha1="cfccdfe81db5928424d02a71367c5456c2ccbf81"/>
        </deployment>
    </deployments>
</server>
```

8.4. nevisIDM Standalone nevisidm-prop.properties

In case of a nevisIDM standalone deployment, you need to adjust the *nevisidm-prop.properties* file of nevisIDM. These are the key elements:

```
server.tls.require-client-auth=true
server.host=0.0.0.0
server.port=8443
server.tls.keystore=/var/opt/certs/nevisidm-keystore.p12
server.tls.keystore-passphrase=password
server.tls.truststore=/var/opt/certs/nevisidm-truststore.p12
server.tls.truststore-passphrase=password
```

8.4.1. Complete nevisidm-prop.properties Example

The following code sample shows a complete *nevisidm-prop.properties* configuration to be used with nevisIDM standalone deployment:

```
# Server configuration
server.tls.enabled=true
server.tls.require-client-auth=true
server.host=0.0.0.0
server.port=8443
server.tls.keystore=/var/opt/certs/nevisidm-keystore.p12
server.tls.keystore-passphrase=password
server.tls.truststore=/var/opt/certs/nevisidm-truststore.p12
server.tls.truststore-passphrase=password

# DB connectivity
database.connection.url=jdbc:mysql://localhost:3306/nevisidm?autocommit=0
database.connection.username=UIDM02
database.connection.password=UIDM02

# Auditing
application.modules.auditing.provider=jsonAuditProvider
application.modules.auditing.file=${server.log.dir}/audit.json

# Folder to store pdf
application.modules.printing.dir.target=/var/tmp/nevisidm_pdfs

# mail server
application.mail.smtp.host=localhost
application.mail.smtp.port=25
application.mail.sender=nevisidm@nevis-security.com

# enables enterprise role feature
application.feature.enterpriserole.enabled=true

# enable application.feature.multiclientmode.enabled
application.feature.multiclientmode.enabled=true

# Experimental REST service
experimentalRest.enabled=true

# Ninja Truststore
server.auth.ninja.truststore=/var/opt/neviskeybox/default/nevis/truststore.jks
multiClientMode=true
```

9. HTTP API

The nevisFIDO HTTP API documentation describes communication between the (FIDO) client, the nevisFIDO server and the relying party applications. It describes where and how communication with the nevisFIDO server is possible, by using what protocol.

The nevisFIDO API is an HTTP API.

The nevisFIDO API offers the following services:

- Registration Request Service
- Registration Response Service
- Authentication Request Service
- Authentication Response Service

- Deregistration Request Service
- Facets Service

The above services are standard FIDO services that are part of the FIDO Universal Authentication Framework (UAF). The following services are specific for nevisFIDO:

- Out-of-band services
  - Dispatch Target Service
  - Dispatch Token Service
  - Redeem Token Service
  - Create Token Service
- Status Service

Per service, the following elements are described:

- Base URL
- HTTP methods
- Request headers
- Request body
- Response headers
- Response body
- Example request
- Example response
- HTTP status code

The context path of nevisFIDO is `/nevisfido`. It is shared by all exposed endpoints.

The first chapter describes the attributes that are shared among the different services.

9.1. Shared Structures

This chapter describes all attributes that are shared among the different operations and services.

The attributes refer to various types or structures of information (for example, some attributes define the *(operation) header*; others define the *channel binding*). All attributes belonging to the same information type or structure are listed in one dictionary (e.g., the *OperationHeader Dictionary* or the *ChannelBinding Dictionary*).

9.1.1. Version

The *Version* attributes refer to the UAF protocol version.

Table 4. Version dictionary

Attribute	Type	Description	Optional
`major`	`number`	Major version, must be `1`.	false
`minor`	`number`	Minor version, must be `1`.	false

nevisFIDO only supports the FIDO UAF protocol version 1.1.

9.1.2. Operation Header

The *Operation Header* attributes define the operation header of UAF request and response messages.

Table 5. OperationHeader dictionary

Attribute	Type	Description	Optional
`upv`	`Version`	Refers to the UAF protocol version. For details, see the Version dictionary.	false
`op`	`String`	Name of the FIDO operation this message relates to.	false
`appID`	`String`	The application identifier that the relying party would like to assert.	true
`serverData`	`String`	A session identifier created by the relying party.	true
`exts`	`Extension[]`	Defines a list of UAF message extensions. For details, see the Extension dictionary.	true

9.1.3. Extension

The *Extension* attributes define (UAF) extensions to the operation header. The extension attributes are set in the operation header attribute `exts` (see also the OperationHeader dictionary).

Table 6. Extension dictionary

Attribute	Type	Description	Optional
`id`	`String`	Identifies the extension.	false
`data`	`String`	Contains arbitrary data with semantics agreed between the server and client. Binary data is base64url-encoded. This field may be empty.	false
`fail_if_unknown`	`Boolean`	Indicates whether unknown extensions must be ignored (false) or must lead to an error (true).	false

9.1.3.1. Proprietary Extensions

Currently, there is one proprietary extension, the **Session ID Extension**. This extension contains a unique session identifier. This unique session identifier can be used to query the current authentication status of the FIDO session through the Status Service.

The Session ID Extension has the following settings:

id
: ch.nevis.auth.fido.uaf.sessionid

data
: <unique session identifier>

fail_if_unknown
: false

Example:

```
{
  "id" : "ch.nevis.auth.fido.uaf.sessionid",
  "data" : "d61e461e-c597-4ed3-9d71-12d1c0e3556c",
  "fail_if_unknown" : false
}
```

9.1.4. Final Challenge Parameters

The *Final Challenge* parameters (or attributes) define all information required for the server to verify the final challenge. For a description of the final challenge parameters, see the *FinalChallengeParams* dictionary below.

The final challenge parameters are sent in the request messages of the Registration and Authentication Response Services, as a base64url-encoded UTF-8 string. The corresponding attribute is *fcParams*. For more information, see the chapters Registration Response Service and Authentication Response Service.

Table 7. FinalChallengeParams dictionary

Attribute	Type	Description	Optional
`appID`	`String`	The application ID. The value must be taken from the `appID` field/attribute of the operation header (see also OperationHeader dictionary).	false
`challenge`	`String`	The challenge. The value must be taken from the `challenge` field/attribute of the request (e.g., `RegistrationRequest.challenge`, `AuthenticationRequest.challenge` - see also RegistrationRequest dictionary and AuthenticationRequest dictionary).	false
`facetID`	`String`	The facet ID. The value is determined by the FIDO client and depends on the calling application.	false
`channelBinding`	`ChannelBinding`	This attribute contains the TLS information that the FIDO client must send to the FIDO server. The server needs this information to bind the TLS channel to the FIDO operation. The value of this attribute is defined by the *Channel Binding* attributes. For more information on the channel binding attributes, see ChannelBinding dictionary.	false

9.1.5. Channel Binding

The *Channel Binding* attributes specify the binding of the TLS channel to the FIDO operation. For a description of the channel binding attributes, see the *ChannelBinding* dictionary below.

The channel binding information is part of the final challenge information. The channel binding attributes are set in the final challenge parameter *channelBinding*.

Table 8. ChannelBinding dictionary

Attribute	Type	Description	Optional
`serverEndPoint`	`String`	If the TLS server certificate is available, the attribute `serverEndPoint` must be set to the base64url-encoded hash of the certificate. If the certificate is not available or the hash cannot be determined, the attribute must be absent.	true
`tlsServerCertificate`	`String`	The attribute `tlsServerCertificate` is set to the base64url-encoded, DER-encoded TLS server certificate. If this data is not available to the FIDO Client, the attribute must be absent.	true

Attribute	Type	Description	Optional
`tlsUnique`	`String`	The attribute `tlsUnique` is set to the base64url-encoded TLS channel *Finished* structure. If this data is not available to the FIDO client, the attribute must be absent.	true
`cid_pubkey`	`String`	The attribute `cid_pubkey` is set to the base64url-encoded serialized *JwkKey* structure using UTF-8 encoding. Set the attribute to "unused", if the TLS *ChannelID* information is supported by the client-side TLS stack but has not been signaled by the TLS (web) server. The attribute must be absent, if the client TLS stack does not provide TLS *ChannelID* information at all to the processing entity (e.g., the web browser or client application).	true

9.1.6. UAF Status Codes

The UAF status codes indicate the result of the UAF operation at the FIDO server side. They are included in the response messages.

The table below describes the UAF status codes.

Table 9. UAF status codes

Code	Meaning
1200	**OK** Operation completed.
1202	**Accepted** Message accepted, but not completed at this time.
1400	**Bad Request** The server did not understand the message.
1401	**Unauthorized** The username must be authenticated to perform this operation, or the key ID is not associated with this username.
1403	**Forbidden** The username is not allowed to perform this operation. The client should not retry.
1404	**Not Found**
1408	**Request Timeout**
1480	**Unknown AAID** The server was unable to locate authoritative metadata for the AAID (Authenticator Attestation ID).
1481	**Unknown Key ID** The server was unable to locate a registration for the given username and key ID combination. This error indicates that there is an invalid registration on the user's device. When this error occurs, it is recommended that the FIDO client deletes the key from the local device.
1490	**Channel Binding Refused** The server refused to service the request due to missing or mismatched channel binding(s).

Code	Meaning
1491	**Request Invalid** The server refused to service the request because the request message nonce was unknown, expired or the server has previously serviced a message with the same nonce and username.
1492	**Unacceptable Authenticator** The authenticator is not acceptable according to the server's policy, for example because the capability registry used by the server reported different capabilities than those reported at the client-side discovery.
1493	**Revoked Authenticator** The authenticator is considered revoked by the server.
1494	**Unacceptable Key** The used key is unacceptable, for example because it is on a list of known weak keys or uses insecure parameter choices.
1495	**Unacceptable Algorithm** The server believes the authenticator to be capable of using a stronger mutually-agreeable algorithm than was presented in the request.
1496	**Unacceptable Attestation** The attestation(s) provided were not accepted by the server.
1497	**Unacceptable Client Capabilities** The server was unable or unwilling to use required capabilities provided supplementary to the authenticator by the client software.
1498	**Unacceptable Content** There was a problem with the contents of the message and the server was unwilling or unable to process it.
1500	**Internal Server Error**

9.1.7. Client Error Codes

The client error codes are generated by the FIDO UAF client, which is not part of the nevisFIDO product. This means that also these client error codes are not part of nevisFIDO. Therefore, the following descriptions of the client error codes are based on the client error code descriptions in the official FIDO UAF documentation (and do not come from our own developers). For more information, please refer to FIDO UAF client error codes.

The client error codes show the result of an UAF operation on the FIDO client side. The client error codes are included in the `SendUAFResponse` sent by the client to the FIDO server. They are exposed by the Status Service.

The table below describes the client error codes.

Table 10. Client error codes

Code	Meaning
0	**No Error** The operation has been completed without encountering any error condition.
1	**Wait User Action** Waiting for the user to perform a certain action, for example, to select an authenticator in the FIDO client user interface, to perform verification, or to complete an enrollment step with an authenticator.

Code	Meaning
2	**Insecure Transport** Indicates that the transport is insecure. For example, the value of the `window.location.protocol` object is not "HTTPS", or the Document Object Model DOM contains insecure mixed content.
3	**User Cancelled** The user declined any part of the interaction necessary to complete the registration.
4	**Unsupported Version** The UAF message does not include a protocol version supported by this FIDO UAF client.
5	**No Suitable Authenticator** There is no authenticator available that matches the authenticator policy specified in the UAF message. Another possibility is that the user declined to consent to the use of a suitable authenticator.
6	**Protocol Error** A violation of the UAF protocol occurred. For example, the interaction may have timed out, the origin associated with the message may not match the origin of the calling DOM context, or the protocol message may be malformed or tampered with.
7	**Untrusted Facet ID** The client declined to process the operation. This is because the caller's calculated facet identifier was missing from the list of trusted application identifiers specified in the request message.
9	**Key Disappeared Permanently** Indicates that the UAuth key disappeared from the authenticator and cannot be restored.
12	**Authenticator Access Denied** The authenticator denied access to the resulting request.
13	**Invalid Transaction Content** Transaction content cannot be rendered. For example, the format does not fit the authenticator's need.
14	**User Not Responsive** The user took too long to follow an instruction. For example, the user did not swipe the finger within the accepted time frame.
15	**Insufficient Authenticator Resources** Indicates that the authenticator has not enough resources to perform the requested task.
16	**User Lockout** If a user produces too many failed verification attempts, he/she will be locked. Usually, the user must perform an unlock operation to re-enable the main verification method. Such an unlock operation could be, for example, an alternative password authentication. This error code indicates that either such an unlock operation does not exist, or that the ASM/authenticator cannot automatically trigger it. The error code will be reported.
17	**User Not Enrolled** The operation failed because the user is not enrolled, and the authenticator cannot automatically trigger user enrollment.
255	**Unknown** Indicates an error condition not described by the above-listed codes.

9.1.8. ASM Status Codes

The ASM status codes are generated by the authenticator module of the FIDO UAF client, which is not part of the nevisFIDO product. This means that also these ASM status codes are not part of nevisFIDO. Therefore, the following descriptions of the ASM status codes are based on the ASM status code descriptions in the official FIDO UAF documentation (and do not come from our own developers). For more information, please refer to FIDO UAF ASM status codes.

The ASM status codes show the result of an UAF operation at the FIDO client side, on the level of the authenticator module (ASM = authenticator-specific module). The ASM status codes are included in the `SendUAFResponse` object sent by the client to the FIDO server. They are exposed by the Status Service.

The table below describes the ASM status codes.

Table 11. ASM status codes

Code	Meaning
0	**OK** No error condition encountered.
1	**Error** An unknown error has been encountered during the processing.
2	**Access Denied** Access to this request is denied.
3	**User Cancelled** Indicates that the user explicitly canceled the request.
4	**Cannot Render Transaction Content** Transaction content cannot be rendered. For example, the format does not fit the authenticator's need.
9	**Key Disappeared Permanently** Indicates that the UAuth key disappeared from the authenticator and cannot be restored.
11	**Authenticator Disconnected** Indicates that the authenticator is no longer connected to the ASM.
14	**User not Responsive** The user took too long to follow an instruction. For example, the user did not swipe the finger within the accepted time frame.
15	**Insufficient Authentication Resources** The authenticator has not enough resources to perform the requested task.
16	**User Lockout** If a user produces too many failed verification attempts, he/she will be locked. Usually, the user must perform an unlock operation to re-enable the main verification method. Such an unlock operation could be, for example, an alternative password authentication. This error code indicates that either such an unlock operation does not exist, or that the ASM/authenticator cannot automatically trigger it. The error code will be reported.
17	**User not Enrolled** The operation failed because the user is not enrolled, and the authenticator cannot automatically trigger user enrollment.

9.1.9. Authenticators

The *Authenticator* attributes contain information about the authenticators.

- In the case of a successfully completed registration or authentication, the *Authenticator* attributes refer to the authenticators used to generate the validated assertions.
- In the case of deregistration, the *Authenticator* attributes refer to the deregistered authenticators.

This information is exposed by the Status Service.

Table 12. Authenticator dictionary

Attribute	Type	Description	Optional
`aaid`	`String`	The AAID of the authenticator used to generate the assertions.	false

9.2. Registration Request Service

By calling the Registration Request Service and requesting a `RegistrationRequest` object from the FIDO server, the user/FIDO client initiates the FIDO registration process. This chapter describes the request and response messages between the FIDO client and Server when calling the Registration Request Service.

It is recommended to protect this service using a SecToken. See Authorization for more information.

9.2.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/uaf/1.1/request/registration
```

9.2.2. HTTP Methods

`POST` is the only supported HTTP method.

9.2.3. Request Headers

The following request headers are mandatory:

Table 13. Mandatory request headers - Registration Request Service

Name	Description
`Accept`	Accept header, **must** be `application/fido+uaf`.
`Content-Type`	Content type header, **must** be `application/fido+uaf;charset=UTF-8`.

9.2.4. Request Body

The Registration Request Service requires from the FIDO client a JSON payload with a `GetUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `GetUAFRequest` object has the following structure:

Table 14. GetUAFRequest object - Registration Request Service

Attribute	Type	Description	Optional
`op`	`String`	The request operation, must be set to `Reg`.	false

Attribute	Type	Description	Optional
`previousRequest`	`String`	If the application is requesting a new UAF request message because the previous one expired, the previous one could be sent to the server.	true
`context`	`String`	The contextual information must be a **stringified JSON** object that conforms to the Context.	false

The `previousRequest` parameter is ignored and not handled by nevisFIDO.

9.2.4.1. Context

The *Context* dictionary contains all attributes that can be included in the `context` part of the request body.

Table 15. Context dictionary

Attribute	Type	Description	Optional
`username`	`String`	Identity information regarding the user on whose behalf the FIDO client is operating. In the case of the `idm` credential repository, the accepted type of username (`loginId`, `email`, etc.) depends on how the username mapper of the credential repository is configured.	false

9.2.5. Response Headers

The following response headers will be set:

Table 16. Response headers - Registration Request Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.2.6. Response Body

The Registration Request Service returns a JSON body with a `ReturnUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `ReturnUAFRequest` object has the following structure:

Table 17. ReturnUAFRequest object - Registration Request Service

Path	Type	Description
`statusCode`	`Number`	UAF status code for the operation.
`uafRequest`	`String`	The new UAF request message if the server decides to issue one.
`op`	`String`	Hint to the client regarding the operation type of the message, must be set to `Reg`.
`lifetimeMillis`	`Number`	Hint informing the client application of the lifetime of the message in milliseconds. Absent if the operation was not successful.

The `uafRequest` part of the `ReturnUAFRequest` object contains the `RegistrationRequest` object. The *RegistrationRequest* dictionary includes the attributes that define a `RegistrationRequest` object. The following table describes the *RegistrationRequest* dictionary:

Table 18. RegistrationRequest dictionary

Attribute	Type	Description	Optional
`header`	`OperationHeader`	The `header` defines the operation header of the UAF messages coming from the Registration Request Service. Within the `header`, the operation header attributes are specified. For a description of these attributes, see OperationHeader dictionary. Note that the attribute `header.op` must be set to "Reg".	false
`challenge`	`String`	Server-provided challenge value.	false
`username`	`String`	A human-readable username intended to allow the user to distinguish and select from among different accounts at the same relying party.	false
`policy`	`Policy`	Describes which types of authenticators are acceptable for this registration operation.	false

9.2.7. Example Request

```
POST /nevisfido/uaf/1.1/request/registration HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 59

{
  "context" : "{\"username\":\"jeff\"}",
  "op" : "Reg"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/request/registration' -i -X
POST \
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "context" : "{\"username\":\"jeff\"}",
  "op" : "Reg"
}'
```

9.2.8. Example Response

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:22 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 750

{
  "lifetimeMillis" : 300000,
  "uafRequest" : "[{\"header\":{\"serverData\":\"MFBTmvvXj2A1ky-
TBchDeS5L5Xm4o6I6HElpiVAi_AOewkOpsMALIEft2j_9-
POox1kSJ_1YiIrJhWxUI4YV4w\",\"upv\":{\"major\":1,\"minor\":1},\"op\":\"Reg\",\"
appID\":\"https://www.siven.ch/appID\",\"exts\":[{\"id\":\"ch.nevis.auth.fido.u
af.sessionid\",\"data\":\"de279baf-ce7b-41d1-a94f-
e3266600201e\",\"fail_if_unknown\":false}]},\"challenge\":\"htPAbDDG2W7Y6e-
gQ9VjPKWxky5LTAxdKuy8KxPJC81LPg93MVWZn-
iGfhdQhepu0gjbHdl67SNwdZljCg6QcQ\",\"username\":\"jeff\",\"policy\":{\"accepted
\":[[{\"userVerification\":1023,\"authenticationAlgorithms\":[1,2,3,4,5,6,7,8,9
],\"assertionSchemes\":[\"UAFV1TLV\"]}]],\"disallowed\":[{\"aaid\":[\"ABCD#1234
\"]}]}}]",
  "statusCode" : 1200,
  "op" : "Reg"
}
```

Note that nevisFIDO includes a proprietary extension in the `header` part of the `RegistrationRequest` object. This extension provides the session ID that can be used to retrieve the registration status. The following JSON snippet represents the session ID extension. Refer to Extension and Proprietary Extensions for details.

```
{
  "id" : "ch.nevis.auth.fido.uaf.sessionid",
  "data" : "d61e461e-c597-4ed3-9d71-12d1c0e3556c",
  "fail_if_unknown" : false
}
```

9.2.9. HTTP Status Codes

The following HTTP status codes are returned by the Registration Request Service:

Table 19. HTTP status codes - Registration Request Service

HTTP Code	Description
200	**OK** The server processed the request successfully. A `ReturnUAFRequest` JSON object containing a `RegistrationRequest` object is returned.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/fido+uaf`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/fido+uaf;charset=UTF-8`.

9.3. Registration Response Service

The FIDO client calls the Registration Response Service when it has processed the `RegistrationRequest` object coming from the FIDO server. The client invokes the Registration Response Service by providing a `RegistrationResponse` object that must be processed by nevisFIDO. This chapter describes the request and response messages between the FIDO client and server when calling the Registration Response Service.

The client can send to nevisFIDO the ASM Status Code received by the ASM in the Context. In case of error, the Registration Response will contain no assertions, but the rest of its contents must be specification compliant. This is not part of the standard specification and can be used notably together with the Status Service.

9.3.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/uaf/1.1/registration
```

9.3.2. HTTP Methods

`POST` is the only supported HTTP method.

9.3.3. Request Headers

The following request headers are mandatory:

Table 20. Mandatory request headers - Registration Response Service

Name	Description
`Accept`	Accept header, **must** be `application/fido+uaf`.
`Content-Type`	Content type header, **must** be `application/fido+uaf;charset=UTF-8`.

9.3.4. Request Body

The Registration Response Service requires a JSON payload with a `SendUAFResponse` object as defined in the FIDO UAF HTTP Transport Specification. The `SendUAFResponse` object has the following structure:

Table 21. SendUAFResponse object - Registration Response Service

Attribute	Type	Description	Optional
`context`	`String`	The contextual information must be a stringified JSON object that conforms to the Context dictionary. It contains the status code returned by the ASM (Authenticator Specific Module) in the authenticating device.	true
`uafResponse`	`String`	The **stringified JSON** containing the registration response sent by the UAF client.	false

The `uafResponse` part of the `SendUAFResponse` object consists of an array with a *single* `RegistrationResponse` object as defined in the FIDO UAF Protocol Specification. The `RegistrationResponse` object contains a registration response for a specific protocol version.

The *RegistrationResponse* dictionary includes all attributes that define a `RegistrationResponse` object. The following table describes the *RegistrationResponse* dictionary:

Table 22. RegistrationResponse dictionary

Attribute	Type	Description	Optional
`header`	`Object`	See OperationHeader dictionary. The operation header of the `uafResponse`.	false
`fcParams`	`String`	Base64url-encoded final challenge parameters using UTF-8 encoding, see FinalChallengeParams dictionary.	false
`assertions`	`Array`	Array of required assertion response data for each authenticator being registered.	true. The assertions can be empty if an error occurred and the client provides the ASM status code in the Context.
`assertions[].assertionScheme`	`String`	Name of the assertion scheme used to encode the assertion. The only currently supported mandatory assertion scheme is: `UAFV1TLV`.	false
`assertions[].assertion`	`String`	Base64url-encoded TAG_UAFV1_REG_ASSERTION object, contains the newly generated `UAuth.pub` key and is signed by the attestation's private key.	false
`assertions[].tcDisplayPNGCharacteristics`	`String`	Supported transaction PNG type for the definition of the `DisplayPNGCharacteristicsDescriptor`.	true
`assertions[].exts`	`Array`	Extensions prepared by the authenticator.	true

If an error occurs on the client side, the client may return a RegistrationResponse containing an empty assertions array. In this case, the server will respond with UAF status code "1498" ("Unacceptable Content"). This is intended behavior: The response is not valid according to the specification.

9.3.4.1. Context

The client can provide the ASM Status Code and the Client Error Code in the context. These codes are exposed by the Status Service. The Status Service uses the codes to better identify problems on the client side, which occur during the authentication with the FIDO authenticators.

The client can also *optionally* provide information required to create a Dispatch Target Service together with the to-be-registered FIDO UAF credentials. This prevents the client from having to authenticate twice in order to create a dispatch target. This is particularly useful in the out-of-band scenarios (with the Create Token Service and the Dispatch Token Service).

Table 23. Context dictionary

Attribute	Type	Description	Optional
`asmStatusCode`	`number`	The ASM Status Code	true
`clientErrorCode`	`number`	The Client Error Code	true
`dispatchTarget`	`Object`	The dispatch target information.	true

Attribute	Type	Description	Optional
`deviceId`	`String`	The string identifying the device, such as a mobile phone, where the dispatch target and the FIDO UAF credentials are stored. The goal of this attribute is to allow administration tools to link the FIDO UAF credentials and the dispatch targets. This identifier should not change during the whole lifetime of the device. If a `deviceId` and a dispatch target are provided, the device identifier will also be used for the created dispatch target.	true

Table 24. Dispatch Target dictionary

Attribute	Type	Description	Optional
`name`	`String`	The name describing the dispatch target. It can be used as a user-friendly representation that helps the end-user to identify this target. It must be unique for all the dispatch targets defined for the user.	false
`dispatcher`	`String`	The name of the default dispatcher as configured in nevisFIDO that must be associated with this dispatch target. This value corresponds to the value of the `type` attribute in the *nevisFIDO* YAML configuration. If the client does not provide the dispatcher to be used in the dispatch token request, this is the dispatcher that will be invoked. This attribute is deprecated and will be ignored in future releases.	true
`target`	`String`	The information required by the dispatcher to dispatch a token. Currently, this is only required when using the FCM dispatcher: it is the Firebase push registration token that nevisFIDO uses to send a push notification. If the FCM dispatcher is not used (i.e. if no push notifications are required), this attribute can be omitted when creating the dispatch target.	true
`signatureKey`	`Object`	The public key that is used by *nevisFIDO* to verify the signature of the messages sent by the client to modify the dispatch target. It must be provided as a JWS object as described in the JSON Web Key (JWK) Format. The `use` attribute of the JWS must be set to `sig` and the `key_ops` attribute must contain the value `sign`.	true
`encryptionKey`	`Object`	The public key used by *nevisFIDO* to encrypt the tokens sent to the dispatch target. It must be provided as a JWS object as described in the JSON Web Key (JWK) Format. The `use` attribute of the JWS must be set to `enc` and the `key_ops` attribute must contain the value `encrypt`.	true

Suppose the following scenario: The client provides a Dispatch Target Service in the context, the FIDO UAF contents are successfully validated, but it is not possible to create the dispatch target. In this case, the server will return UAF status code "1498" ("Unacceptable Content"). Furthermore, also no FIDO UAF credential will be created, as the whole operation is handled atomically.

9.3.5. Response Headers

The following response headers will be set:

Table 25. Response headers - Registration Response Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.3.6. Response Body

The Registration Response Service will return a `ServerResponse` object as described in FIDO UAF HTTP Transport Specification.

The `ServerResponse` object has the following structure:

Table 26. ServerResponse object - Registration Response Service

Path	Type	Description
`statusCode`	`Number`	UAF status code for the operation, see UAF Status Codes. If the client provided a `dispatchTarget` in the `SendUAFResponse` context, the validation of the FIDO UAF contents was successful, but the dispatch target could not be created, nevisFIDO will return a UAF status code `UNACCEPTABLE_CONTENT` (1498).
`description`	`String`	Detailed message containing a user-friendly description of the registration result and the dispatch target ID (if one was created). This is the stringified representation of the JSON described in the Registration Server Response dictionary.
`additionalTokens`	`Array`	New authentication or authorization token(s) for the client that are not natively handled by HTTP transport.
`location`	`String`	If present, indicates to the client web application that it should navigate to the URI contained in this field.
`postData`	`String`	If present, and in combination with `location`, indicates the client should POST the contents to the specified location.

Path	Type	Description
`newUAFRequest`	`String`	Server may return a new UAF protocol message with this field. This might be used to supply a fresh request to retry an operation in response to a transient failure, to request additional confirmation for a transaction, or to send a deregistration message in response to a permanent failure.

The following attributes are ignored and not handled by nevisFIDO:

- `additionalTokens`
- `location`
- `postData`
- `newUAFRequest`

Table 27. Registration Server Response dictionary

Attribute	Type	Description	Optional
`message`	`String`	A user-friendly description of the result of the registration operation.	true
`dispatchTargetId`	`String`	The identifier of the dispatch target (if created).	true

9.3.7. Example Request

```
POST /nevisfido/uaf/1.1/registration HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 3626

{
  "uafResponse" :
"[{\"header\":{\"serverData\":\"serverdata\",\"upv\":{\"major\":1,\"minor\":1},
\"op\":\"Reg\",\"appID\":\"appID\"},\"fcParams\":\"eyJjaGFsbGVuZ2UiOiJjaGFsbGVu
Z2UiLCJjaGFubmVsQmluZGluZyI6eyJjaWRfcHVia2V5IjoicHVia2V5In0sImFwcElEIjoiYXBwSUQ
iLCJmYWNldElEIjoiZmFjZXRJRCJ9\",\"assertions\":[{\"assertionScheme\":\"UAFV1TLV
\",\"assertion\":\"AT5TAAM-
PgALLgkAQUJCQiM5OTk5Di4HAAEAAQMAAAEKLgAACS4FAGtleUlkDS4IAAEAAAAFAAAADC4JAHB1Ymx
pY0tleQc-DQAGLgkAc2lnbmF0dXJl\"}]}]",
  "context" :
"{\"asmStatusCode\":0,\"clientErrorCode\":0,\"dispatchTarget\":{\"name\":\"My
Mobile
Phone\",\"target\":\"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P198aDPO\",\"dispa
tcher\":\"firebase-cloud-
messaging\",\"signatureKey\":{\"kty\":\"RSA\",\"x5t#S256\":\"6vzaxXiwrgpnFxMu4T
M6ITNhlv4R4SCE9qQfN0Nt5ro\",\"e\":\"AQAB\",\"use\":\"sig\",\"kid\":\"1271366980
875316127\",\"x5c\":[\"MIICujCCAaKgAwIBAgIIEaTNiHo7658wDQYJKoZIhvcNAQELBQAwHTEL
MAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMzOVoXDTI0MDgwMzE1MTMzOVo
wHTELMAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCg
KCAQEAz2azSusth1jEKLpvRyTmSG/vNKlMmANu1+0GY5STs8pFGgD0q8Ey37zmA8+xkassO0S5DOfo0
YptZ/kw70Vbr5SK+bX0RmaYETR0Lx6GUS88bZGX8Z9/DmBAvTG1clZUL+ftOLbRB3meTCOPSvLdKBhB
3FGg7XnkFC74p7a/zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVFXRc1J3yGgeYGqvEb2NYJMR
4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvRyXUtjmFNVxMnv+0fS9uEFI
```

```
Ap3fBLl2fIjXaV/dz+WcCbEXN1cvKGfQIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQCF4SSmNCTa+fnoN
SzR5xOqLf+5g4Ofuxmon5KEppFFoKEIgxlbc8FQxOLNuBsLvNNhzODimgp//PE8w++KihlZA5m/76ue
ZP40OZCGKEuaVvCYgIY5F/yVUeoJBau2Js2MOBFnEWjS6zK4xiXEBWTrveBjTsQNrP6ReInUAVVtEAN
dl776+tGS//vkVSb1nQumMAz1b5q4PtP8wbD3JyY12tV0jtDNYaEOj6m7fQL+WZkfnbQS77OnZvUrE8
NNcULC8t32yv+qCMdC/8mL6dbP/widzWwvPOdjdMgO+hn+TbAf0DtjFvpXA37vyCBKLwU3f284+hjPZ
7NmQARvTpUH\"],\"n\":\"z2azSusth1jEKLpvRyTmSG_vNKlMmANu1-
0GY5STs8pFGgD0q8Ey37zmA8-xkassO0S5DOfo0YptZ_kw70Vbr5SK-
bX0RmaYETR0Lx6GUS88bZGX8Z9_DmBAvTG1clZUL-
ftOLbRB3meTCOPSvLdKBhB3FGg7XnkFC74p7a_zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVF
XRc1J3yGgeYGqvEb2NYJMR4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvR
yXUtjmFNVxMnv-0fS9uEFIAp3fBLl2fIjXaV_dz-
WcCbEXN1cvKGfQ\"},\"encryptionKey\":{\"kty\":\"RSA\",\"x5t#S256\":\"6vzaxXiwrgp
nFxMu4TM6ITNhlv4R4SCE9qQfN0Nt5ro\",\"e\":\"AQAB\",\"use\":\"enc\",\"kid\":\"127
1366980875316127\",\"x5c\":[\"MIICujCCAaKgAwIBAgIIEaTNiHo7658wDQYJKoZIhvcNAQELB
QAwHTELMAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMzOVoXDTI0MDgwMzE1
MTMzOVowHTELMAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8
AMIIBCgKCAQEAz2azSusth1jEKLpvRyTmSG/vNKlMmANu1+0GY5STs8pFGgD0q8Ey37zmA8+xkassO0
S5DOfo0YptZ/kw70Vbr5SK+bX0RmaYETR0Lx6GUS88bZGX8Z9/DmBAvTG1clZUL+ftOLbRB3meTCOPS
vLdKBhB3FGg7XnkFC74p7a/zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVFXRc1J3yGgeYGqvE
b2NYJMR4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvRyXUtjmFNVxMnv+0
fS9uEFIAp3fBLl2fIjXaV/dz+WcCbEXN1cvKGfQIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQCF4SSmNC
Ta+fnoNSzR5xOqLf+5g4Ofuxmon5KEppFFoKEIgxlbc8FQxOLNuBsLvNNhzODimgp//PE8w++KihlZA
5m/76ueZP40OZCGKEuaVvCYgIY5F/yVUeoJBau2Js2MOBFnEWjS6zK4xiXEBWTrveBjTsQNrP6ReInU
AVVtEANdl776+tGS//vkVSb1nQumMAz1b5q4PtP8wbD3JyY12tV0jtDNYaEOj6m7fQL+WZkfnbQS77O
nZvUrE8NNcULC8t32yv+qCMdC/8mL6dbP/widzWwvPOdjdMgO+hn+TbAf0DtjFvpXA37vyCBKLwU3f2
84+hjPZ7NmQARvTpUH\"],\"n\":\"z2azSusth1jEKLpvRyTmSG_vNKlMmANu1-
0GY5STs8pFGgD0q8Ey37zmA8-xkassO0S5DOfo0YptZ_kw70Vbr5SK-
bX0RmaYETR0Lx6GUS88bZGX8Z9_DmBAvTG1clZUL-
ftOLbRB3meTCOPSvLdKBhB3FGg7XnkFC74p7a_zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVF
XRc1J3yGgeYGqvEb2NYJMR4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvR
yXUtjmFNVxMnv-0fS9uEFIAp3fBLl2fIjXaV_dz-WcCbEXN1cvKGfQ\"}}}"
}
```


cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/registration' -i -X POST \
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "uafResponse" :
"[{\"header\":{\"serverData\":\"serverdata\",\"upv\":{\"major\":1,\"minor\":1},
\"op\":\"Reg\",\"appID\":\"appID\"},\"fcParams\":\"eyJjaGFsbGVuZ2UiOiJjaGFsbGVu
Z2UiLCJjaGFubmVsQmluZGluZyI6eyJjaWRfcHVia2V5IjoicHVia2V5In0sImFwcElEIjoiYXBwSUQ
iLCJmYWNldElEIjoiZmFjZXRJRCJ9\",\"assertions\":[{\"assertionScheme\":\"UAFV1TLV
\",\"assertion\":\"AT5TAAM-
PgALLgkAQUJCQiM5OTk5Di4HAAEAAQMAAAEKLgAACS4FAGtleUlkDS4IAAEAAAAFAAAADC4JAHB1Ymx
pY0tleQc-DQAGLgkAc2lnbmF0dXJl\"}]}]",
  "context" :
"{\"asmStatusCode\":0,\"clientErrorCode\":0,\"dispatchTarget\":{\"name\":\"My
Mobile
Phone\",\"target\":\"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P198aDPO\",\"dispa
tcher\":\"firebase-cloud-
messaging\",\"signatureKey\":{\"kty\":\"RSA\",\"x5t#S256\":\"6vzaxXiwrgpnFxMu4T
M6ITNhlv4R4SCE9qQfN0Nt5ro\",\"e\":\"AQAB\",\"use\":\"sig\",\"kid\":\"1271366980
875316127\",\"x5c\":[\"MIICujCCAaKgAwIBAgIIEaTNiHo7658wDQYJKoZIhvcNAQELBQAwHTEL
MAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMzOVoXDTI0MDgwMzE1MTMzOVo
wHTELMAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCg
KCAQEAz2azSusth1jEKLpvRyTmSG/vNKlMmANu1+0GY5STs8pFGgD0q8Ey37zmA8+xkassO0S5DOfo0
YptZ/kw70Vbr5SK+bX0RmaYETR0Lx6GUS88bZGX8Z9/DmBAvTG1clZUL+ftOLbRB3meTCOPSvLdKBhB
3FGg7XnkFC74p7a/zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVFXRc1J3yGgeYGqvEb2NYJMR
4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvRyXUtjmFNVxMnv+0fS9uEFI
Ap3fBLl2fIjXaV/dz+WcCbEXN1cvKGfQIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQCF4SSmNCTa+fnoN
SzR5xOqLf+5g4Ofuxmon5KEppFFoKEIgxlbc8FQxOLNuBsLvNNhzODimgp//PE8w++KihlZA5m/76ue
ZP40OZCGKEuaVvCYgIY5F/yVUeoJBau2Js2MOBFnEWjS6zK4xiXEBWTrveBjTsQNrP6ReInUAVVtEAN
dl776+tGS//vkVSb1nQumMAz1b5q4PtP8wbD3JyY12tV0jtDNYaEOj6m7fQL+WZkfnbQS77OnZvUrE8
NNcULC8t32yv+qCMdC/8mL6dbP/widzWwvPOdjdMgO+hn+TbAf0DtjFvpXA37vyCBKLwU3f284+hjPZ
7NmQARvTpUH\"],\"n\":\"z2azSusth1jEKLpvRyTmSG_vNKlMmANu1-
0GY5STs8pFGgD0q8Ey37zmA8-xkassO0S5DOfo0YptZ_kw70Vbr5SK-
bX0RmaYETR0Lx6GUS88bZGX8Z9_DmBAvTG1clZUL-
ftOLbRB3meTCOPSvLdKBhB3FGg7XnkFC74p7a_zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVF
XRc1J3yGgeYGqvEb2NYJMR4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvR
yXUtjmFNVxMnv-0fS9uEFIAp3fBLl2fIjXaV_dz-
WcCbEXN1cvKGfQ\"},\"encryptionKey\":{\"kty\":\"RSA\",\"x5t#S256\":\"6vzaxXiwrgp
nFxMu4TM6ITNhlv4R4SCE9qQfN0Nt5ro\",\"e\":\"AQAB\",\"use\":\"enc\",\"kid\":\"127
1366980875316127\",\"x5c\":[\"MIICujCCAaKgAwIBAgIIEaTNiHo7658wDQYJKoZIhvcNAQELB
QAwHTELMAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMzOVoXDTI0MDgwMzE1
MTMzOVowHTELMAkGA1UEBhMCY2gxDjAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8
AMIIBCgKCAQEAz2azSusth1jEKLpvRyTmSG/vNKlMmANu1+0GY5STs8pFGgD0q8Ey37zmA8+xkassO0
S5DOfo0YptZ/kw70Vbr5SK+bX0RmaYETR0Lx6GUS88bZGX8Z9/DmBAvTG1clZUL+ftOLbRB3meTCOPS
vLdKBhB3FGg7XnkFC74p7a/zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVFXRc1J3yGgeYGqvE
b2NYJMR4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvRyXUtjmFNVxMnv+0
fS9uEFIAp3fBLl2fIjXaV/dz+WcCbEXN1cvKGfQIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQCF4SSmNC
Ta+fnoNSzR5xOqLf+5g4Ofuxmon5KEppFFoKEIgxlbc8FQxOLNuBsLvNNhzODimgp//PE8w++KihlZA
5m/76ueZP40OZCGKEuaVvCYgIY5F/yVUeoJBau2Js2MOBFnEWjS6zK4xiXEBWTrveBjTsQNrP6ReInU
AVVtEANdl776+tGS//vkVSb1nQumMAz1b5q4PtP8wbD3JyY12tV0jtDNYaEOj6m7fQL+WZkfnbQS77O
nZvUrE8NNcULC8t32yv+qCMdC/8mL6dbP/widzWwvPOdjdMgO+hn+TbAf0DtjFvpXA37vyCBKLwU3f2
84+hjPZ7NmQARvTpUH\"],\"n\":\"z2azSusth1jEKLpvRyTmSG_vNKlMmANu1-
0GY5STs8pFGgD0q8Ey37zmA8-xkassO0S5DOfo0YptZ_kw70Vbr5SK-
bX0RmaYETR0Lx6GUS88bZGX8Z9_DmBAvTG1clZUL-
ftOLbRB3meTCOPSvLdKBhB3FGg7XnkFC74p7a_zvFjQ0yEPEPOX9LdoYXUZ2cxtN9aGxJXM7cfqjVVF
XRc1J3yGgeYGqvEb2NYJMR4zlC9yCsPlHu1tj25OohtxanPUf6V1a0LdzwoWaorrsmO7jdMwAtyiWvR
yXUtjmFNVxMnv-0fS9uEFIAp3fBLl2fIjXaV_dz-WcCbEXN1cvKGfQ\"}}}"
}'
```

9.3.8. Example Response

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:39 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 226

{
  "statusCode" : 1200,
  "description" : "{\"message\":\"OK. Operation completed. A device with ID
a83dc0d7-bc97-435f-a260-cae5fa91fd84 has been
registered.\",\"dispatchTargetId\":\"a83dc0d7-bc97-435f-a260-cae5fa91fd84\"}"
}
```

9.3.9. HTTP Status Codes

The following HTTP status codes are returned by the Registration Response Service:

Table 28. HTTP status codes - Registration Response Service

HTTP Code	Description
200	**OK** The server processed the request successfully. Check the response body for UAF specific status information.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to "application/fido+uaf".
415	**Unsupported media type** The `Content-Type` header is not properly set to "application/fido+uaf;charset=UTF-8".

9.4. Authentication Request Service

By calling the Authentication Request Service and requesting an `AuthenticationRequest` object from the FIDO server, the user/FIDO client initiates the FIDO authentication process. The Authentication Request Service is also used for transaction confirmation (see the FIDO UAF Protocol Specification). The client must send the relevant transactions in the `context` part of the request message body. For more information, see Context and Transaction Confirmation.
This chapter describes the request and response messages between the FIDO client and Server when calling the Authentication Request Service.

9.4.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/uaf/1.1/request/authentication
```

9.4.2. HTTP Methods

`POST` is the only supported HTTP method.

9.4.3. Request Headers

The following request headers are mandatory:

Table 29. Mandatory requests headers - Authentication Request Service

Name	Description
`Accept`	Accept header, **must** be `application/fido+uaf`.
`Content-Type`	Content type header, **must** be `application/fido+uaf;charset=UTF-8`.

9.4.4. Request Body

The Authentication Request Service requires from the FIDO client a JSON payload with a `GetUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `GetUAFRequest` object has the following structure:

Table 30. GetUAFRequest object - Authentication Request Service

Attribute	Type	Description	Optional
`op`	`String`	The request operation, must be set to `Auth`.	false
`previousRequest`	`String`	If the application is requesting a new UAF request message because the previous one expired, the previous one could be sent to the server.	true
`context`	`String`	The contextual information must be a **stringified JSON** object that conforms to the Context dictionary.	false

The `previousRequest` parameter is ignored and not handled by nevisFIDO.

9.4.4.1. Context

As part of the authentication operation, the client can provide the username of the user to be authenticated to the Authentication Request Service. One possibility is to provide the username inside the `context` part of the `GetUAFRequest` object. See the Authorization section of the Reference Guide for more information.

When the username is provided, nevisFIDO will assume that the authentication is a step-up authentication. This has certain implications in the way nevisFIDO handles the request:

1. If the username is not known (i.e. no credential was registered for the provided username), nevisFIDO will return an error response.
2. If the username is known (i.e. at least one credential was registered for the provided username), nevisFIDO will return a policy that only contains the AAID and KeyIDs registered for the user that match the policy configured. See the "Configuration" section of the Reference Guide describing the policy for more information.

The client can also include transaction information in the `context` part of the `GetUAFRequest` object, in the form of a `TransactionContent` object. See Transaction Confirmation for more information.

The *Context* dictionary below lists all attributes that can be included in the `context` part of the `GetUAFRequest` object.

Table 31. Context dictionary

Attribute	Type	Description	Optional
`username`	`String`	Identity information regarding the user on whose behalf the FIDO client is operating. In the case of the `idm` credential repository, the accepted type of username (`loginId`, `email`, etc.) depends on how the username mapper of the credential repository is configured.	false
`transaction`	`TransactionContent[]` (see Transaction Content dictionary)	The transaction information sent from the client to the FIDO Server. See Transaction Confirmation for more information.	true

9.4.4.2. Transaction Confirmation

In the context of transaction confirmation, nevisFIDO expects the client to provide the transaction content in the `context` part of the `GetUAFRequest`, in the form of a `TransactionContent` object. The *Transaction Content* dictionary below describes the attributes that define the `TransactionContent` object.

The Transaction Content dictionary is part of the Authentication Request Service, but not of the official FIDO UAF 1.1 Specification.

Table 32. Transaction Content dictionary

Attribute	Type	Description	Optional
`contentType`	`String`	Contains the MIME content type supported by the authenticator according to its metadata statement. Note that nevisFIDO only supports the values "text/plain" or "image/png".	false
`content`	`String`	Contains the (base64url-encoded) transaction content according to the setting of the `contentType` attribute.	false

nevisFIDO uses the transaction content information in the `TransactionContent` objects as input for the generation of `transaction` objects, which are included in the `AuthenticationRequests` sent from the FIDO server to the FIDO UAF client. However, before nevisFIDO sends back the `transaction` objects, it validates the information coming from the FIDO client, with the following outcome:

If no username was provided in the initial `GetUafRequest`:

- nevisFIDO only supports `plain/text` content type and will return the content as provided in the *Transaction Content*.

If a username was provided in the initial `GetUafRequest` (i.e. step-up authentication):

- If a `TransactionContent` object contains a content type that is not supported by any of the registered credentials for the user, no transaction will be included in the `AuthenticationRequest`. nevisFIDO uses the metadata statements associated with the registered credentials to determine if the content type is supported.
- If a `TransactionContent` object contains the content type `image/png`, nevisFIDO will look for the `DisplayPngCharacteristicsDescriptor` provided during registration (if any). If the descriptor was not provided during registration, nevisFIDO will look for descriptors in the metadata, and add the retrieved descriptors to the returned transaction in the `AuthenticationRequest`. If no display PNG descriptor is found, the transaction will be discarded. Note that if several display PNG descriptors are found, nevisFIDO will generate several `transaction` objects.

Below, see the *Transaction* dictionary as described in the FIDO UAF Protocol Specification. The *Transaction* dictionary

includes the attributes that define a `transaction` object. For more information about the `AuthenticationRequest` object, see Response Body further on.

Table 33. Transaction dictionary

Attribute	Type	Description	Optional
`contentType`	`String`	Contains the MIME content type supported by the authenticator according to its metadata statement. Note that this version of nevisFIDO only supports the values "text/plain" or "image/png".	false
`content`	`String`	Contains the (base64url-encoded) transaction content according to the setting of the `contentType` attribute.	false
`tcDisplayPNGCharacteristics`	`DisplayPNGCharacteristicsDescriptor`	Defines the PNG characteristics of the transaction content and sets the attributes of the *DisplayPNGCharacteristicsDescriptor* (such as `width`, `height` or `colorType`). Click here for a description of the *DisplayPNGCharacteristicsDescriptor* dictionary.	false (if `contentType` is "image/png")

9.4.4.3. Example

The following `GetUAFRequest` contains a PNG image transaction:

```
{
  "context" : "{\"username\":\"jeff\",\"transaction\":[{\"contentType\":
\"image/png\",\"content\":\"dGhpcyBpcyBzdXBwb3NlZCB0byBiZSBhIHBuZyBpbWFnZQ\"}]}
",
  "op" : "Auth"
}
```

9.4.5. Response Headers

The following response headers will be set:

Table 34. Response headers - Authentication Request Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.4.6. Response Body

The Authentication Request Service returns a JSON body with a `ReturnUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `ReturnUAFRequest` object has the following structure:

Table 35. ReturnUAFRequest object - Authentication Request Service

Path	Type	Description
`statusCode`	`Number`	UAF status code for the operation.
`uafRequest`	`String`	The new UAF request message if the server decides to issue one.

Path	Type	Description
`op`	`String`	Hint to the client regarding the operation type of the message, must be set to `Auth`.
`lifetimeMillis`	`Number`	Hint informing the client application of the lifetime of the message in milliseconds. Absent if the operation was not successful.

The `uafRequest` part of the `ReturnUAFRequest` object contains the `AuthenticationRequest` object. The *AuthenticationRequest* dictionary includes the attributes that define a `AuthenticationRequest` object. The following table describes the *AuthenticationRequest* dictionary:

Table 36. AuthenticationRequest dictionary

Attribute	Type	Description	Optional
`header`	`OperationHeader`	The `header` defines the operation header of the UAF messages coming from the Authentication Request Service. Within the `header`, the operation header attributes are specified. For a description of these attributes, see OperationHeader dictionary. Note that the attribute `header.op` must be set to "Auth".	false
`challenge`	`ServerChallenge`	Server-provided challenge value.	false
`transaction`	`Transaction[]`	Transaction data to be explicitly confirmed by the user. This data is contained in `transaction` objects. For more information about the structure of `transaction` objects, see the Transaction dictionary.	true
`policy`	`Policy`	Describes which types of authenticators are acceptable for this authentication operation.	false

9.4.7. Example Request

```
POST /nevisfido/uaf/1.1/request/authentication HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 187

{
  "context" :
"{\"username\":\"jeff\",\"transaction\":[{\"contentType\":\"text/plain\",\"cont
ent\":\"Q29uZmlybSB5b3VyIHB1cmNoYXNlIGZvciBhIHZhbHVlIG9mIENIRjIwMC4\"}]}",
  "op" : "Auth"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/request/authentication' -i -X
POST \
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "context" :
"{\"username\":\"jeff\",\"transaction\":[{\"contentType\":\"text/plain\",\"cont
ent\":\"Q29uZmlybSB5b3VyIHB1cmNoYXNlIGZvciBhIHZhbHVlIG9mIENIRjIwMC4\"}]}",
  "op" : "Auth"
}'
```

9.4.8. Example Response

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:21 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 813

{
  "lifetimeMillis" : 120000,
  "uafRequest" : "[{\"header\":{\"serverData\":\"YxHI-
E0SLDf7uH_GvQB10Nb50oB8f8GBiBFO5KiyK0lZfGmoz_QH_jueHmMdBOSF2XIq5Be-
UcQAEJ4O33XK9Q\",\"upv\":{\"major\":1,\"minor\":1},\"op\":\"Auth\",\"appID\":\"
https://www.siven.ch/appID\",\"exts\":[{\"id\":\"ch.nevis.auth.fido.uaf.session
id\",\"data\":\"a0b84187-28e5-4672-92f7-
f4cbbb598568\",\"fail_if_unknown\":false}]},\"challenge\":\"QRh3CZwG8t7xGKBetB7
PmpX8rFr9AgThKbWR58pSy6nFWh9-
AH1yYokrjZlSURqPGQ0Kd6DU16JUAAu5c1bVRA\",\"policy\":{\"accepted\":[[{\"userVeri
fication\":1023,\"authenticationAlgorithms\":[1,2,3,4,5,6,7,8,9],\"assertionSch
emes\":[\"UAFV1TLV\"]}]]},\"transaction\":[{\"contentType\":\"text/plain\",\"co
ntent\":\"Q29uZmlybSB5b3VyIHB1cmNoYXNlIGZvciBhIHZhbHVlIG9mIENIRjIwMC4\"}]}]",
  "statusCode" : 1200,
  "op" : "Auth"
}
```

nevisFIDO includes a proprietary extension in the `header` part of the `AuthenticationRequest` object. This extension provides the session ID that can be used to retrieve the authentication status. The following JSON snippet represents the session ID extension. Refer to Extension and Proprietary Extensions for details.

```
{
  "id" : "ch.nevis.auth.fido.uaf.sessionid",
  "data" : "d61e461e-c597-4ed3-9d71-12d1c0e3556c",
  "fail_if_unknown" : false
}
```

9.4.9. HTTP Status Codes - Authentication Request Service

The following HTTP status codes are returned by the Authentication Request Service:

Table 37. HTTP status codes - Authentication Request Service

HTTP Code	Description
200	**OK** The server processed the request successfully. A `ReturnUAFRequest` JSON object containing an `AuthenticationRequest` is returned.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to "application/fido+uaf".
415	**Unsupported Media Type** The `Content-Type` header is not properly set to "application/fido+uaf;charset=UTF-8".

9.5. Authentication Response Service

The FIDO client calls the Authentication Response Service when it has processed the `AuthenticationRequest` object coming from the FIDO server. The client invokes the Authentication Response Service by providing an `AuthenticationResponse` object that must be processed by nevisFIDO. This chapter describes the request and response messages between the FIDO client and Server when calling the Authentication Response Service.

The client can send to nevisFIDO the ASM Status Code received by the ASM in the Context. In case of error, the Authentication Response will contain no assertions, but the rest of its contents must be specification compliant. This is not part of the standard specification and can be used notably in conjunction with the Status Service.

9.5.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/uaf/1.1/authentication
```

9.5.2. HTTP Methods

`POST` is the only supported HTTP method.

9.5.3. Request Headers

The following request headers are mandatory:

Table 38. Mandatory request headers - Authentication Response Service

Name	Description
`Accept`	Accept header, **must** be `application/fido+uaf`.
`Content-Type`	Content type header, **must** be `application/fido+uaf;charset=UTF-8`.

9.5.4. Request Body

The Authentication Response Service requires a JSON payload with a `SendUAFResponse` object as defined in the FIDO UAF HTTP Transport Specification. The `SendUAFResponse` object has the following structure:

Table 39. SendUAFResponse object - Authentication Response Service

Attribute	Type	Description	Optional
`context`	`String`	The contextual information must be a stringified JSON object that conforms to the Context dictionary. It contains the status code returned by the ASM (Authenticator Specific Module) in the authenticating device.	true
`uafResponse`	`String`	The **stringified JSON array** with a single element. The element is the authentication response sent by the UAF client.	false

The `uafResponse` part of the `SendUAFResponse` object consists of an array with a *single* `AuthenticationResponse` object as defined in the FIDO UAF Protocol Specification. The *AuthenticationResponse* dictionary includes the attributes that define an `AuthenticationResponse` object. The following table describes the *AuthenticationResponse* dictionary:

Table 40. AuthenticationResponse dictionary

Attribute	Type	Description	Optional
`header`	`Object`	See OperationHeader dictionary. The operation header of the `uafResponse`.	false
`fcParams`	`String`	Base64url-encoded final challenge parameters using UTF-8 encoding, see FinalChallengeParams dictionary.	false
`assertions`	`Array`	Array of required authenticator responses related to the authentication.	true. The assertions can be empty if an error occurred and the client provides the ASM status code in the Context.
`assertions[].assertionScheme`	`String`	Name of the assertion scheme used to encode the assertion. The only currently supported mandatory assertion scheme is: `UAFV1TLV`.	false
`assertions[].assertion`	`String`	Base64url-encoded TAG_UAFV1_AUTH_ASSERTION object, contains the assertion signed by the attestation's private key.	false
`assertions[].exts`	`Array`	Extensions prepared by the authenticator.	true

If an AuthenticationResponse containing an empty assertions array is sent back to the server indicating a client error, the server will respond with UAF status code 1498 (Unacceptable Content). This is intended behaviour because the response is not "valid" according to the specification.

9.5.4.1. Context

The client can provide the ASM Status Code and the Client Error Code in the context. These codes are exposed by the Status Service and can be used to better identify what was the problem that occurred in the client side while authenticating with the FIDO authenticator.

Table 41. Context dictionary

Attribute	Type	Description	Optional
`asmStatusCode`	`number`	The ASM Status Code	true
`clientErrorCode`	`number`	The Client Error Code	true

9.5.5. Response Headers

The following response headers will be set:

Table 42. Response headers - Authentication Response Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.5.6. Response Body

The Authentication Response Service will return a `ServerResponse` object as described in FIDO UAF HTTP Transport Specification. The `ServerResponse` object has the following structure:

Table 43. ServerResponse object - Authentication Response Service

Path	Type	Description
`statusCode`	`Number`	UAF status code for the authentication operation, see UAF Status Codes.
`description`	`String`	Detailed message containing a user-friendly description of the authentication result. This is the stringified representation of the JSON described in the Authentication Server Response Description dictionary.
`additionalTokens`	`Array`	New authentication or authorization token(s) for the client that are not natively handled by HTTP transport.
`location`	`String`	If present, indicates to the client web application that it should navigate to the URI contained in this field.
`postData`	`String`	If present, and in combination with `location`, indicates the client should POST the contents to the specified location.

Path	Type	Description
`newUAFRequest`	`String`	Server may return a new UAF protocol message with this field. This might be used to supply a fresh request to retry an operation in response to a transient failure, to request additional confirmation for a transaction, or to send a deauthentication message in response to a permanent failure.

Table 44. Authentication Server Response Description dictionary

Attribute	Type	Description	Optional
`message`	`String`	A user-friendly description of the result of the authentication operation.	true

The following attributes are ignored and not handled:

- `additionalTokens`
- `location`
- `postData`
- `newUAFRequest`

9.5.7. Example Request

```
POST /nevisfido/uaf/1.1/authentication HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 572

{
  "uafResponse" :
"[{\"header\":{\"serverData\":\"serverdata\",\"upv\":{\"major\":1,\"minor\":1},
\"op\":\"Auth\",\"appID\":\"appID\"},\"fcParams\":\"eyJjaGFsbGVuZ2UiOiJjaGFsbGV
uZ2UiLCJjaGFubmVsQmluZGluZyI6eyJjaWRfcHVia2V5IjoicHVia2V5In0sImFwcElEIjoiYXBwSU
QiLCJmYWNldElEIjoiZmFjZXRJRCJ9\",\"assertions\":[{\"assertionScheme\":\"UAFV1TL
V\",\"assertion\":\"Aj5pAAQ-
WAALLgkAQUJDRCNBQkNEDi4FAAEAAQkADy4SAGF1dGhlbnRpY2F0b3JOb25jZQouBABoYXNoEC4PAHR
yYW5zYWN0aW9uSGFzaAkuBQBrZXlJZA0uBAABAAAABi4JAHNpZ25hdHVyZQ\"}]}]",
  "context" : "{\"asmStatusCode\":0,\"clientErrorCode\":0}"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/authentication' -i -X POST \
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "uafResponse" :
"[{\"header\":{\"serverData\":\"serverdata\",\"upv\":{\"major\":1,\"minor\":1},
\"op\":\"Auth\",\"appID\":\"appID\"},\"fcParams\":\"eyJjaGFsbGVuZ2UiOiJjaGFsbGV
uZ2UiLCJjaGFubmVsQmluZGluZyI6eyJjaWRfcHVia2V5IjoicHVia2V5In0sImFwcElEIjoiYXBwSU
QiLCJmYWNldElEIjoiZmFjZXRJRCJ9\",\"assertions\":[{\"assertionScheme\":\"UAFV1TL
V\",\"assertion\":\"Aj5pAAQ-
WAALLgkAQUJDRCNBQkNEDi4FAAEAAQkADy4SAGF1dGhlbnRpY2F0b3JOb25jZQouBABoYXNoEC4PAHR
yYW5zYWN0aW9uSGFzaAkuBQBrZXlJZA0uBAABAAAABi4JAHNpZ25hdHVyZQ\"}]}]",
  "context" : "{\"asmStatusCode\":0,\"clientErrorCode\":0}"
}'
```

9.5.8. Example Response

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:45 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 89

{
  "statusCode" : 1200,
  "description" : "{\"message\":\"OK. Operation completed.\"}"
}
```

9.5.9. HTTP Status Codes

The following HTTP status codes are returned by the Authentication Response Service:

Table 45. HTTP status codes - Authentication Response Service

HTTP Code	Description
200	**OK** The server processed the request successfully. Check the response body for UAF specific status information.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to "application/fido+uaf".
415	**Unsupported Media Type** The `Content-Type` header is not properly set to "application/fido+uaf;charset=UTF-8".

9.6. Deregistration Request Service

By calling the Deregistration Request Service and requesting a `DeregistrationRequest` object from the FIDO server, the user/FIDO client initiates the FIDO deregistration process. This chapter describes the request and response messages between the FIDO client and Server when calling the Deregistration Request Service.

It is recommended to protect this service using a SecToken. See Authorization for more information.

9.6.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/uaf/1.1/request/deregistration
```

9.6.2. HTTP Methods

`POST` is the only supported HTTP method.

9.6.3. Request Headers

The following request headers are mandatory:

Table 46. Mandatory request headers - Deregistration Request Service

Name	Description
`Accept`	Accept header, **must** be `application/fido+uaf`.
`Content-Type`	Content type header, **must** be `application/fido+uaf;charset=UTF-8`.

9.6.4. Request Body

The Deregistration Request Service requires from the FIDO client a JSON payload with a `GetUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `GetUAFRequest` object has the following structure:

Table 47. GetUAFRequest object - Deregistration Request Service

Attribute	Type	Description	Optional
`op`	`String`	The request operation, must be set to `Dereg`.	false
`previousRequest`	`String`	If the application is requesting a new UAF request message because the previous one expired, the previous one could be sent to the server.	true
`context`	`String`	The contextual information must be a **stringified JSON** object that conforms to the Context dictionary.	false

The `previousRequest` parameter is ignored and not handled

9.6.4.1. Context

The client must provide the credentials to be deregistered for a specific user in the `context` part of the `GetUAFRequest` object. The following three options are available and can be configured via the `mode` attribute.

- Deregister all credentials associated with the user. The `mode` attribute value is `username`.
- Deregister all credentials associated with the user and with any of the provided AAIDs. The `mode` attribute value is `aaid`.
- Deregister all credentials associated with the user and with any of the provided tuples of AAID and Key ID. The `mode` attribute value is `aaid_and_keyid`.

The *Context* dictionary below lists all attributes that can be included in the `context` part of the `GetUAFRequest` object.

Table 48. Context dictionary

Attribute	Type	Description	Optional
`mode`	`String`	The deregistration mode. The value is either `username`, `aaid` or `aaid_and_keyid`.	false
`aaid`	`String[]`	The `AAID`s of the credentials to be deregistered.	false (if the deregistration `mode` is `aaid`)
`aaid_and_keyid`	`AaidAndKeyId[]` (see AAID and Key ID dictionary for details)	The `AAID` and `KeyID` tuples of the credentials to be deregistered.	false (if the deregistration `mode` is `aaid_and_keyid`)
`username`	`String`	Identity information regarding the user on whose behalf the FIDO client is operating. In the case of the `idm` credential repository, the accepted type of username (`loginId`, `email`, etc.) depends on how the username mapper of the credential repository is configured.	false

9.6.4.2. AAID And Key ID Dictionary

Table 49. AAID and Key ID dictionary

Attribute	Type	Description	Optional
`aaid`	`String`	The AAID of the credential to be deregistered.	false
`keyid`	`String`	The base64url-encoded value of the key ID of the credential to be deregistered.	false

9.6.5. Response Headers

The following response headers will be set:

Table 50. Response headers - Deregistration Request Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.6.6. Response Body

The Deregistration Request Service returns a JSON body with a `ReturnUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `ReturnUAFRequest` object has the following structure:

Table 51. ReturnUAFRequest object - Deregistration Request Service

Path	Type	Description
`statusCode`	`Number`	UAF status code for the operation.
`uafRequest`	`String`	The new UAF request message if the server decides to issue one.

Path	Type	Description
`op`	`String`	Hint to the client regarding the operation type of the message, must be set to `Dereg`.

The `uafRequest` part of the `ReturnUAFRequest` object contains the `DeregistrationRequest` object. The *DeregistrationRequest* dictionary includes the attributes that define a `DeregistrationRequest` object. The following table describes the *DeregistrationRequest* dictionary:

Table 52. DeregistrationRequest dictionary

Attribute	Type	Description	Optional
`header`	`OperationHeader`	The `header` defines the operation header of the UAF messages coming from the Deregistration Request Service. Within the *header*, the operation header attributes are specified. For a description of these attributes, see OperationHeader dictionary. Note that the attribute `header.op` must be set to "Dereg".	false
`authenticators`	`DeregisterAuthenticator[]`	List of authenticators to be deregistered.	false

nevisFIDO follows the Deregistration Request Generation Rules for FIDO Server: It will return empty strings in the `authenticators` attribute of the `DeregistrationRequest` when the `aaid` or the `username` deregistration modes are used.

9.6.7. Example Request Using aaid_and_keyid Mode

```
POST /nevisfido/uaf/1.1/request/deregistration HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 168

{
  "context" :
"{\"username\":\"jeff\",\"mode\":\"aaid_and_keyid\",\"aaid_and_keyid\":[{\"aaid
\":\"1234#ABCD\",\"keyID\":\"a2V5SWRJbkJhc2U2NA\"}]}",
  "op" : "Dereg"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/request/deregistration' -i -X
POST \
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "context" :
"{\"username\":\"jeff\",\"mode\":\"aaid_and_keyid\",\"aaid_and_keyid\":[{\"aaid
\":\"1234#ABCD\",\"keyID\":\"a2V5SWRJbkJhc2U2NA\"}]}",
  "op" : "Dereg"
}'
```

9.6.8. Example Response Using aaid_and_keyid Mode

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:21 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 384

{
  "statusCode" : 1200,
  "uafRequest" :
"[{\"header\":{\"upv\":{\"major\":1,\"minor\":1},\"op\":\"Dereg\",\"appID\":\"h
ttps://www.siven.ch/appID\",\"exts\":[{\"id\":\"ch.nevis.auth.fido.uaf.sessioni
d\",\"data\":\"3d248b64-ca5c-4f01-b486-
0158ccfe3f6f\",\"fail_if_unknown\":false}]},\"authenticators\":[{\"aaid\":\"123
4#ABCD\",\"keyID\":\"a2V5SWRJbkJhc2U2NA\"}]}]",
  "op" : "Dereg"
}
```

9.6.9. Example Request Using username Mode

```
POST /nevisfido/uaf/1.1/request/deregistration HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 83

{
  "context" : "{\"username\":\"jeff\",\"mode\":\"username\"}",
  "op" : "Dereg"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/request/deregistration' -i -X
POST \
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "context" : "{\"username\":\"jeff\",\"mode\":\"username\"}",
  "op" : "Dereg"
}'
```

9.6.10. Example Response Using username Mode

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:22 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 357

{
  "statusCode" : 1200,
  "uafRequest" :
"[{\"header\":{\"upv\":{\"major\":1,\"minor\":1},\"op\":\"Dereg\",\"appID\":\"h
ttps://www.siven.ch/appID\",\"exts\":[{\"id\":\"ch.nevis.auth.fido.uaf.sessioni
d\",\"data\":\"f3d9971e-0b71-4c3c-937d-
56d90e4f5913\",\"fail_if_unknown\":false}]},\"authenticators\":[{\"aaid\":\"\",
\"keyID\":\"\"}]}]",
  "op" : "Dereg"
}
```

9.6.11. HTTP Status Codes

The following HTTP status codes are returned by the Deregistration Request Service:

Table 53. HTTP status codes - Deregistration Request Service

HTTP Code	Description
200	**OK** The server processed the request successfully. A `ReturnUAFRequest` JSON object containing a `DeregistrationRequest` object is returned.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/fido+uaf`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/fido+uaf;charset=UTF-8`.

9.7. Facets Service

This section describes the FIDO UAF Facets Service. This public HTTP API is concerned with *application facets*. According to the offical FIDO documentation, the concept of an *Application Facet* is used to describe the identities of a single logical application across various platforms. For example, the application MyBank may have an Android app, an iOS app, and a Web app accessible from a browser. These three apps are all *facets* of the MyBank application.

The FIDO client calls the Facets Service to check whether a certain facet is trusted or not. Based on the list of trusted facet IDs in the response, the client evaluates whether to proceed with or abort its operation.

See the FIDO AppID and Facet Specification for details.

9.7.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/uaf/1.1/facets
```

9.7.2. HTTP Methods

"GET" is the only supported HTTP method.

9.7.3. Request Headers

No request headers must be set.

9.7.4. Response Headers

The following response headers will be set:

Table 54. Response headers - Facets Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido.trusted-apps+json`.

9.7.5. Response Body

The Facets Service returns a `TrustedFacets` object as described in FIDO AppID and Facet Specification. The `TrustedFacets` object has the following structure:

Table 55. TrustedFacets objects - Facets Service

Path	Type	Description
`trustedFacets`	`Array`	Array of TrustedFacets dictionaries.
`trustedFacets[].version`	`Object`	Dictionary containing the UAF protocol version.
`trustedFacets[].ids`	`Array`	An array of URLs identifying authorized facets for this appID.

The `version` attribute in the `TrustedFacets` object refers to the version of the UAF protocol. See the *Version* dictionary below for more details:

Table 56. Version dictionary - Facets Service

Path	Type	Description
`major`	`Number`	Major UAF protocol version.
`minor`	`Number`	Minor UAF protocol version.

The currently supported protocol version is:

- `major`: *"1"*
- `minor`: *"1"*

9.7.6. Example Request Using GET

```
GET /nevisfido/uaf/1.1/facets HTTP/1.1
Host: fido.siven.ch
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/facets' -i -X GET
```

9.7.7. Example Response Using GET

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:14 GMT
Content-Type: application/fido.trusted-apps+json
Transfer-Encoding: chunked
Content-Length: 306

{
  "trustedFacets" : [ {
    "version" : {
      "major" : 1,
      "minor" : 1
    },
    "ids" : [ "https://register.siven.ch", "https://fido.siven.ch",
"http://www.siven.ch", "http://www.muvonda.com", "https://www.siven.ch:444",
"android:apk-key-hash:324234234", "ios:bundle-id:my.ios.bundle" ]
  } ]
}
```

9.7.8. Example Request Using Unsupported Method

```
POST /nevisfido/uaf/1.1/facets HTTP/1.1
Content-Type: application/x-www-form-urlencoded; charset=ISO-8859-1
Host: fido.siven.ch
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/uaf/1.1/facets' -i -X POST \
    -H 'Content-Type: application/x-www-form-urlencoded; charset=ISO-8859-1'
```

9.7.9. Example Response Using Unsupported Method

```
HTTP/1.1 405 Method Not Allowed
Allow: GET
Cache-Control: must-revalidate,no-cache,no-store
```

9.7.10. HTTP Status Codes

The following HTTP status codes are returned by the Facets Service:

Table 57. HTTP status codes - Facets Service

HTTP Code	Description
200	**OK** The server processed the request successfully. Check the response body for UAF specific status information.

HTTP Code	Description
405	**Method Not Allowed** The method of the received request was not “GET”.
406	**Not Acceptable** The provided `Accept` header forbids "application/fido.trusted-apps+json" content.

9.8. Out-of-Band Services

This chapter describes services intended to be used in out-of-band scenarios. Because the FIDO specification does not include such scenarios, the services described in this chapter are **not standard FIDO services** but proprietary nevisFIDO functionality.

The available out-of-band services are:

- Dispatch Target Service: This service manages dispatch target entities. You must register the target entities as dispatch targets a priori, to be able to involve out-of-band clients in an operation.
- Dispatch Token Service: This service generates tokens and dispatches them to the dispatch targets. Use the Redeem Token Service to redeem such a token later on. The redemption of the token triggers a standard FIDO operation.
- Redeem Token Service: Use this service to redeem tokens previously generated and dispatched by the Dispatch Token Service. Redeeming a token triggers a FIDO operation, executed by the client redeeming the token.
- Create Token Service: Use this service to create tokens that will be redeemed later.

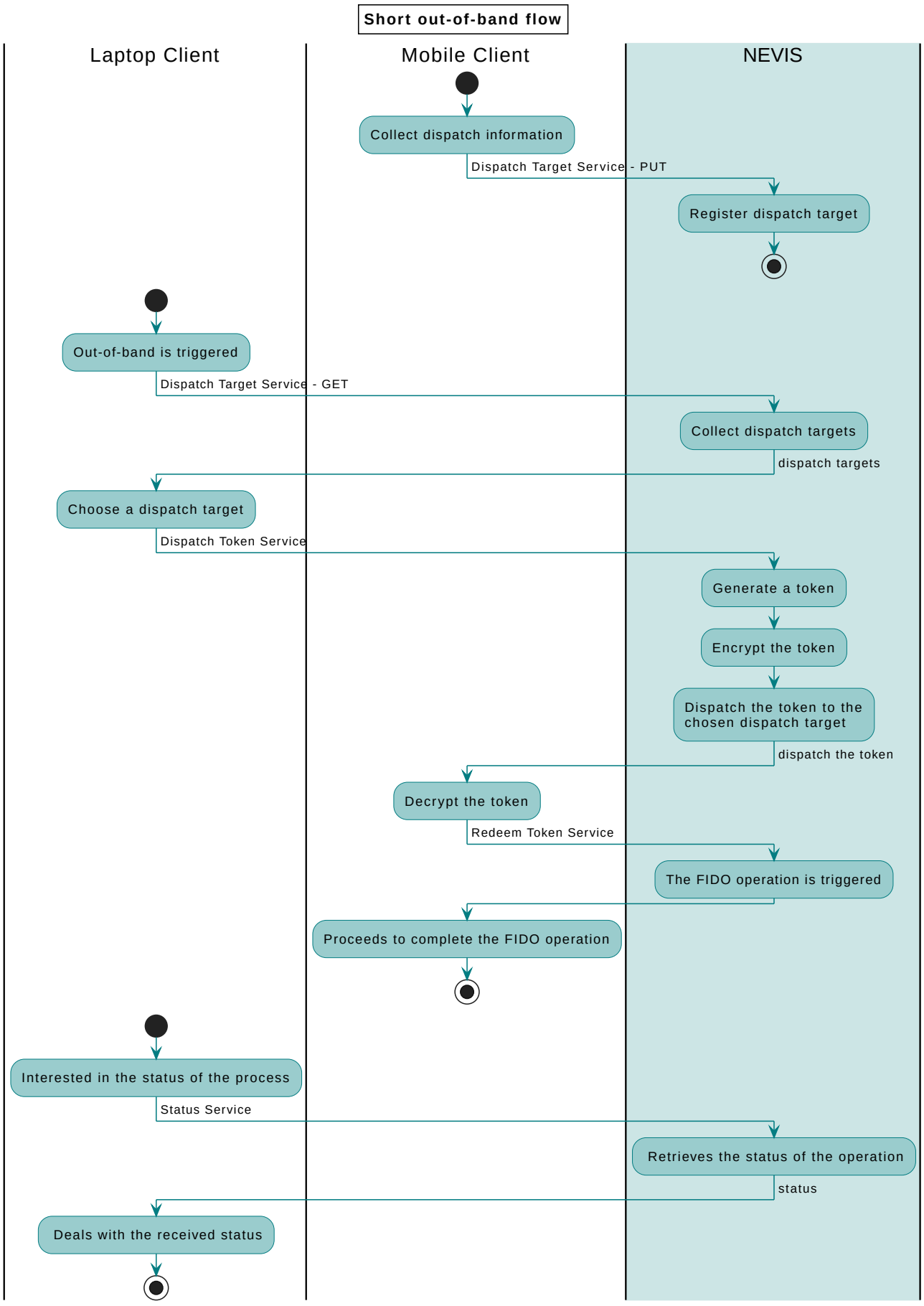
Do not use the Create Token Service for out-of-band scenarios. Instead, use the Dispatch Token Service.

In *out-of-band* scenarios, always access the services in the following (pseudo) order:

1. Dispatch Target Service: Create a dispatch target.
2. Dispatch Target Service: Query dispatch targets.
3. Dispatch Token Service: Generate and dispatch a token to a dispatch target.
4. Redeem Token Service: Redeem a token (that has been dispatched).

After the redemption of the token, a standard FIDO operation is triggered. The flow continues according to the FIDO specification.

The following diagram shows how the above-mentioned services should be used together.



1. The Mobile Client registers itself as a dispatch target.
2. The Laptop Client queries dispatch targets.
3. The Laptop Client chooses a desired dispatch target.

4. The Laptop Client requests a dispatch to the chosen dispatch target.
5. NEVIS generates a token.
6. NEVIS encrypts the token.
7. NEVIS dispatches the token to the chosen dispatch target.
8. The Mobile Client receives the dispatched token.
9. The Mobile Client decrypts the token.
10. The Mobile Client redeems the token.
11. NEVIS triggers a FIDO operation based on the token.
12. The Mobile Client proceeds to complete the FIDO operation.
13. The Laptop Client monitors the status of the operation.

9.8.1. Dispatch Target Service

This chapter describes the Dispatch Target Service. The Dispatch Target Service is **not a standard FIDO service** but a proprietary nevisFIDO functionality. The Dispatch Target Service is a public HTTP API with which you manage dispatch targets in nevisFIDO.

A dispatch target is a destination to which nevisFIDO can dispatch a token. For example, nevisFIDO may send push notifications with the registration token to an application in a mobile device. Here, the mobile device is the dispatch target. The dispatch target can also be an email address if the tokens are sent via an email server.

The Dispatch Target Service consists of four parts or endpoints: the Create, Modify, Delete and Query Dispatch Target.

9.8.1.1. Create Dispatch Target

This section describes the Create part of the Dispatch Target Service.

9.8.1.1.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/token/dispatch/targets
```

9.8.1.1.2. HTTP Methods

`POST` is the only supported HTTP method.

9.8.1.1.3. Request Headers

The following request headers are mandatory:

Table 58. Mandatory request headers - Dispatch Target Service / Create Dispatch Target part

Name	Description
`Accept`	Accept header, **must** be `application/json`.
`Content-Type`	Content type header, **must** be `application/json`.

9.8.1.1.4. Request Body

The Create Dispatch Target Service requires a JSON payload with the following structure:

Table 59. Request body - Dispatch Target Service / Create Dispatch Target part

Attribute	Type	Description	Optional
`name`	`String`	The name describing the dispatch target. It can be used as a user-friendly representation that helps the end-user to identify this target. It must be unique for all the dispatch targets defined for the user.	false
`deviceId`	`String`	The String identifying the device (for instance a mobile phone) where the dispatch target and the FIDO UAF credentials are stored. The goal of this attribute is to allow administration tools to link the FIDO UAF credentials and the dispatch targets. This identifier should not change during the whole lifetime of the device.	true
`dispatcher`	`String`	The name of the default dispatcher as configured in nevisFIDO that must be associated with this dispatch target. This value corresponds to the value of the `type` attribute in the *nevisFIDO* YAML configuration. If the client does not provide the dispatcher to be used in the dispatch token request, this is the dispatcher that will be invoked. This attribute is deprecated and will be ignored in future releases.	true
`target`	`String`	The information required by the dispatcher to dispatch a token. Currently, this is only required when using the FCM dispatcher: it is the Firebase push registration token that nevisFIDO uses to send a push notification. If the FCM dispatcher is not used (i.e. if no push notifications are required), this attribute can be omitted when creating the dispatch target.	true
`signatureKey`	`Object`	The public key that is used by *nevisFIDO* to verify the signature of the messages sent by the client to modify the dispatch target. It must be provided as a JWS object as described in the JSON Web Key (JWK) Format. The `use` attribute of the JWS must be set to `sig` and the `key_ops` attribute must contain the value `sign`.	false
`encryptionKey`	`Object`	The public key used by *nevisFIDO* to encrypt the tokens sent to the dispatch target. It must be provided as a JWS object as described in the JSON Web Key (JWK) Format. The `use` attribute of the JWS must be set to `enc` and the `key_ops` attribute must contain the value `encrypt`.	true
`username`	`String`	Identity information of the user whose dispatch target will be created. In the case of the `idm` credential repository, the accepted type of username (`loginId`, `email` or `extId`) depends on how the username mapper of the dispatch target repository is configured.	true

9.8.1.1.5. Response Headers

The following response headers will be set:

Table 60. Response headers - Dispatch Target Service / Create Dispatch Target part

Name	Description
`Content-Type`	Content type header, fixed to `application/json`.

9.8.1.1.6. Response Body

The body of the response message coming from the Create Dispatch Target Service contains the identifier of the created dispatch target. If the dispatch creation was successful, the HTTP status code is “201”. The table below lists all elements of the response body.

Table 61. Response body - Dispatch Target Service / Create Dispatch Target part

Path	Type	Description
`id`	`String`	The identifier of the created dispatch target. This identifier is immutable and must be used by the client to update and delete the dispatch target. It must also be used to select the dispatch target to which the generated tokens must be sent. This identifier is to be used by *nevisFIDO* and its format is not related to the type of the dispatcher.

9.8.1.1.7. Example Request

```
POST /nevisfido/token/dispatch/targets HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: fido.siven.ch
Content-Length: 3209

{
  "name" : "My Mobile Phone",
  "target" : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P198aDPO",
  "signatureKey" : {
    "kty" : "RSA",
    "x5t#S256" : "cCrhkZ3Q5kspKpZTjUfqTT89PZKI4EW7_4QtfniYiBk",
    "e" : "AQAB",
    "use" : "sig",
    "kid" : "7852318338397317936",
    "x5c" : [
"MIICujCCAaKgAwIBAgIIbPkJ7Od1KzAwDQYJKoZIhvcNAQELBQAwHTELMAkGA1UEBhMCY2gxDjAMBg
NVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMyM1oXDTI0MDgwMzE1MTMyM1owHTELMAkGA1UEBhMCY2gxD
jAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoi42hm1FJNZQVzeK
868l2la8zkaVwSFfvTbm1mJBbkHzV2bJ2eNmUsfSNSbHrCMuVH3iBy6m9lfNcJB5buEGPA+8PFfIPJP
elbxVijXYco2CIRvuCPLGb3n1wCEftMKW0Fw946qA9EXNZj9BYRTSPqgBxYlCfenBpztxtBkxKibsXc
OCKdKwBua6kvKwhtgobdZhSIxc5Bb+ZUhJ/jEEE5mTiRA9XlDCtT1cw4N3lnrQuvCJBz9eJpLeT2trB
U0IaeIdkavKhijYmY5KlDNqnXPRbP5jSiFqiG/FmRe2dGsZgpmfihAnxDxxCkyZ+36OqdEBhyaSHWqx
aYz/UBGRuwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQBuSX0zeGZe9JwhsjYo1XQF0GSnNIEQRsFgfw1
JYP0qiWnYTPJK42ZatSLaFBB4le8hVGZSiI8fTVDTjLKyQZL2lLqu2KR9eSlA8G+E47R3R4mFYcOA65
XFCMS1SV+BijaqGnRnfMFLMmvmd4RKk49n7fuaEdHBA2rPaTgsjYhn4rFtmxa3Dp4CNYqNeJLRAuZaQ
zPxk4zlPeLLoXb+oag4mlKlnI0GE1ULX4RWDXaDy1awzsSFtGuhEJpiO+AJy/No7i+0sif/s/J+4Ul4
MrrDE9W+RPckrBuiSx/9XGd7+wk5yzwxpnCY+KAKFQ1Hd1Y+yRfRP+uLYGYjV1H2+1ml" ],
    "n" :
```

```
"oi42hm1FJNZQVzeK868l2la8zkaVwSFfvTbm1mJBbkHzV2bJ2eNmUsfSNSbHrCMuVH3iBy6m9lfNcJ
B5buEGPA-
8PFfIPJPelbxVijXYco2CIRvuCPLGb3n1wCEftMKW0Fw946qA9EXNZj9BYRTSPqgBxYlCfenBpztxtB
kxKibsXcOCKdKwBua6kvKwhtgobdZhSIxc5Bb-
ZUhJ_jEEE5mTiRA9XlDCtT1cw4N3lnrQuvCJBz9eJpLeT2trBU0IaeIdkavKhijYmY5KlDNqnXPRbP5
jSiFqiG_FmRe2dGsZgpmfihAnxDxxCkyZ-36OqdEBhyaSHWqxaYz_UBGRuw"
  },
  "encryptionKey" : {
    "kty" : "RSA",
    "x5t#S256" : "z6iXW9YfMv3NdDf_L9JjMTct_FyJzLiI2eAj3BcTJf8",
    "e" : "AQAB",
    "use" : "enc",
    "kid" : "17100409740410464738",
    "x5c" : [
"MIICuzCCAaOgAwIBAgIJAO1Q3A6ay6niMA0GCSqGSIb3DQEBCwUAMB0xCzAJBgNVBAYTAmNoMQ4wDA
YDVQQDEwVzaXZlbjAeFw0yMjA4MDIxNTEzMjNaFw0yNDA4MDMxNTEzMjNaMB0xCzAJBgNVBAYTAmNoM
Q4wDAYDVQQDEwVzaXZlbjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMkHHUi8AR4esykb
o7YaAjAhuN20zFj2iN+VxwMfFVJ1H6OKUtAa3+qzibDBz+93gt0obyp0FckfEHYs9Hg4qBkbYzw7ccR
ed4nnYa8mVmGXJRaxrFkpqHdQi3LAiJOZ4LZAI2jeneVN85H+tj2MYeoIn7c+Xw02HutVWnNVcWrrGS
3NHrTctwBIEzsuB3MnflJFkUtcjkINSeDCDYoaC93Jqt2jyOFqx/kS0ubQU5yc169N3vYwo70JdmDFw
0k7HhhLzqegmdNSQ845uuo6FNSg0LG/xVkauseUkeN1+oZB0m33cT3P2Y8/dV29xOvNkgWQp8qBKGPU
brZUXq3grc0CAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAvJlqfe/NfAr5DtU1wnmSaYevEAXMjbD5zcR
ISDFxoLXtSqj0raD3siDoT52AgHY/l9gr4pGgnxy4d62EMRX9MGTjssv9VqUnkcGhQc9QXu52QCQZlm
ppko0MJhmD8twdx1k7dDI88TxFn+yVPaWDsKBsb2XLrVMP1OFeLe4eTbrfVejG0dgG5eTGZbNUcVxbj
pOhHkBDIt8du6qVVEJ5beNNnbA4mcdZq6mpeebGtRzzCoxrz9wZK1e8I42O0eTvsBEDQ6jCNSkUjV1q
qtN25Xbc1275KHE2J3DDlc5LQeWWQEnbLu6Oq1xHMvtpIYeESVrUqHezFQ9V4kOBWRwhZw==" ],
    "n" :
"yQcdSLwBHh6zKRujthoCMCG43bTMWPaI35XHAx8VUnUfo4pS0Brf6rOJsMHP73eC3ShvKnQVyR8Qdi
z0eDioGRtjPDtxxF53iedhryZWYZclFrGsWSmod1CLcsCIk5ngtkAjaN6d5U3zkf62PYxh6giftz5fD
TYe61Vac1VxausZLc0etNy3AEgTOy4Hcyd-UkWRS1yOQg1J4MINihoL3cmq3aPI4WrH-
RLS5tBTnJzXr03e9jCjvQl2YMXDSTseGEvOp6CZ01JDzjm66joU1KDQsb_FWRq6x5SR43X6hkHSbfdx
Pc_Zjz91Xb3E682SBZCnyoEoY9RutlRereCtzQ"
  },
  "username" : "username",
  "deviceId" : "Acme Inc Phone. Serial Number Hash:
e14c2cec1f8c448a47874b5e164df11727a9e0ad"
}
```


cURL:


```
$ curl 'https://fido.siven.ch/nevisfido/token/dispatch/targets' -i -X POST \
    -H 'Accept: application/json' \
    -H 'Content-Type: application/json' \
    -d '{
  "name" : "My Mobile Phone",
  "target" : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P198aDPO",
  "signatureKey" : {
    "kty" : "RSA",
    "x5t#S256" : "cCrhkZ3Q5kspKpZTjUfqTT89PZKI4EW7_4QtfniYiBk",
    "e" : "AQAB",
    "use" : "sig",
    "kid" : "7852318338397317936",
    "x5c" : [
"MIICujCCAaKgAwIBAgIIbPkJ7Od1KzAwDQYJKoZIhvcNAQELBQAwHTELMAkGA1UEBhMCY2gxDjAMBg
NVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMyM1oXDTI0MDgwMzE1MTMyM1owHTELMAkGA1UEBhMCY2gxD
jAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoi42hm1FJNZQVzeK
868l2la8zkaVwSFfvTbm1mJBbkHzV2bJ2eNmUsfSNSbHrCMuVH3iBy6m9lfNcJB5buEGPA+8PFfIPJP
```

```
elbxVijXYco2CIRvuCPLGb3n1wCEftMKW0Fw946qA9EXNZj9BYRTSPqgBxYlCfenBpztxtBkxKibsXc
OCKdKwBua6kvKwhtgobdZhSIxc5Bb+ZUhJ/jEEE5mTiRA9XlDCtT1cw4N3lnrQuvCJBz9eJpLeT2trB
U0IaeIdkavKhijYmY5KlDNqnXPRbP5jSiFqiG/FmRe2dGsZgpmfihAnxDxxCkyZ+36OqdEBhyaSHWqx
aYz/UBGRuwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQBuSX0zeGZe9JwhsjYo1XQF0GSnNIEQRsFgfw1
JYP0qiWnYTPJK42ZatSLaFBB4le8hVGZSiI8fTVDTjLKyQZL2lLqu2KR9eSlA8G+E47R3R4mFYcOA65
XFCMS1SV+BijaqGnRnfMFLMmvmd4RKk49n7fuaEdHBA2rPaTgsjYhn4rFtmxa3Dp4CNYqNeJLRAuZaQ
zPxk4zlPeLLoXb+oag4mlKlnI0GE1ULX4RWDXaDy1awzsSFtGuhEJpiO+AJy/No7i+0sif/s/J+4Ul4
MrrDE9W+RPckrBuiSx/9XGd7+wk5yzwxpnCY+KAKFQ1Hd1Y+yRfRP+uLYGYjV1H2+1ml" ],
    "n" :
"oi42hm1FJNZQVzeK868l2la8zkaVwSFfvTbm1mJBbkHzV2bJ2eNmUsfSNSbHrCMuVH3iBy6m9lfNcJ
B5buEGPA-
8PFfIPJPelbxVijXYco2CIRvuCPLGb3n1wCEftMKW0Fw946qA9EXNZj9BYRTSPqgBxYlCfenBpztxtB
kxKibsXcOCKdKwBua6kvKwhtgobdZhSIxc5Bb-
ZUhJ_jEEE5mTiRA9XlDCtT1cw4N3lnrQuvCJBz9eJpLeT2trBU0IaeIdkavKhijYmY5KlDNqnXPRbP5
jSiFqiG_FmRe2dGsZgpmfihAnxDxxCkyZ-36OqdEBhyaSHWqxaYz_UBGRuw"
  },
  "encryptionKey" : {
    "kty" : "RSA",
    "x5t#S256" : "z6iXW9YfMv3NdDf_L9JjMTct_FyJzLiI2eAj3BcTJf8",
    "e" : "AQAB",
    "use" : "enc",
    "kid" : "17100409740410464738",
    "x5c" : [
"MIICuzCCAaOgAwIBAgIJAO1Q3A6ay6niMA0GCSqGSIb3DQEBCwUAMB0xCzAJBgNVBAYTAmNoMQ4wDA
YDVQQDEwVzaXZlbjAeFw0yMjA4MDIxNTEzMjNaFw0yNDA4MDMxNTEzMjNaMB0xCzAJBgNVBAYTAmNoM
Q4wDAYDVQQDEwVzaXZlbjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMkHHUi8AR4esykb
o7YaAjAhuN20zFj2iN+VxwMfFVJ1H6OKUtAa3+qzibDBz+93gt0obyp0FckfEHYs9Hg4qBkbYzw7ccR
ed4nnYa8mVmGXJRaxrFkpqHdQi3LAiJOZ4LZAI2jeneVN85H+tj2MYeoIn7c+Xw02HutVWnNVcWrrGS
3NHrTctwBIEzsuB3MnflJFkUtcjkINSeDCDYoaC93Jqt2jyOFqx/kS0ubQU5yc169N3vYwo70JdmDFw
0k7HhhLzqegmdNSQ845uuo6FNSg0LG/xVkauseUkeN1+oZB0m33cT3P2Y8/dV29xOvNkgWQp8qBKGPU
brZUXq3grc0CAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAvJlqfe/NfAr5DtU1wnmSaYevEAXMjbD5zcR
ISDFxoLXtSqj0raD3siDoT52AgHY/l9gr4pGgnxy4d62EMRX9MGTjssv9VqUnkcGhQc9QXu52QCQZlm
ppko0MJhmD8twdx1k7dDI88TxFn+yVPaWDsKBsb2XLrVMP1OFeLe4eTbrfVejG0dgG5eTGZbNUcVxbj
pOhHkBDIt8du6qVVEJ5beNNnbA4mcdZq6mpeebGtRzzCoxrz9wZK1e8I42O0eTvsBEDQ6jCNSkUjV1q
qtN25Xbc1275KHE2J3DDlc5LQeWWQEnbLu6Oq1xHMvtpIYeESVrUqHezFQ9V4kOBWRwhZw==" ],
    "n" :
"yQcdSLwBHh6zKRujthoCMCG43bTMWPaI35XHAx8VUnUfo4pS0Brf6rOJsMHP73eC3ShvKnQVyR8Qdi
z0eDioGRtjPDtxxF53iedhryZWYZclFrGsWSmod1CLcsCIk5ngtkAjaN6d5U3zkf62PYxh6giftz5fD
TYe61Vac1VxausZLc0etNy3AEgTOy4Hcyd-UkWRS1yOQg1J4MINihoL3cmq3aPI4WrH-
RLS5tBTnJzXr03e9jCjvQl2YMXDSTseGEvOp6CZ01JDzjm66joU1KDQsb_FWRq6x5SR43X6hkHSbfdx
Pc_Zjz91Xb3E682SBZCnyoEoY9RutlRereCtzQ"
  },
  "username" : "username",
  "deviceId" : "Acme Inc Phone. Serial Number Hash:
e14c2cec1f8c448a47874b5e164df11727a9e0ad"
}'
```

9.8.1.1.8. Example Response

```
HTTP/1.1 201 Created
Date: Wed, 03 Aug 2022 13:13:24 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 51

{
  "id" : "79f66b85-76dd-4e40-9cb7-7b32f1db305b"
}
```

9.8.1.1.9. HTTP Status Codes

The following HTTP status codes are returned by the Create Dispatch Target part/endpoint of the Dispatch Target Service:

Table 62. HTTP status codes - Dispatch Target Service / Create Dispatch Target part

HTTP Code	Description
201	**Created** The server successfully created the dispatch target.
400	**Bad Request** The provided payload is not properly formatted.
401	**Unauthorized** The request was not authorized. There is an invalid SecToken or unresolved username.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/json`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/json;charset=UTF-8`.
422	**Unprocessable Entity** The request could not be processed. For example, because the name of the provided dispatch target is already in use by another dispatch target of the user, or because the specified dispatcher is not configured.
500	**Internal Server Error** The server could not process the request because of an unexpected error.

9.8.1.2. Modify Dispatch Target

This section describes the Modify part of the Dispatch Target Service.

The modify dispatch target HTTP API is particular regarding the approach used to guarantee that the HTTP client is authorized to perform the operation (i.e. to modify the dispatch target). Instead of using authorization headers (containing notably a `SecToken`) to do the authorization check, the client must send the payload with a signature. The signature is generated with the private signature key of the dispatch target to be modified. nevisFIDO will only process request that are properly signed. This guarantees that only clients possessing the private signature key of the dispatch target are allowed to modify it. This is the reason why the private signature key is assumed to be safely stored by the HTTP client.

9.8.1.2.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/token/dispatch/targets/{id}
```

`id` is the identifier of the dispatch target to be modified.

9.8.1.2.2. HTTP Methods

`PATCH` is the only supported HTTP method.

9.8.1.2.3. Request Headers

The following request headers are mandatory:

Table 63. Mandatory request headers - Dispatch Target Service / Modify Dispatch Target part

Name	Description
`Content-Type`	Content type header, **must** be `application/jose;charset=UTF-8`.

9.8.1.2.4. Request Body

The Modify Dispatch Target Service body is a JWS (JSON Web Signature) using compact serialization. The JWS must be signed with the private key of the dispatch target that is being modified.

Request body - Dispatch Target Service / Modify Dispatch Target part


```
eyJraWQiOiJjZXJ0QWxpYXMiLCJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiTXkgTmV3IE1vYmlsZSBQaG
9uZSIsInRhcmdldCI6ImJrM1JOd1RlM0gwOkNJMmtfSEh3Z0lwb0RLQ0ladnZETUV4VWRGUTNQMTk4Y
URQTyJ9.h4zGAQK8tMm7Y1bw8GSRP6czdt6dfO4kY8EqHWlIMtI8vZXXIo4uzL8dbD_Bn-
rmsHCYPOPQjBGqSLJ8wf4vrnyCEBxIak70V0AhkuAYqTd4uYUwdWYQtGncV6ZuQmzfeNF-
H_Zk3cGfsZA-
s4l89E90vdIX6eHjO2jJida9Sn_R2nq4B8FXd28bDR7b4l0p4SHFcwDr7zoDHjs4xiKtYT8OWguD2Du
Qsaqu5Toiz9vdRfN9XdOguP3vOeRbdxjUfgmbEGBqECfx0wbccpeyJuSPyT6iT5YsvRcenjuQ9RNM_w
VdcCCGyLS957MSpx3nxOrtWxWfOu3nh38GlDAy_A
```


The payload inside the JWS has the following structure:

Table 64. JWS payload structure - Dispatch Target Service / Modify Dispatch Target part

Attribute	Type	Description	Optional
`name`	String	The new name of the dispatch target. Choose a user-friendly name that helps the user to identify this target. The name must be unique for each dispatch target defined for the user.	true
`dispatcher`	String	The name of the default dispatcher as configured in nevisFIDO that must be associated with this dispatch target. This value corresponds to the value of the `type` attribute in the nevisFIDO YAML configuration. If the client does not provide the dispatcher to be used in the dispatch token request, this is the dispatcher that will be invoked.	true

Attribute	Type	Description	Optional
`target`	String	The new information required by the dispatcher to dispatch a token. This information can be a simple identifier (for example, a fcm push identifier or an e-mail address) or more complex data (like a tuple consisting of an e-mail SMTP server and e-mail address). The format of the information depends on the dispatcher implementation. Both JSON and plain text are supported.	true
`signatureKey`	String	The new public key. nevisFIDO uses this key to verify the signature of the messages sent by the client to modify the dispatch target. The key must be provided as a JWS object as described in the JSON Web Key (JWK) Format. Either the `use` or the `key_ops` attribute must be present. If present, the `use` attribute of the JWS must be set to `sig`. If present, the `key_ops` attribute must contain the value `sign`.	true
`encryptionKey`	String	The new public key that is used by nevisFIDO to encrypt the tokens sent to the dispatch target. The key must be provided as a JWS object as described in the JSON Web Key (JWK) Format. Either the `use` or the `key_ops` attribute must be present. If present, the `use` attribute of the JWS must be set to `enc`. If present, the `key_ops` attribute must contain the value `encrypt`.	true

9.8.1.2.5. Response Headers

The response message has no body. Therefore, no headers will be set in the response either.

9.8.1.2.6. Response Body

The body of the response message is empty. Clients must check the returned HTTP status code.

9.8.1.2.7. Example Request

```
PATCH /nevisfido/token/dispatch/targets/d344f32e-379f-482a-8b1f-678db50cb6f0
HTTP/1.1
Content-Type: application/jose;charset=UTF-8
Host: fido.siven.ch
Content-Length: 508

eyJraWQiOiJjZXJ0QWxpYXMiLCJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiTXkgTmV3IE1vYmlsZSBQaG
9uZSIsInRhcmdldCI6ImJrM1JOd1RlM0gwOkNJMmtfSEh3Z0lwb0RLQ0ladnZETUV4VWRGUTNQMTk4Y
URQTyJ9.h4zGAQK8tMm7Y1bw8GSRP6czdt6dfO4kY8EqHWlIMtI8vZXXIo4uzL8dbD_Bn-
rmsHCYPOPQjBGqSLJ8wf4vrnyCEBxIak70V0AhkuAYqTd4uYUwdWYQtGncV6ZuQmzfeNF-
H_Zk3cGfsZA-
s4l89E90vdIX6eHjO2jJida9Sn_R2nq4B8FXd28bDR7b4l0p4SHFcwDr7zoDHjs4xiKtYT8OWguD2Du
Qsaqu5Toiz9vdRfN9XdOguP3vOeRbdxjUfgmbEGBqECfx0wbccpeyJuSPyT6iT5YsvRcenjuQ9RNM_w
VdcCCGyLS957MSpx3nxOrtWxWfOu3nh38GlDAy_A
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/token/dispatch/targets/d344f32e-379f-
482a-8b1f-678db50cb6f0' -i -X PATCH \
    -H 'Content-Type: application/jose;charset=UTF-8' \
    -d
'eyJraWQiOiJjZXJ0QWxpYXMiLCJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiTXkgTmV3IE1vYmlsZSBQa
G9uZSIsInRhcmdldCI6ImJrM1JOd1RlM0gwOkNJMmtfSEh3Z0lwb0RLQ0ladnZETUV4VWRGUTNQMTk4
YURQTyJ9.h4zGAQK8tMm7Y1bw8GSRP6czdt6dfO4kY8EqHWlIMtI8vZXXIo4uzL8dbD_Bn-
rmsHCYPOPQjBGqSLJ8wf4vrnyCEBxIak70V0AhkuAYqTd4uYUwdWYQtGncV6ZuQmzfeNF-
H_Zk3cGfsZA-
s4l89E90vdIX6eHjO2jJida9Sn_R2nq4B8FXd28bDR7b4l0p4SHFcwDr7zoDHjs4xiKtYT8OWguD2Du
Qsaqu5Toiz9vdRfN9XdOguP3vOeRbdxjUfgmbEGBqECfx0wbccpeyJuSPyT6iT5YsvRcenjuQ9RNM_w
VdcCCGyLS957MSpx3nxOrtWxWfOu3nh38GlDAy_A'
```

The payload inside the JWS:

```
{
  "name" : "My New Mobile Phone",
  "target" : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P198aDPO"
}
```

9.8.1.2.8. Example Response

```
HTTP/1.1 204 No Content
Date: Wed, 03 Aug 2022 13:13:31 GMT
```

9.8.1.2.9. HTTP Status Codes

The following HTTP status codes are returned by the Modify Dispatch Target part/endpoint of the Dispatch Target Service:

Table 65. HTTP status codes - Dispatch Target Service / Modify Dispatch Target part

HTTP Code	Description
204	**No Content** The server successfully modified the dispatch target.
400	**Bad Request** The provided payload is not properly formatted.
401	**Unauthorized** The request was not authorized. It was not possible to verify the signature of the request.
404	**Not Found** The provided dispatch target identifier could not be found.
405	**Method Not Allowed** The method of the received request was not "PATCH".
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/jose;charset=UTF-8`.
422	**Unprocessable Entity** The request could not be processed. For example, because the provided name in the dispatch target modification is already in use by another dispatch target of the user, or because the specified dispatcher is not configured.

HTTP Code	Description
500	**Internal Server Error** The server could not process the request because of an unexpected error.

9.8.1.3. Delete Dispatch Target

This section describes the Delete part of the Dispatch Target Service. This endpoint is used by

- Administrators who want to remove dispatch targets of users.
- Users who cannot access the device holding the private key of the dispatch target and therefore want to remove the dispatch target.

9.8.1.3.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/token/dispatch/targets/{id}
```

`id` is the identifier of the dispatch target to be deleted.

9.8.1.3.2. HTTP Methods

`DELETE` is the only supported HTTP method.

9.8.1.3.3. Request Headers

There are no mandatory request headers for the Delete Dispatch Target Service.

9.8.1.3.4. Request Body

The Delete Dispatch Target Service requires no body. Any provided body will be ignored.

9.8.1.3.5. Response Headers

The response message has no body. Therefore, no headers will be set in the response either.

9.8.1.3.6. Response Body

The body of the response message is empty. Clients must check the returned HTTP status code.

9.8.1.3.7. Example Request

```
DELETE /nevisfido/token/dispatch/targets/5dd59467-33ef-4a18-ae7c-ddefe790fc2c
HTTP/1.1
Host: fido.siven.ch
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/token/dispatch/targets/5dd59467-33ef-
4a18-ae7c-ddefe790fc2c' -i -X DELETE
```

9.8.1.3.8. Example Response

```
HTTP/1.1 204 No Content
Date: Wed, 03 Aug 2022 13:13:32 GMT
```

9.8.1.3.9. HTTP Status Codes

The following HTTP status codes are returned by the Delete Dispatch Target part of the Dispatch Target Service:

Table 66. HTTP status codes - Dispatch Target Service / Delete Dispatch Target part

HTTP Code	Description
204	**No Content** The server successfully deleted the dispatch target.
400	**Bad Request** The provided payload is not properly formatted.
401	**Unauthorized** The request was not authorized. There is an invalid SecToken or unresolved username.
403	**Forbidden** The request is forbidden. The user does not have the right to delete the specified dispatch target.
404	**Not Found** The provided dispatch target identifier could not be found.
405	**Method Not Allowed** The method of the received request was not "DELETE".
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/json;charset=UTF-8`.
500	**Internal Server Error** The server could not process the request because of an unexpected error.

9.8.1.4. Query Dispatch Target

This section describes the Query part of the Dispatch Target Service. Use this service to retrieve the dispatch targets for a given user.

9.8.1.4.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/token/dispatch/targets/
```

9.8.1.4.2. HTTP Methods

`GET` is the only supported HTTP method.

9.8.1.4.3. Request Parameters

The following request parameter is mandatory. You must provide it in the request URL.

Table 67. Request Parameters - Dispatch Target Service / Query Dispatch Target part

Name	Description
`username`	Identity information of the user whose dispatch targets will be retrieved. In the case of the `idm` credential repository, the accepted type of username (`loginId`, `email`, etc.) depends on how the username mapper of the dispatch target repository is configured.

9.8.1.4.4. Request Headers

The following request headers are mandatory:

Table 68. Optional request headers - Dispatch Target Service / Query Dispatch Target part

Name	Description
`Accept`	Accept header, **must** be `application/json`.

9.8.1.4.5. Request Body

The Query Dispatch Target Service requires no body. Any provided body will be ignored.

9.8.1.4.6. Response Headers

The following response headers will be set:

Table 69. Response headers - Dispatch Target Service / Querty Dispatch Target part

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.8.1.4.7. Response Body

The body of the response message contains the dispatch targets of the user. The response body is empty if no dispatch targets were found. In this case, an HTTP Not Found (404) status code will be returned.

The table below lists all elements of the response body.

Table 70. Response body - Dispatch Target Service / Create Dispatch Target part

Path	Type	Description
`dispatchTargets`	`Array`	The array containing all the dispatch targets for the provided user name.
`dispatchTargets[].id`	`String`	The identifier of the dispatch target. This identifier is immutable and must be used by the client to update and delete the dispatch target. It must also be used to select the dispatch target to which the generated tokens must be sent. This identifier is to be used by *nevisFIDO* and its format is not related to the type of the dispatcher.
`dispatchTargets[].name`	`String`	The name describing the dispatch target. It can be used as a user-friendly representation that helps the end-user to identify this target. It must be unique for all the dispatch targets defined for the user.

Path	Type	Description
`dispatchTargets[].dispatcher`	`String`	The name of the dispatcher as configured in *nevisFIDO*. This value corresponds to the value of the `type` attribute in the *nevisFIDO* YAML configuration. The dispatcher attribute has been deprecated in the dispatch target. This attribute will not be returned by the Query Dispatch Target Service in future releases.

9.8.1.4.8. Example Request

```
GET /nevisfido/token/dispatch/targets?username=username HTTP/1.1
Accept: application/json
Host: fido.siven.ch
```

cURL:

```
$ curl
'https://fido.siven.ch/nevisfido/token/dispatch/targets?username=username' -i
-X GET \
    -H 'Accept: application/json'
```

9.8.1.4.9. Example Response

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:24 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 117

{
  "dispatchTargets" : [ {
    "id" : "42950470-0209-47f6-85de-ae7ab2826bfd",
    "name" : "My Mobile Phone"
  } ]
}
```

9.8.1.4.10. HTTP Status Codes

The following HTTP status codes are returned by the Query Dispatch Target part/endpoint of the Dispatch Target Service:

Table 71. HTTP status codes - Dispatch Target Service / Query Dispatch Target part

HTTP Code	Description
200	**OK** The server retrieved dispatch targets for the user.

HTTP Code	Description
400	**Bad Request** The provided payload is not properly formatted.
401	**Unauthorized** The request was not authorized. There is an invalid SecToken or unresolved username.
404	**Not Found** No dispatch target could be found.
405	**Method Not Allowed** The method of the received request was not "GET".
406	**Not Acceptable** The `Accept` header is not properly set to `application/json`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/json;charset=UTF-8`.
500	**Internal Server Error** The server could not process the request because of an unexpected error.

9.8.2. Dispatch Token Service

This chapter describes the Dispatch Token Service. The Dispatch Token Service is **not a standard FIDO service** but a proprietary nevisFIDO functionality. It is a public HTTP API with which clients can create and dispatch tokens. These tokens can be redeemed by other trusted parties via the Redeem Token Service.

Depending on the nature of the dispatcher, the client must provide the dispatch target. For instance, in the case of the FCM (Firebase Cloud Messaging) Dispatcher a dispatch target ID must be provided in the request to be able to send the push message.

The client can specify the dispatcher to be used explicitly in the request or implicitly by providing the dispatch target identifier (a default dispatcher is associated with each dispatch target). So, if you want to use a dispatcher with a dispatch target associated by default with another dispatcher, **both** the dispatcher and the dispatch target identifier must be provided. Note that some dispatchers (like the QR Code Dispatcher) do not require a dispatch target.

If a dispatch target is required by the dispatcher, the dispatch target determines the parameters required for the dispatching to take place. Such targets are managed by the Dispatch Target Service. Make sure to create and query targets before you access this service.

Currently dispatching is always done synchronously. This means that the result of the dispatching is always returned immediately in the HTTP response.

9.8.2.1. Base URL

```
https://fido.siven.ch/nevisfido/token/dispatch/<operation>
```

The `<operation>` in the base URL can be `registration`, `authentication` or `deregistration`.

Having split endpoints allows you to protect nevisFIDO differently for each operation, by means of nevisProxy and nevisAuth.
It is necessary to protect the registration and deregistration endpoints. Otherwise, malicious clients could easily initiate unprotected FIDO operations and plant undesired credentials they can take advantage of. You should also protect the authentication dispatch endpoint, if possible. Allowing any form of unauthorized dispatching may

result in malicious parties sending messages directly to target devices and thus to end users.

9.8.2.2. HTTP Methods

`POST` is the only supported HTTP method.

9.8.2.3. Request Headers

The following request headers are mandatory:

Table 72. Mandatory request headers - DispatchRequest

Name	Description
`Accept`	Accept header, **must** be `application/json`.
`Content-Type`	Content type header, **must** be `application/json`.

9.8.2.4. Request Body

The Dispatch Token Service requires from the FIDO client a JSON payload with a `GetUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. This `GetUAFRequest` object is used to initiate the given FIDO operation after the redemption of the token. Also a `ReturnUafRequest` will be returned for the redeemer party.

The request body contains the following elements:

Table 73. DispatchRequest object - Dispatch Token Service

Attribute	Type	Description	Optional
`dispatchTargetId`	`String`	The identifier of the dispatch target. The dispatch target identifier, the dispatcher or both must be provided.	true
`dispatcher`	`String`	The dispatcher that must be used to do the dispatching. The dispatch target identifier, the dispatcher or both must be provided. Currently `firebase-cloud-messaging` (for the FCM dispatcher), `png-qr-code` (for the QR Code dispatcher) and `link` (for the link dispatcher) are supported.	true
`dispatchInformation`	`Object`	Information explicitly intended for the specific dispatcher being used. The content of this object depends on the dispatcher implementation. Both JSON and plain text are supported.	true
`getUafRequest`	`Object`	The GetUafRequest that will be associated with the dispatched token. When the token is redeemed, this request will be used as the base request for which a ReturnUafRequest will be generated.	false

Table 74. GetUafRequest object - Dispatch Token Service

Attribute	Type	Description	Optional
`op`	`String`	The operation requested by the GetUafRequest.	false
`previousRequest`	`String`	If the application is requesting a new UAF request message because the previous one expired, the previous one could be sent to the server.	true

Attribute	Type	Description	Optional
`context`	`String`	The contextual information must be a stringified JSON object that conforms to the Context dictionary.	false

9.8.2.5. Response Headers

The following response headers will be set:

Table 75. Response headers - Dispatch Token Service

Name	Description
`Content-Type`	Content type header, fixed to `application/json`.

9.8.2.6. Response Body

The response body consists of the following elements:

Table 76. Response body - Dispatch Token Service

Path	Type	Description
`dispatchResult`	`String`	The result of the dispatching. Since dispatching is done synchronously, this directly indicates whether a valid dispatcher and dispatch target could be found for the request and whether the token could be transmitted to the dispatcher.
`dispatcherInformation`	`Object`	Information about the dispatcher. Only present when the dispatcher was invoked.
`dispatcherInformation.name`	`String`	The dispatcher that was being used for dispatching.
`dispatcherInformation.response`	`Varies`	Free text that a particular dispatcher returns after it was invoked.

Path	Type	Description
`token`	`String`	The token generated by *nevisFIDO*. This is the token that can be used by a client to trigger the UAF operation (registration, authentication or deregistration). Therefore, the token must be sent to the corresponding endpoint ("/nevisfido/token/redeem/registration" for registration, "/nevisfido/token/redeem/authentication" for authentication and "/nevisfido/token/redeem/deregistration" for deregistration), in order to redeem the token and trigger the FIDO UAF operation with the `GetUAFRequest` sent in this request. It will be returned only if it could be successfully dispatched.
`sessionId`	`String`	The identifier of the session generated by *nevisFIDO*. This session identifier can be used by a client to retrieve the status of the authentication. The session ID must be sent to the "/nevisfido/status" endpoint to get the operation status. It will be returned only if it could be successfully dispatched.

The response message only includes a `dispatcherInformation` object when a dispatcher has been invoked.

The `dispatchResult` attribute of the response message holds information about the result of the dispatching. The attribute can have the following values:

`dispatched`
: A token has been generated and successfully dispatched to the target.

`dispatchError`
: An error occurred when dispatching the token.

`dispatchTargetNotFound`
: nevisFIDO has not found a dispatch target that corresponds to the ID in the request. Make sure to query targets before trying to dispatch a token.

`dispatcherNotFound`
: The target refers to a dispatcher that has no associated dispatcher implementation. It is also possible that the dispatcher implementation has not been loaded successfully. This may point to a configuration error.

`internalError`
: Indicates that an internal problem occurred, for example during the generation of the token.

9.8.2.7. Example Request Using FCM Dispatcher

Refer to Dispatch Token Request Format for additional details.

```
POST /nevisfido/token/dispatch/authentication HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: fido.siven.ch
Content-Length: 432

{
  "dispatchTargetId" : "1c564833-e0c7-470f-8192-e036a1f60c66",
  "dispatcher" : "firebase-cloud-messaging",
  "dispatchInformation" : {
    "notification" : {
      "title" : "Dirk Gently Bank - Confirm the payment"
    },
    "data" : {
      "channelLinking" : {
        "mode" : "visualString",
        "content" : "AB"
      }
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Auth"
  }
}
```

9.8.2.8. Example Response Using FCM Dispatcher

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:31 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 240

{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "5bbeb466-c794-402e-8ea1-086baa1a0379",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "firebase-cloud-messaging",
    "response" : 0
  }
}
```

9.8.2.9. Example Request Using QR Code Dispatcher

```
POST /nevisfido/token/dispatch/authentication HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: fido.siven.ch
Content-Length: 484

{
  "dispatchTargetId" : "1c564833-e0c7-470f-8192-e036a1f60c66",
  "dispatcher" : "png-qr-code",
  "dispatchInformation" : {
    "data" : {
      "attributeName" : "some additional data to be included in the QR code"
    },
    "encodingParameters" : {
      "width" : 300,
      "height" : 300,
      "backgroundColor" : "rgb(255, 255, 255)",
      "foregroundColor" : "rgb(0, 0, 0)"
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Auth"
  }
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/token/dispatch/authentication' -i -X
POST \
    -H 'Accept: application/json' \
    -H 'Content-Type: application/json' \
    -d '{
  "dispatchTargetId" : "1c564833-e0c7-470f-8192-e036a1f60c66",
  "dispatcher" : "png-qr-code",
  "dispatchInformation" : {
    "data" : {
      "attributeName" : "some additional data to be included in the QR code"
    },
    "encodingParameters" : {
      "width" : 300,
      "height" : 300,
      "backgroundColor" : "rgb(255, 255, 255)",
      "foregroundColor" : "rgb(0, 0, 0)"
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Auth"
  }
}'
```

9.8.2.10. Example Response Using QR Code Dispatcher

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:31 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 269

{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "c9fa045e-2d2b-47d8-83de-6af50d938e7f",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "png-qr-code",
    "response" : "<QR Code as PNG encoded using base64 URL>"
  }
}
```

9.8.2.11. HTTP Status Codes

The following HTTP status codes are returned by the Dispatch Token Service endpoint:

Table 77. HTTP status codes - Dispatch Token Service

HTTP Code	Description
200	**OK** The server created and successfully dispatched a token.
400	**Bad Request** The provided payload was not properly formatted.
401	**Unauthorized** The request was not authorized. There is an invalid SecToken or unresolved username.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/json`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/json;charset=UTF-8`.
500	**Internal Server Error** The server could not process the request because of an unexpected error.

9.8.3. Redeem Token Service

The FIDO client uses the Redeem Token Service to trigger a FIDO UAF registration, authentication or deregistration. Therefore, the client must provide the token it previously obtained.

9.8.3.1. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/token/redeem/<operation>
```

The `<operation>` in the base URL can be `registration`, `authentication` or `deregistration`.

Having split endpoints allows you to protect nevisFIDO differently for each operation, by means of nevisProxy and nevisAuth.
If you cannot guarantee a secure transmission of the token to the targeted user, you better protect the registration and deregistration endpoints. Otherwise, malicious clients might use stolen tokens to register new credentials and to deregister existing ones.

9.8.3.2. HTTP Methods

`POST` is the only supported HTTP method.

9.8.3.3. Request Headers

The following request headers are mandatory:

Table 78. Mandatory request headers - Redeem Token Service

Name	Description
`Accept`	Accept header, **must** be `application/fido+uaf`.
`Content-Type`	Content type header, **must** be `application/json`.

9.8.3.4. Request Body

In order to trigger the registration, authentication or deregistration process, the Redeem Token Service requires a request body with a JSON payload containing a token (in the `token` attribute). The next table shows the elements of the request body (JSON payload).

Table 79. Request body - Redeem Token Service

Attribute	Type	Description	Optional
`token`	`String`	The token previously obtained using the token creation endpoint in "/nevisfido/token/create/authentication".	false

9.8.3.5. Response Headers

The following response headers will be set:

Table 80. Response headers - Redeem Token Service

Name	Description
`Content-Type`	Content type header, fixed to `application/fido+uaf;charset=UTF-8`.

9.8.3.6. Response Body

The Redeem Token Service returns a JSON response body containing a `ReturnUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification.

The `ReturnUAFRequest` object has the following structure:

Table 81. ReturnUAFRequest object - Redeem Token Service

Path	Type	Description
`statusCode`	`Number`	UAF status code for the operation.

Path	Type	Description
`uafRequest`	`String`	The new UAF request message if the server decides to issue one.
`op`	`String`	Hint to the client regarding the operation type of the message, must be set to either one of `Reg`, `Auth`, or `Dereg`.
`lifetimeMillis`	`Number`	Hint informing the client application of the lifetime of the message in milliseconds. Absent if the operation was not successful.

The FIDO client either wants to trigger a FIDO registration, authentication or deregistration operation. Depending on the requested operation, the `uafRequest` part of the returned `ReturnUAFRequest` object will look different. If the FIDO client requested a registration operation, the `uafRequest` part will contain a `RegistrationRequest` object (see the RegistrationRequest dictionary for details). In case of an authentication operation, the `uafRequest` part contains an `AuthenticationRequest` object (see the AuthenticationRequest dictionary for details).

9.8.3.7. Example Request

```
POST /nevisfido/token/redeem/authentication HTTP/1.1
Accept: application/fido+uaf
Content-Type: application/json
Host: fido.siven.ch
Content-Length: 54

{
  "token" : "3e88dce0-dbbb-4297-9233-182d0961e9bc"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/token/redeem/authentication' -i -X POST
\
    -H 'Accept: application/fido+uaf' \
    -H 'Content-Type: application/json' \
    -d '{
  "token" : "3e88dce0-dbbb-4297-9233-182d0961e9bc"
}'
```

9.8.3.8. Example Response (1)

If the token provided in the `token` attribute of the request was created by nevisFIDO and not previously redeemed, the response will look like this:

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:07 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 659

{
  "lifetimeMillis" : 120000,
  "uafRequest" :
"[{\"header\":{\"serverData\":\"n_HYP3myMR48bWaQAL5Fxnp_RPUBAlgyegJg38jTPvl-
WGl0BwzTcL08D2679rwGWWGYLGNUYs_-
ABvrkZ_Egg\",\"upv\":{\"major\":1,\"minor\":1},\"op\":\"Auth\",\"appID\":\"http
s://www.siven.ch/appID\",\"exts\":[{\"id\":\"ch.nevis.auth.fido.uaf.sessionid\"
,\"data\":\"sessionId\",\"fail_if_unknown\":false}]},\"challenge\":\"dkhdADyDoL
0Sh28_qrtvrlTPdGiHOG466gitSrK9sKzHWInrNC8vODTlzBSCIBQC7dS7sUYSjFD2HEYd55h1Aw\",
\"policy\":{\"accepted\":[[{\"userVerification\":1023,\"authenticationAlgorithm
s\":[1,2,3,4,5,6,7,8,9],\"assertionSchemes\":[\"UAFV1TLV\"]}]]}}]",
  "statusCode" : 1200,
  "op" : "Auth"
}
```

Note that in the above sample a `ReturnUAFRequest` object with operation type "Auth" is returned (`op` = "Auth"). This indicates that the initial `GetUAFRequest` object coming from the FIDO client/nevisAuth requested for an authentication operation. This also means that in the above sample, the `uafRequest` part of the `ReturnUAFRequest` object contains an `AuthenticationRequest`. Furthermore, the returned UAF status code is "1200", which indicates that the operation was successful (`statusCode` = "1200"). For a complete list of all possible UAF status codes that can be returned by the FIDO server, see UAF Status Codes.

9.8.3.9. Example Response (2)

If the `token` provided in the request was not created by nevisFIDO or it had been previously redeemed, the response will look like this:

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:07 GMT
Content-Type: application/fido+uaf;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 25

{
  "statusCode" : 1491
}
```

Note that in this case, the returned UAF status code is "1491". This means that the FIDO server considers the request as invalid. In such a case, the `ReturnUAFRequest` object does not contain a UAF request (`uafRequest` part), as you can see in the sample above.

9.8.3.10. HTTP Status Codes

The following HTTP status codes are returned by the Redeem Token Service:

Table 82. HTTP status codes - Redeem Token Service

HTTP Code	Description
200	**OK** The server processed the request successfully. It returns a `ReturnUAFRequest` JSON object containing a `RegistrationRequest` or an `AuthenticationRequest` (depending on the operation specified in the initial `GetUAFRequest`).
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/fido+uaf`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/json;charset=UTF-8`.

9.8.4. Create Token Service

Do not use the Create Token Service for out-of-band scenarios. Instead, use the Dispatch Token Service.

You use the Create Token Service to generate tokens. To trigger the generation of a token, you must provide a `GetUAFRequest` object to the Create Token Service when accessing the service.

9.8.4.1. HTTP Methods

`POST` is the only supported HTTP method.

9.8.4.2. Base URL

All URLs referenced in this section have the following base:

```
https://fido.siven.ch/nevisfido/token/create/<operation>
```

The `<operation>` in the base URL can be `registration`, `authentication` or `deregistration`.

Having split endpoints allows you to protect nevisFIDO differently for each operation, by means of nevisProxy and nevisAuth.
The registration and deregistration endpoints must be protected or it would be easy for malicious clients to register new credentials and deregister existing ones.

9.8.4.3. Request Headers

The following request headers are mandatory:

Table 83. Mandatory request headers - Create Token Service

Name	Description
`Accept`	Accept header, **must** be `application/json`.
`Content-Type`	Content type header, **must** be `application/fido+uaf;charset=UTF-8`.

9.8.4.4. Request Body

The Create Token Service requires a JSON payload with a `GetUAFRequest` object as defined in the FIDO UAF HTTP Transport Specification. The `GetUAFRequest` object has the following structure:

Table 84. GetUAFRequest object - Create Token Service

Attribute	Type	Description	Optional
`op`	`String`	The request operation, must be set to either one of `Reg`, `Auth`, or `Dereg`.	false
`previousRequest`	`String`	If the application is requesting a new UAF request message because the previous one expired, the previous one could be sent to the server.	true
`context`	`String`	The contextual information must be a **stringified JSON** object that conforms to the relevant Context dictionary.	false

9.8.4.5. Response Headers

The following response headers will be set:

Table 85. Response headers - Create Token Service

Name	Description
`Content-Type`	Content type header, fixed to `application/json`.

9.8.4.6. Response Body

The body of the response message coming from the Create Token Service contains the token (in the `token` attribute). The `statusCode` attribute shows the HTTP status code of the token creation. If the token creation was successful, the code is "1200". If the token creation was not successful, the status code indicates why. The table below lists all elements of the response body.

Table 86. Response body - Create Token Service

Path	Type	Description
`token`	`String`	The token generated by *nevisFIDO*. This is the token that can be used by a client to trigger the UAF operation (registration, authentication or deregistration). Therefore, the token must be sent to the corresponding endpoint ("/nevisfido/token/redeem/registration" for registration, "/nevisfido/token/redeem/authentication" for authentication and "/nevisfido/token/redeem/deregistration" for deregistration), in order to redeem the token and trigger the FIDO UAF operation with the `GetUAFRequest` sent in this request.
`sessionId`	`String`	The identifier of the session generated by *nevisFIDO*. This session identifier can be used by a client to retrieve the status of the authentication. The session ID must be sent to the "/nevisfido/status" endpoint to get the operation status.

Path	Type	Description
`statusCode`	`Number`	The result of the token creation. Possible values: `1,200` if the token could be created. `1,400` if the token could not be created because the request was not a valid `GetUAFRequest`. `1,401` if the user did not provide authentication credentials or if the credentials were invalid. `1,403` if the user is not allowed to create a token. `1,498` if there was a problem with the contents of the `GetUAFRequest`. For example the operation (registration, authentication, deregistration) was not sent to the correct endpoint. `1,500` if the token could not be created because of an unexpected error in the server.

9.8.4.7. Example Request

```
POST /nevisfido/token/create/registration HTTP/1.1
Accept: application/json
Content-Type: application/fido+uaf;charset=UTF-8
Host: fido.siven.ch
Content-Length: 59

{
  "context" : "{\"username\":\"jeff\"}",
  "op" : "Reg"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/token/create/registration' -i -X POST \
    -H 'Accept: application/json' \
    -H 'Content-Type: application/fido+uaf;charset=UTF-8' \
    -d '{
  "context" : "{\"username\":\"jeff\"}",
  "op" : "Reg"
}'
```

9.8.4.8. Example Response

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:12:59 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 133

{
  "token" : "232ce5e5-bd26-4d68-ad82-e397eb55eb87",
  "sessionId" : "d7545aaa-d7f3-4e6e-8111-afda4ea02375",
  "statusCode" : 1200
}
```

9.8.4.9. HTTP Status Codes

The following HTTP status codes are returned by the Create Token Service:

Table 87. HTTP status codes - Create Token Service

HTTP Code	Description
200	**OK** The server processed the request successfully.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/json`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/fido+uaf;charset=UTF-8`.

9.9. Status Service

This chapter describes the Status Service. The Status Service is **not a standard FIDO service** but a proprietary nevisFIDO functionality. The service is a public HTTP API that you can call to find out the status of an operation (registration or authentication) for a given user in nevisFIDO. It allows nevisAuth to check whether a given user has been registered/authenticated or not.

9.9.1. Base URL

All URLs referenced in this chapter have the following base:

```
https://fido.siven.ch/nevisfido/status
```

9.9.2. HTTP Methods

`POST` is the only supported HTTP method.

9.9.3. Request Headers

The following request headers are mandatory:

Table 88. Mandatory request headers - Status Service

Name	Description
`Accept`	Accept header, **must** be `application/json`.
`Content-Type`	Content type header, **must** be `application/json`.

9.9.4. Request Body

The Status Service requires a request message containing a JSON payload with the following structure:

Table 89. Request body - Status Service

Attribute	Type	Description	Optional
`sessionId`	`String`	The session ID identifying the authentication session established in *nevisFIDO*.	false

9.9.5. Response Headers

The following response headers will be set:

Table 90. Response headers - Status Service

Name	Description
`Content-Type`	Content type header, fixed to `application/json`.

9.9.6. Response Body

The Status Service returns a response message that shows the operation status for the provided session ID.

The table further below describes the structure of the response message's (JSON) body.

- The `status` attribute in the response message holds the status information. The attribute can have the following values:

  `tokenCreated`
  : nevisFIDO has generated a token through the Dispatch Token Service, but the token has not been redeemed yet. If the initial request included a dispatching of the token, being in this state implies that nevisFIDO successfully transmitted the token to the dispatching service. However, the final recipient may not have received the token yet.

  `clientRegistering`
  : nevisFIDO has sent a `RegistrationRequest` to the client, but the FIDO client has not sent a response yet. That is, nevisFIDO is waiting for the user to interact with the FIDO UAF authenticator, and for the client to send a `RegistrationResponse`. nevisFIDO generated the registration request after either redeeming a token in the Redeem Token Service, or receiving a `GetUAFRequest` via the FIDO UAF Registration Request Service.

  `clientAuthenticating`
  : nevisFIDO has sent an `AuthenticationRequest` to the client, but the FIDO client has not sent a response yet. That is, nevisFIDO is waiting for the user to interact with the FIDO UAF authenticator, and for the client to send a `AuthenticationResponse`. nevisFIDO generated the authentication request after either redeeming a token in the Redeem Token Service, or receiving a `GetUAFRequest` via the FIDO UAF Authentication Request Service.

  `failed`
  : The registration or authentication operation failed.

  `succeeded`
  : The user completed the operation successfully in nevisFIDO.

`unknown`
: The provided session ID is unknown. This can happen if the session ID does not correspond to any session started with nevisFIDO or if nevisFIDO purged the information regarding the session.

- The `uafStatusCode` attribute in the response message provides the UAF status code returned by nevisFIDO when the operation is completed.

The Status Service returns a JSON response body with the following structure:

Table 91. Response body - Status Service

Path	Type	Description
`status`	`String`	The operation status for the provided session ID.
`timestamp`	`String`	ISO-8601 timestamp in UTC when the status was last updated, for example, "2018-04-19T14:48:24.429Z". Absent when the status is unknown.
`uafStatusCode`	`Number`	The UAF status code for the operation, see UAF Status Codes. Absent if the operation is not completed.
`asmStatusCode`	`Number`	The ASM status code sent by the FIDO client when sending a Registration or an Authentication response. The possible values are defined in ASM Status Codes. Absent if the operation is not completed or if the client did not send it.
`clientErrorCode`	`Number`	The client error code sent by the FIDO client when sending a Registration or an Authentication response. The possible values are defined in Client Error Codes. Absent if the operation is not completed or if the client did not send it.
`userId`	`String`	The technical identifier of the user who successfully completed the operation. The nature of this attribute depends on the type of credential repository used and in general is not the same as the username provided by the FIDO clientin the context of the `GetUAFRequest`. This value is only present if the operation is completed successfully.

Path	Type	Description
`tokenInformation`	`Object`	The object describing the token managing status. This object is available when the Dispatch Token Service was used and the token could be successfully created and dispatched.
`tokenInformation.dispatcherInformation.name`	`String`	The dispatcher used to dispatch the token. It is present if the client asked to dispatch the token.
`tokenInformation.dispatcherInformation.response`	`String`	The information provided by the dispatcher after dispatching the token. If there was an unexpected error, this is also reported here.
`tokenInformation.tokenResult`	`String`	The result of the token management. This is only available when the server finished processing the token. It can contain the following values: `tokenRedeemed`: if the token was successfully redeemed; `tokenTimedOut`: if the token was not redeemed and has timed-out (i.e. it cannot be redeemed anymore).
`authenticators`	`Array`	See Authenticators. In the case of a successfully completed registration or authentication, this field contains information regarding the authenticators used to generate the validated assertions. In the case of deregistration, information regarding the deregistered authenticators.
`authenticators[].aaid`	`String`	The AAID of the authenticator.

9.9.7. Example Request (Successful Operation)

```
POST /nevisfido/status HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: fido.siven.ch
Content-Length: 58

{
  "sessionId" : "3ab14f07-5bb4-49ea-a1b9-e7b86a77c4c0"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/status' -i -X POST \
    -H 'Accept: application/json' \
    -H 'Content-Type: application/json' \
    -d '{
  "sessionId" : "3ab14f07-5bb4-49ea-a1b9-e7b86a77c4c0"
}'
```

9.9.8. Example Response (Successful Operation)

Note that the `tokenInformation` attribute is present, which indicates that the operation was triggered using the Dispatch Token Service.

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:14 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 415

{
  "status" : "succeeded",
  "timestamp" : "2022-08-03T13:13:14.691Z",
  "tokenInformation" : {
    "tokenResult" : "tokenRedeemed",
    "dispatcherInformation" : {
      "name" : "firebase-cloud-messaging",
      "response" : "successful dispatch"
    }
  },
  "uafStatusCode" : 1200,
  "asmStatusCode" : 0,
  "clientErrorCode" : 0,
  "userId" : "userId",
  "authenticators" : [ {
    "aaid" : "ABBA#0001"
  } ]
}
```

9.9.9. Example Request (Failed Operation)

```
POST /nevisfido/status HTTP/1.1
Accept: application/json
Content-Type: application/json
Host: fido.siven.ch
Content-Length: 58

{
  "sessionId" : "a0cc590d-f3d3-469e-bca0-eeb1b13c512d"
}
```

cURL:

```
$ curl 'https://fido.siven.ch/nevisfido/status' -i -X POST \
    -H 'Accept: application/json' \
    -H 'Content-Type: application/json' \
    -d '{
  "sessionId" : "a0cc590d-f3d3-469e-bca0-eeb1b13c512d"
}'
```

9.9.10. Example Response (Failed Operation)

Note that the `tokenInformation` attribute is not present, which indicates that the operation was **not** triggered using the Dispatch Token Service.

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2022 13:13:14 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Content-Length: 145

{
  "status" : "failed",
  "timestamp" : "2022-08-03T13:13:14.823Z",
  "uafStatusCode" : 1498,
  "asmStatusCode" : 11,
  "clientErrorCode" : 12
}
```

9.9.11. HTTP Status Codes

The following HTTP status codes are returned by the Status Service:

Table 92. HTTP status codes - Status Service

HTTP Code	Description
200	**OK** The server processed the request successfully. Check the response body for the status information.
400	**Bad Request** The provided JSON payload does not match the defined structure in the Request Body section.
405	**Method Not Allowed** The method of the received request was not "POST".
406	**Not Acceptable** The `Accept` header is not properly set to `application/json`.
415	**Unsupported Media Type** The `Content-Type` header is not properly set to `application/json;charset=UTF-8`.

10. Dispatcher

A dispatcher sends a token generated by nevisFIDO. If the token must be sent to a precise recipient (usually a third party), the dispatcher uses a *dispatch target*. A *dispatch target* contains the information required by the dispatcher to send the token to a third entity and guarantee that only that third party is able to consume it.

For concept information regarding dispatchers, refer to the Mobile Authentication Concept and Integration Guide.

The Dispatch Target Service is used to create, delete and modify dispatch targets managed by nevisFIDO.
The Dispatch Token Service is used to generate and dispatch tokens. The HTTP client asks nevisFIDO to generate and dispatch such a token, by providing the identifier of a dispatch target to the Dispatch Token Service and/or the name of the dispatcher to be used.

Dispatchers are implemented using Plug-Ins.

10.1. FCM (Firebase Cloud Messaging) Dispatcher

The FCM (Firebase Cloud Messaging) dispatcher supports out-of-band scenarios. Besides the Out-of-Band Services it relies on Google's Firebase Cloud Messaging service. Thus, the FCM dispatcher is able to dispatch tokens and additional dispatch information to mobile devices using push messages.

As a prerequisite a Firebase project has to be created using a Google account.

The *FCM Dispatcher* plug-in needs the service account file of your Firebase project to cooperate with Cloud services. Carefully read the related documentation on how to generate and download the service account JSON file.

The service account JSON file contains sensitive information, including your service account's private encryption key. Keep it confidential and never store it in a public repository.

10.1.1. Configuration

Use the *firebase-cloud-messaging* dispatcher type in the Dispatchers Configuration to enable FCM dispatching.

The table below describes the possible configuration options.

Table 93. FCM Dispatcher Configuration Options

Configuration Key	Mandatory	Type	Default value	Description
`service-account-json`	yes	String	-	Path to the valid FCM service account file to be used.
`dry-run`	no	Boolean	false	Flag for testing the dispatch functionality without actually delivering the message to the target device.
`endpoint-base-url`	no	String	https://fcm.googleapis.com	The base URL of the FCM Service endpoint. This URL could be used to configure a mock service for testing.
`proxy-url`	no	String	-	The URL of the HTTP proxy required to access the FCM service. This URL is required if the FCM service is only accessible through a proxy. Example: https://proxy.siven.ch:3128.
`registration-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/registration	The URL that must be used to redeem the registration tokens.
`authentication-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/authentication	The URL that must be used to redeem the authentication tokens.

Configuration Key	Mandatory	Type	Default value	Description
`deregistration-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/deregistration	The URL that must be used to redeem the deregistration tokens.

10.1.1.1. Configuration Examples

- The next example shows a typical production environment setup. Such a setup usually only contains the mandatory *service-account-json* configuration option:

```
dispatchers:
    - type: firebase-cloud-messaging
      service-account-json: /var/opt/nevisfido/default/conf/service-
account.json
      registration-redeem-url:
https://siven.ch/nevisfido/token/redeem/registration
      authentication-redeem-url:
https://siven.ch/nevisfido/token/redeem/authentication
      deregistration-redeem-url:
https://siven.ch/nevisfido/token/redeem/deregistration
```

- The next example shows a typical integration environment setup. In such a setup, the *dry-run* flag may be enabled to call Google's FCM service, for example to test the connection. However, there are no push messages sent out to the target devices.

```
dispatchers:
    - type: firebase-cloud-messaging
      service-account-json: /var/opt/nevisfido/default/conf/service-
account.json
      dry-run: true
      registration-redeem-url:
https://integration.siven.ch/nevisfido/token/redeem/registration
      authentication-redeem-url:
https://integration.siven.ch/nevisfido/token/redeem/authentication
      deregistration-redeem-url:
https://integration.siven.ch/nevisfido/token/redeem/deregistration
```

- The next example shows a typical developer environment setup. Such a setup may use an FCM mockup service instead of the real service, and overriding the default *endpoint-base-url* configuration value:

```
dispatchers:
    - type: firebase-cloud-messaging
      service-account-json: /var/opt/nevisfido/default/conf/service-
account.json
      endpoint-base-url: http://localhost:8080/mockfcm
      registration-redeem-url:
http://localhost:9080/nevisfido/token/redeem/registration
      authentication-redeem-url:
http://localhost:9080/nevisfido/token/redeem/authentication
      deregistration-redeem-url:
http://localhost:9080/nevisfido/token/redeem/deregistration
```

An FCM mock implementation is not provided by the nevisFIDO component.

10.1.2. Encryption

For encryption of tokens sent through the Firebase Cloud Messaging push service, the *encryption algorithms* used are currently fixed and cannot be configured. Note that the encryption keys can be administered using the Dispatch Target Service.

For encryption, both RSA and Elliptic Curve (EC) keys are supported. For each of those, the encryption algorithm used is the following:

- **RSA**: RSA-OAEP-256 algorithm ('RSAES using Optimal Asymmetric Encryption Padding with SHA-256 hash function') with encryption method A256CBC_HS512 ('AES_256_CBC_HMAC_SHA_512 authenticated encryption using a 512 bit').
- **EC**: ECDH-ES+A256KW algorithm ('Elliptic Curve Diffie-Hellman Ephemeral Static key agreement, where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the A256KW function') with encryption method A256CBC_HS512 ('AES_256_CBC_HMAC_SHA_512 authenticated encryption using a 512 bit').

10.1.3. Dispatch Target Format

In addition to configuring the FCM Dispatcher plug-in clients, you must register the dispatch targets with the HTTP API of the Dispatch Target Service. Configure the dispatch targets as follows:

- The registered dispatch targets **must** contain an `encryptionKey`.
- Set the `target` attribute of the dispatch target to the firebase registration token of the client to which you want to dispatch the token. See the Message API of the Firebase HTTP API for details.

10.1.3.1. Dispatch Target Example

```
{
  "name" : "My Mobile Phone",
  "target" : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P198aDPO",
  "signatureKey" : {
    "kty" : "RSA",
    "x5t#S256" : "cCrhkZ3Q5kspKpZTjUfqTT89PZKI4EW7_4QtfniYiBk",
    "e" : "AQAB",
    "use" : "sig",
    "kid" : "7852318338397317936",
    "x5c" : [
"MIICujCCAaKgAwIBAgIIbPkJ7Od1KzAwDQYJKoZIhvcNAQELBQAwHTELMAkGA1UEBhMCY2gxDjAMBg
NVBAMTBXNpdmVuMB4XDTIyMDgwMjE1MTMyM1oXDTI0MDgwMzE1MTMyM1owHTELMAkGA1UEBhMCY2gxD
jAMBgNVBAMTBXNpdmVuMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoi42hm1FJNZQVzeK
868l2la8zkaVwSFfvTbm1mJBbkHzV2bJ2eNmUsfSNSbHrCMuVH3iBy6m9lfNcJB5buEGPA+8PFfIPJP
```

```
elbxVijXYco2CIRvuCPLGb3n1wCEftMKW0Fw946qA9EXNZj9BYRTSPqgBxYlCfenBpztxtBkxKibsXc
OCKdKwBua6kvKwhtgobdZhSIxc5Bb+ZUhJ/jEEE5mTiRA9XlDCtT1cw4N3lnrQuvCJBz9eJpLeT2trB
U0IaeIdkavKhijYmY5KlDNqnXPRbP5jSiFqiG/FmRe2dGsZgpmfihAnxDxxCkyZ+36OqdEBhyaSHWqx
aYz/UBGRuwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQBuSX0zeGZe9JwhsjYo1XQF0GSnNIEQRsFgfw1
JYP0qiWnYTPJK42ZatSLaFBB4le8hVGZSiI8fTVDTjLKyQZL2lLqu2KR9eSlA8G+E47R3R4mFYcOA65
XFCMS1SV+BijaqGnRnfMFLMmvmd4RKk49n7fuaEdHBA2rPaTgsjYhn4rFtmxa3Dp4CNYqNeJLRAuZaQ
zPxk4zlPeLLoXb+oag4mlKlnI0GE1ULX4RWDXaDy1awzsSFtGuhEJpiO+AJy/No7i+0sif/s/J+4Ul4
MrrDE9W+RPckrBuiSx/9XGd7+wk5yzwxpnCY+KAKFQ1Hd1Y+yRfRP+uLYGYjV1H2+1ml" ],
    "n" :
"oi42hm1FJNZQVzeK868l2la8zkaVwSFfvTbm1mJBbkHzV2bJ2eNmUsfSNSbHrCMuVH3iBy6m9lfNcJ
B5buEGPA-
8PFfIPJPelbxVijXYco2CIRvuCPLGb3n1wCEftMKW0Fw946qA9EXNZj9BYRTSPqgBxYlCfenBpztxtB
kxKibsXcOCKdKwBua6kvKwhtgobdZhSIxc5Bb-
ZUhJ_jEEE5mTiRA9XlDCtT1cw4N3lnrQuvCJBz9eJpLeT2trBU0IaeIdkavKhijYmY5KlDNqnXPRbP5
jSiFqiG_FmRe2dGsZgpmfihAnxDxxCkyZ-36OqdEBhyaSHWqxaYz_UBGRuw"
  },
  "encryptionKey" : {
    "kty" : "RSA",
    "x5t#S256" : "z6iXW9YfMv3NdDf_L9JjMTct_FyJzLiI2eAj3BcTJf8",
    "e" : "AQAB",
    "use" : "enc",
    "kid" : "17100409740410464738",
    "x5c" : [
"MIICuzCCAaOgAwIBAgIJAO1Q3A6ay6niMA0GCSqGSIb3DQEBCwUAMB0xCzAJBgNVBAYTAmNoMQ4wDA
YDVQQDEwVzaXZlbjAeFw0yMjA4MDIxNTEzMjNaFw0yNDA4MDMxNTEzMjNaMB0xCzAJBgNVBAYTAmNoM
Q4wDAYDVQQDEwVzaXZlbjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMkHHUi8AR4esykb
o7YaAjAhuN20zFj2iN+VxwMfFVJ1H6OKUtAa3+qzibDBz+93gt0obyp0FckfEHYs9Hg4qBkbYzw7ccR
ed4nnYa8mVmGXJRaxrFkpqHdQi3LAiJOZ4LZAI2jeneVN85H+tj2MYeoIn7c+Xw02HutVWnNVcWrrGS
3NHrTctwBIEzsuB3MnflJFkUtcjkINSeDCDYoaC93Jqt2jyOFqx/kS0ubQU5yc169N3vYwo70JdmDFw
0k7HhhLzqegmdNSQ845uuo6FNSg0LG/xVkauseUkeN1+oZB0m33cT3P2Y8/dV29xOvNkgWQp8qBKGPU
brZUXq3grc0CAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAvJlqfe/NfAr5DtU1wnmSaYevEAXMjbD5zcR
ISDFxoLXtSqj0raD3siDoT52AgHY/l9gr4pGgnxy4d62EMRX9MGTjssv9VqUnkcGhQc9QXu52QCQZlm
ppko0MJhmD8twdx1k7dDI88TxFn+yVPaWDsKBsb2XLrVMP1OFeLe4eTbrfVejG0dgG5eTGZbNUcVxbj
pOhHkBDIt8du6qVVEJ5beNNnbA4mcdZq6mpeebGtRzzCoxrz9wZK1e8I42O0eTvsBEDQ6jCNSkUjV1q
qtN25Xbc1275KHE2J3DDlc5LQeWWQEnbLu6Oq1xHMvtpIYeESVrUqHezFQ9V4kOBWRwhZw==" ],
    "n" :
"yQcdSLwBHh6zKRujthoCMCG43bTMWPaI35XHAx8VUnUfo4pS0Brf6rOJsMHP73eC3ShvKnQVyR8Qdi
z0eDioGRtjPDtxxF53iedhryZWYZclFrGsWSmod1CLcsCIk5ngtkAjaN6d5U3zkf62PYxh6giftz5fD
TYe61Vac1VxausZLc0etNy3AEgTOy4Hcyd-UkWRS1yOQg1J4MINihoL3cmq3aPI4WrH-
RLS5tBTnJzXr03e9jCjvQl2YMXDSTseGEvOp6CZ01JDzjm66joU1KDQsb_FWRq6x5SR43X6hkHSbfdx
Pc_Zjz91Xb3E682SBZCnyoEoY9RutlRereCtzQ"
  },
  "username" : "username",
  "deviceId" : "Acme Inc Phone. Serial Number Hash:
e14c2cec1f8c448a47874b5e164df11727a9e0ad"
}
```


10.1.4. Dispatch Token Request Format

Use the `dispatchInformation` attribute of the Dispatch Token Service to specify information for the dispatcher. In case of the FCM Dispatcher, the client can provide the content of the `notification` attribute that will be sent in the Message to the FCM infrastructure.

The client can also provide additional data in JSON format that will be encrypted and included in the push message. This is used for example to implement the Channel Linking.

Table 94. FCM Dispatch Information Dictionary

Attribute	Type	Description	Optional
`notification`	`Object`	The contents of the notification that will be included in the `notification` attribute of the push message.  *If neither the `title` nor the `body` property is defined inside `notification`, or the `dispatchInformation` is not defined or it does not contain a `notification` attribute, then a Data Message will be sent out via FCM instead of a Notification Message. This might affect the behavior of the mobile client. For further details regarding the different message types, consult the documentation About FCM messages.*  *It is recommended that you always specify at least the notification `title` as an information for the end user.*	true (but it is recommended to include it)
`data`	`Object`	The additional data that will be encrypted into the `nma_data` attribute of the push message.  *This information will be sent inside the push message, so its size limitations apply here.*	true

10.1.4.1. Dispatch Token Request Example

```
{
  "dispatchTargetId" : "1c564833-e0c7-470f-8192-e036a1f60c66",
  "dispatcher" : "firebase-cloud-messaging",
  "dispatchInformation" : {
    "notification" : {
      "title" : "Dirk Gently Bank - Confirm the payment"
    },
    "data" : {
      "channelLinking" : {
        "mode" : "visualString",
        "content" : "AB"
      }
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Auth"
  }
}
```

The above example of a dispatch token request will generate a push message including a notification with the title "Dirk Gently Bank - Confirm the payment".

In addition to that, the specified information in the `data` attribute (i.e the `channelLinking` object) will be encrypted and sent with the token and the redeem URL in the `nma_data` attribute of the push message. The data in the example corresponds to what is used in the context of Channel Linking.

10.1.5. Push Message Dispatching

When a dispatch token request arrives on the Dispatch Token Service interface, the FCM Dispatcher will compose and send a push message via the HTTP v1 API of FCM. For example the FCM Dispatcher will send the following message:

```
{
  "validate_only" : false,
  "message" : {
    "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
    "data" : {
      "nma_data_version" : "1",
      "nma_data_content_type" : "application/jose",
      "nma_data" : "<the encrypted data>"
    },
    "notification" : {
      "title" : "Dirk Gently Bank - Confirm the payment"
    },
    "android" : {
      "priority" : "high"
    },
    "apns" : {
      "headers" : {
        "apns-priority" : "10"
      },
      "payload" : {
        "aps" : {
          "sound" : "default"
        }
      }
    }
  }
}
```

Push messages are always delivered with the highest priority to the devices. To achieve this the `priority` is set to `high` in the Android specific section of the message while for iOS devices the `apns-priority` header is set to `10` for the same purpose. For further details please check the related chapter of the FCM documentation where additional platform specific documentations are also linked.

Currently the value at key `nma_data_version` is always set to `"1"`, and the value at key `nma_data_content_type` is always set to `"application/jose"`.

The data at key `nma_data` contains a standard JWE using compact serialization with the following structure:

Table 95. FCM Dispatch Message Encrypted Payload Dictionary

Attribute	Type	Description	Optional
`token`	`String`	The NEVIS Mobile Authentication token.	false
`redeem_url`	`String`	NEVIS Mobile Authentication token redemption URL.	false

In addition to the always present attributes above, the payload can contain optional JSON provided in the dispatch information of the request (see Dispatch Token Request Format).

The encrypted data payload in the example above is the following one:

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "redeem_url" : "https://fido.siven.ch/nevisfido/token/redeem/authentication",
  "channelLinking" : {
    "mode" : "visualString",
    "content" : "AB"
  }
}
```

For more information about how to receive and process incoming push messages on different target device platforms, refer to the Cloud Messaging documentation of FCM.

10.2. QR Code Dispatcher

The QR Code dispatcher allow to generate a QR code that is included in the HTTP response. This QR code has the information required by the mobile application (notably the token and the redeem URL) to proceed with the operation (registration, authentication or deregistration).

This can be used for example as a fallback mechanism when a push message could not be sent to the client. The HTTP client can display the returned QR code to the end user. Then the QR code can be scanned by the end-user using the mobile authenticating device. So this can be used as an "out-of-band" mechanism to transmit the required information to proceed with the operation.

The QR codes are generated in PNG format.

10.2.1. Configuration

Use the *png-qr-code* dispatcher type in the Dispatchers Configuration to enable PNG QR code dispatching. This is the value that must also be used by the HTTP client in the Dispatch Token Service when specifying the value of the `dispatcher` attribute, if the QR code dispatcher must be used.

The table below describes the possible configuration options.

Table 96. QR Code Dispatcher Configuration Options

Configuration Key	Mandatory	Type	Default value	Description
`registration-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/registration	The URL that must be used to redeem the registration tokens.
`authentication-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/authentication	The URL that must be used to redeem the authentication tokens.
`deregistration-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/deregistration	The URL that must be used to redeem the deregistration tokens.

10.2.1.1. Configuration Example

- The next example shows a development environment setup where the PNG QR code dispatcher is enabled:

```
dispatchers:
- type: png-qr-code
  registration-redeem-url:
http://localhost:9080/nevisfido/token/redeem/registration
  authentication-redeem-url:
http://localhost:9080/nevisfido/token/redeem/authentication
  deregistration-redeem-url:
http://localhost:9080/nevisfido/token/redeem/deregistration
```

10.2.2. Encryption

If the client specifies a Dispatch Target Service identifier in the dispatch token request, then the encryption key of the dispatch target will be used to encrypt the contents that will be encoded in the QR code. By using encryption it is guaranteed that only the device associated with the dispatch target can use the generated token. The use of encryption avoids social engineering attacks such as QRLJacking.

The same algorithms described in the FCM dispatcher Encryption section are used with this dispatcher.

Note that to use encryption, dispatch targets must be configured as described in the Dispatch Target Format section.

10.2.3. Dispatch Targets

This dispatcher does not require a `target` to be provided in the dispatch target and thus the dispatch targetes created for the FCM dispatcher can be used with this dispatcher.

It is recommended to reuse the dispatch targets created for the FCM dispatcher when using this dispatcher (i.e. do not create new dispatch targets for this dispatcher if dispatch targets for the FCM dispatcher are defined). Reusing the same dispatch target simplifies the managing of the dispatch targets on the authenticating (mobile) device.

10.2.4. Dispatch Token Request Format

Use the `dispatchInformation` attribute of the Dispatch Token Service to specify information for the dispatcher. In case of the QR code dispatcher, the client can provide the additional data in JSON format that will be included in the generated QR code. In addition to this data, the client can provide the size and colors to be used to generate the QR code.

Table 97. QR Code Dispatch Information Dictionary

Attribute	Type	Description	Default Value	Optional
`data`	`Object`	The additional data that will be encrypted into the `nma_data` attribute of the QR code.  *This information will be sent inside the QR code, so its size limitations apply here.*	-	true
`width`	`number`	The width (in pixels) of the QR code. The maximum allowed width is 512 pixels.	300	true
`height`	`number`	The height (in pixels) of the QR code. The maximum allowed height is 512 pixels.	300	true

Attribute	Type	Description	Default Value	Optional
`foregroundColor`	`String`	The color (as RGB String) of the foreground of the QR code.	`rgb(0, 0, 0)` (Black)	true
`backgroundColor`	`String`	The color (as RGB String) of the background of the QR code.	`rgb(255, 255, 255)` (White)	true

10.2.4.1. Dispatch Token Request Example with Encryption

```
{
  "dispatchTargetId" : "1c564833-e0c7-470f-8192-e036a1f60c66",
  "dispatcher" : "png-qr-code",
  "dispatchInformation" : {
    "data" : {
      "attributeName" : "some additional data to be included in the QR code"
    },
    "encodingParameters" : {
      "width" : 300,
      "height" : 300,
      "backgroundColor" : "rgb(255, 255, 255)",
      "foregroundColor" : "rgb(0, 0, 0)"
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Auth"
  }
}
```

In the example above a dispatch target is specified, and thus some of the contents that will be encoded as a QR code will be encrypted.

10.2.5. Dispatch Token Response Format

The format of the dispatch token is described in Dispatch Token Service. The dispatcher returns the QR code in the `dispatcherInformation.response` attribute. The QR code is returned as a base64 encoded String. This way the QR code can be easily rendered as an image using the data scheme described in RFC 2397. For example:

```
<img src=" [PNG as base 64]"/>
```

The String in the generated QR code is JSON encoded as a base64 URL String. The JSON uses UTF-8 encoding. The reason to use base64 URL instead of directly JSON with UTF-8 is to maximize the compatibility with different QR code scanners: may of the scanners assume UTF-8 as the character encoding, but others use other character encodings such as ISO-8859 or JIS8, which are the default character encoding for some QR code standards. Using only ASCII characters avoids having character encoding issues at the scanning level.

10.2.5.1. Dispatch Token Response Example without Encryption

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "86584e0e-a13c-4ff8-8fcc-0ecf517dbd4a",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "png-qr-code",
    "response" : "<QR Code as PNG encoded using base64 URL>"
  }
}
```

The data that is encoded in the QR code is a base 64 URL String. The base64 URL String, once decoded as a UTF-8 String, contains the following JSON:

```
{
  "nma_data" : {
    "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
    "redeem_url" : "
https://fido.siven.ch/nevisfido/token/redeem/authentication",
    "attributeName" : "some additional data to be included in the QR code"
  },
  "nma_data_content_type" : "application/json",
  "nma_data_version" : "1"
}
```

The data structure in the `nma_data` attribute is the same as the one generated by the FCM dispatcher (but in this example it is not encrypted).

10.2.5.2. Dispatch Token Response Example with Encryption

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "c9fa045e-2d2b-47d8-83de-6af50d938e7f",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "png-qr-code",
    "response" : "<QR Code as PNG encoded using base64 URL>"
  }
}
```

The data that is encoded in the QR code is a base 64 URL String. The base64 URL String, once decoded as a UTF-8 String, contains the following JSON:

```
{
  "nma_data" : "<encrypted data>",
  "nma_data_content_type" : "application/jose",
  "nma_data_version" : "1"
}
```

The data in the `nma_data` attribute is encrypted using the standard JWE using compact serialization. The data structure is the same as the one generated by the FCM dispatcher. In this example, the encrypted contents are:

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "redeem_url" : "https://fido.siven.ch/nevisfido/token/redeem/authentication",
  "attributeName" : "some additional data to be included in the QR code"
}
```

Note that the specified information in the `data` attribute in the dispatch token request is present in the encrypted payload included in the `nma_data` attribute.

10.3. Link Dispatcher

The Link dispatcher allow to generate links that are included in the HTTP response. These links can be opened by a browser running in a mobile device to launch the access application where the user can register credentials or authenticate (depending on the context). The generated links have the information required by the mobile application (notably the token and the redeem URL) to proceed with the operation (registration, authentication or deregistration). The mobile device client can use the returned links to open the authentication application.

This can be used for example as a fallback mechanism when the device requiring authentication is the same as the authenticating device (usually a mobile phone). In this case using a QR Code Dispatcher is not possible: the device displaying the QR code (device requiring authentication) is the same as the device that must scan the QR code (using the authentication application).

10.3.1. Configuration

Use the *link* dispatcher type in the Dispatchers Configuration to enable link dispatching. This is the value that must also be used by the HTTP client in the Dispatch Token Service when specifying the value of the `dispatcher` attribute, if the link dispatcher must be used.

The administrator must provide the base URL in the `nevisfido.yml` file that will be used by nevisFIDO to generate the links.

The table below describes the possible configuration options:

Table 98. Link Dispatcher Configuration Options

Configuration Key	Mandatory	Type	Default value	Description
`registration-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/registration	The URL that must be used to redeem the registration tokens.
`authentication-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/authentication	The URL that must be used to redeem the authentication tokens.
`deregistration-redeem-url`	no	String	https://[nevisFIDO hostname]/nevisfido/token/redeem/deregistration	The URL that must be used to redeem the deregistration tokens.
`base-url`	yes	String	https://link	The base URL that must be used to generate the links. See Base URL Configuration for details.

Note that at least one operation URL (`registration-redeem-url`, `authentication-redeem-url` or `deregistration-redeem-url`) must be provided for the configuration to be considered valid.

10.3.1.1. Base URL Configuration

The base URL is used to generate the links returned by nevisFIDO. This URL must be the same as the one configured in the mobile application to do the association of the application with a URL.

This URL is composed of a URI scheme (HTTPS or a custom scheme) and a domain. If **no** query parameters are specified in this URL, the generated payload will be attached to the URL as a query parameter with the `?` separator as a standalone query parameter. In case the URL includes query parameters, nevisFIDO will attach the generated payload to the URL with the `&` separator.

There are two approaches to define this URL:

1. Use a Custom Scheme

   Use a custom URI scheme (for example `nevisaccessapp`) that is unique to the application. The provided domain is not relevant when a custom scheme is used.

   The main advantages of this approach are:

   - Only nevisFIDO and the mobile application must be configured.
   - It is compatible with most browsers in the market: it has been tested in Android with Edge, Firefox, Opera, Chrome, QQ, Mi Browser (from Xiaomi), Baidu Browser (6.0.2), UC Turbo, Samsung browser and Huawei browser. The only incompatible browsers found in Android are the UC Browser and UC Lite Browser.

   The main disadvantage is that the custom scheme must be unique for the application.

2. Use an HTTPS URL

   When an HTTPS scheme is used, then the URL must refer a system exposing data used by both Android and iOS to associate the link with the application and to do additional security verifications.

   If the base URL is `https://domain.name`, then a JSON file containing data for Android must be accessible in `https://domain.name/.well-known/assetlinks.json`, and another JSON file containing data for iOS must be accessible in `https://domain.name/.well-known/apple-app-site-association` (or `https://domain.name/apple-app-site-association`).

   See Verify Android App Links and Support Universal Links for details regarding the contents that this 3rd party must expose.

   The main advantages of this approach are:

   - There is no possibility of conflict with other applications.
   - This is considered to be a best practice in both iOS and Android because of the additional security provided by the JSON data verification.

   The main disadvantages are:

   - The number of compatible browsers (Firefox, Safari, Chrome, Edge) is smaller than the one supporting custom scheme.
   - This solution requires to configure a system exposing the JSON data to be verified. This makes this approach to be more difficult to deploy.

10.3.1.2. Configuration Examples

- The next example shows a configuration where the link dispatcher is enabled and uses a custom URI scheme:

```
dispatchers:
- type: link
  registration-redeem-url: https://siven.ch/nevisfido/token/redeem/registration
  authentication-redeem-url:
https://siven.ch/nevisfido/token/redeem/authentication
  deregistration-redeem-url:
https://siven.ch/nevisfido/token/redeem/deregistration
  base-url: nevisaccessapp://x-callback-url/authenticate?x-
success=https://success.siven.ch&x-error=https://error.siven.ch&x-
cancel=https://cancel.siven.ch
```

- The next example shows a configuration where the link dispatcher is enabled and uses HTTPS as scheme:

```
dispatchers:
- type: link
  registration-redeem-url: https://siven.ch/nevisfido/token/redeem/registration
  authentication-redeem-url:
https://siven.ch/nevisfido/token/redeem/authentication
  deregistration-redeem-url:
https://siven.ch/nevisfido/token/redeem/deregistration
  base-url: https://siven.ch
```

10.3.2. Encryption

If the client specifies a Dispatch Target Service identifier in the dispatch token request, then the encryption key of the dispatch target will be used to encrypt the contents that will be included in the link. By using encryption it is guaranteed that only the device associated with the dispatch target can use the generated token.

The same algorithms described in the FCM dispatcher Encryption section are used with this dispatcher.

Note that to use encryption, dispatch targets must be configured as described in the Dispatch Target Format section.

10.3.3. Dispatch Targets

This dispatcher does not require a `target` to be provided in the dispatch target and thus the dispatch targetes created for the FCM dispatcher can be used with this dispatcher.

It is recommended to reuse the dispatch targets created for the FCM dispatcher when using this dispatcher (i.e. do not create new dispatch targets for this dispatcher if dispatch targets for the FCM dispatcher are defined). Reusing the same dispatch target simplifies the managing of the dispatch targets on the authenticating (mobile) device.

10.3.4. Dispatch Token Request Format

Use the `dispatchInformation` attribute of the Dispatch Token Service to specify information for the dispatcher. In case of the link dispatcher, the client can provide the additional data in JSON format that will be included in the generated link.

Table 99. Link Dispatch Information Dictionary

Attribute	Type	Description	Default Value	Optional
`data`		`Object`	The additional data that will be encrypted into the `nma_data` attribute of the payload in the link.	true

10.3.4.1. Dispatch Token Request Example without Encryption

```
{
  "dispatcher" : "link",
  "dispatchInformation" : {
    "data" : {
      "attributeName" : "some additional data to be included in the link
contents"
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Reg"
  }
}
```

In the example above a dispatch target is not specified, and thus the contents in the link will **not** be encrypted.

10.3.4.2. Dispatch Token Request Example with Encryption

```
{
  "dispatchTargetId" : "1c564833-e0c7-470f-8192-e036a1f60c66",
  "dispatcher" : "link",
  "dispatchInformation" : {
    "data" : {
      "attributeName" : "some additional data to be included in the link
contents"
    }
  },
  "getUafRequest" : {
    "context" : "{\"username\":\"jeff\"}",
    "op" : "Auth"
  }
}
```

In the example above a dispatch target is specified, and thus some of the contents that will be encoded in the link will be encrypted.

10.3.5. Dispatch Token Response Format

The format of the dispatch token is described in Dispatch Token Service. The dispatcher returns the link(s) in the `dispatcherInformation.response` attribute.

10.3.5.1. Dispatch Token Response Example without Encryption

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "adb7af74-893c-4cd2-a4b5-fba16ca86105",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "link",
    "response" : "https://siven.ch?dispatchTokenResponse=<dispatch contents
base64 URL encoded>"
  }
}
```

The data that is included in the link is a base 64 URL String. The base64 URL String, once decoded as a UTF-8 String, contains the following JSON:

```
{
  "nma_data" : {
    "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
    "redeem_url" : "https://fido.siven.ch/nevisfido/token/redeem/registration",
    "attributeName" : "some additional data to be included in the link
contents"
  },
  "nma_data_content_type" : "application/json",
  "nma_data_version" : "1"
}
```

The data structure in the `nma_data` attribute is the same as the one generated by the FCM dispatcher (but in this example it is not encrypted).

10.3.5.2. Dispatch Token Response Example with Encryption

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "sessionId" : "400bc2f0-0625-496b-bc47-748ab7f59fe6",
  "dispatchResult" : "dispatched",
  "dispatcherInformation" : {
    "name" : "link",
    "response" : "https://siven.ch?dispatchTokenResponse=<dispatch contents
base64 URL encoded>"
  }
}
```

The data that is included in the link is a base 64 URL String. The base64 URL String, once decoded as a UTF-8 String, contains the following JSON:

```
{
  "nma_data" : "<encrypted data>",
  "nma_data_content_type" : "application/jose",
  "nma_data_version" : "1"
}
```

The data in the `nma_data` attribute is encrypted using the standard JWE using compact serialization. The data structure is the same as the one generated by the FCM dispatcher. In this example, the encrypted contents are:

```
{
  "token" : "2e36626e-96e7-422e-9f0b-eb575f985929",
  "redeem_url" : "https://fido.siven.ch/nevisfido/token/redeem/authentication",
  "attributeName" : "some additional data to be included in the link contents"
}
```

Note that the specified information in the `data` attribute in the dispatch token request is present in the encrypted payload included in the `nma_data` attribute.

11. Plug-Ins

Experimental
The plug-in system and provided APIs are considered experimental. They are prone to change in future releases with no ensured backwards-compatibility.

Some functionality of nevisFIDO can be extended via user-provided plug-ins.

Plug-ins can be made available to nevisFIDO as `java.util.ServiceLoader` services. The nevisFIDO core framework will discover and load plug-ins, and initialize them with similarly user-defined corresponding configuration.

This chapter describes the requirements and steps involved to create plug-ins, as well as the available plug-in extension points. The plug-in API interfaces are distributed in a separate JAR artifact.

Currently the only plug-in API available is `ch.nevis.auth.fido.uaf.sdk.dispatcher.Dispatcher`, a service for dispatching tokens to third-party token-consuming services.

11.1. Using Plug-In Hooks

The implementation of a nevisFIDO plug-in involves creating a standard JDK `java.util.ServiceLoader` service, and making it available on the classpath of nevisFIDO. The plug-in API also provides hooks for letting plug-ins consume configuration from the main configuration file.

11.1.1. ServiceLoader Plug-Ins

Plug-ins use the JDK `ServiceLoader` framework. Essentially, a `ServiceLoader` plug-in consists of two elements:

- An implementation of a plug-in interface, such as the `Dispatcher` below.
- A provider configuration file located in `META-INF/services` that links the implementation class to the plug-in interface.

In addition to the standard procedure, plug-in implementations must be annotated with the `@ch.nevis.auth.fido.uaf.sdk.ConfigurationKey` annotation. This annotation is used to specify a unique identifier for the plug-in.

11.1.1.1. Example

Suppose you want to create the `Dispatcher` plug-in, a dispatcher that sends tokens via text messages. Perform the following steps to implement this dispatcher plug-in:

1. Write an implementation of `ch.nevis.auth.fido.uaf.sdk.dispatcher.Dispatcher` (a class implementing this interface). This class should contain the "business logic" that sends the token via text message.
2. Annotate your implementation with `@ch.nevis.auth.fido.uaf.sdk.ConfigurationKey` to declare your desired plug-in identifier.
3. Create a provider configuration file at `META-INF/services/ch.nevis.auth.fido.uaf.sdk.dispatcher.Dispatcher` with the following content:

```
com.example.dispatcher.MyDispatcher
```

The content of this file must be the fully qualified class name of your dispatcher implementation from the first step.

4. Pack your implementation and provider configuration file in a JAR file.
5. Make your JAR available on the Java classpath of nevisFIDO. A restart is required.

After you have completed these steps, you can configure your dispatcher in the main configuration file of nevisFIDO.

11.1.2. Configuration

Plug-in implementations can obtain configuration data from the main configuration file. Configuration data is opt-in.

A plug-in class can choose to implement the interface `ch.nevis.auth.fido.uaf.sdk.Configurable`. If the class implements `Configurable`, it must also implement the method `initialize(Map<String,Object>)`. By doing so, the plug-in class will receive a map with configuration data from `nevisfido.yml` at plug-in loading time. The configuration data is specific to the plug-in. The data is selected from the main configuration file using the plug-in identifier declared with the `@ConfigurationKey` annotation.

11.1.3. Deployment

To use a plug-in JAR with nevisFIDO, the JAR needs to be deployed to a nevisFIDO instance.

Each nevisFIDO instance has an associated instance directory. A nevisFIDO instance directory contains an (initially empty) `plugins` directory. JARs in this directory are scanned at start-up for plug-in implementations. Place your plug-in JARs in this directory to make them available to the nevisFIDO instance.

12. JavaScript Login Application

Experimental

The JavaScript Login Application is considered an experimental feature. It is prone to change in future releases with no ensured backwards compatibility.

To enable integration of out-of-band login flows, a JavaScript application library needs to be served to browser clients that interact with NEVIS Mobile Authentication. Specifically, the component involved in serving the JavaScript library is nevisLogRend.

Setting up nevisLogRend to use the JavaScript login library is an integration task; refer to the *NEVIS Mobile Authentication Concept and Integration Guide* for detailed instructions for out-of-band authentication integration.

12.1. Installation

The JavaScript library is installed using the client RPM. After installing the RPM, the JavaScript library is located under `/opt/nevisfidocl/nevislogrend/lib/nevisMobileAuthLogin.js`.

12.2. Configuration

The plug-in offers several configuration options passed at initialization in order to adjust to project specifics.

Example configuration

```
authLogin.init({debug: true});
```

Available configuration options are:

autoSubmit *(string, false)*

If set to true, the plug-in will directly submit the form after initialisation. This feature is useful if no further user action is required because the loginId of the user is already known at initialisation time.

formDomSelector *(string, "form[name=oobloginform]")*

A valid DOM selector of the login form. The plug-in "binds" itself to this form and will only react and initialise if the form is present in the HTML page.

redeemUrlDomSelector *(string, "input[name=redeemUrl]")*

A valid DOM selector of an element with a value attribute containing the URL to the redeem authentication token endpoint. This field is required in order to display the authentication QR code.

usernameDomSelector *(string, "input[name=isiwebuserid]")*

A valid DOM selector of an element with a value attribute containing the loginId of the user.

submitDomSelector *(string, "button[name=submit]")*

A valid DOM selector for the submit form element.

dispatchTargetsListDomContent *(string, '<ul id="dispatchTargets">
')*

A string containing an HTML element which will contain the selection list of dispatch targets (if more than one dispatch target exists for the loginId).

dispatchTargetDomContent *(string, '<li class="list-group-item list-group-item-action">
')*

A string containing an HTML element which will contain a dispatch target. This element will be rendered into the *dispatchTargetsListDomContent* element.

qrCodeDomContent *(string, ' ')*

A string containing an HTML element which will contain the authentication QR code.

spinnerDomContent *(string)*

A string containing an HTML element which represents a spinner indicating to the user that an operation is going on in the backend. The default value of this property is:

```
<div id="spinner" style="padding: 10px">
  <div class="d-flex justify-content-center">
    <div class="spinner-border" role="status" style="width: 8rem; height: 8rem
;">
      <span class="sr-only">Loading...</span>
    </div>
  </div>
</div>
```

statusDomContent *(string, ' ')*

A string containing an HTML element which will contain status information during the login process.

pollingIntervalMs *(integer, 500)*

Value defining the polling interval during the mobile authentication in milliseconds.

dispatchTitle *(string, "Nevis Mobile Authentication Login")*

A string containing the title of the push message sent to the user.

debug *(boolean, true)*

If set to true, additional debug information is logged to the browsers JavaScript console.

renderQRCode *(callback, function(data,parentElement))*

A function defining how to render the authentication QR code. The function takes the data to be rendered as

well as a reference to the DOM element which servers as a container. The default function provided is:

```
function (data, parentElement) {
  QRCode.toCanvas(data, {
    color: {
      dark: '#047D82',
      light: '#ffffff'
    },
    errorCorrectionLevel: 'M'
  }, function (err, canvas) {
    if (err) {
      throw err;
    }
    parentElement.appendChild(canvas);
  });
}
```

done *(callback, function(form))*

A function which will be executed if the authentication finished successfully. The default function provided is:

```
function (form) {
  window.location.replace(loginForm.getAttribute("action").replace('?login', '
'));
}
```

fail *(callback, function(form))*

A function which will be executed if the authentication fails.

```
function (form) {
  updateStatus({status: "failed"});
}
```

statusMessages *(dictionary)*

This dictionary contains status messages presentable to the user based on backend status reports.

The current plug-in implementation doesn't support multiple languages

The default dictionary provided is:

```
{
 dispatchError: "There has been an error upon dispatching the token.",
 dispatchTargetNotFound:
    "No matching target device found, please contact support.",
 dispatcherNotFound:
    "No means of sending the message found, please contact support",
 internalError: "An unspecified error occurred. Please contact support.",
 tokenRedeemed: "The token has been successfully redeemed.",
 tokenTimedOut: "The operation has timed out, please retry.",
 tokenCreated: "The login token has been created.",
 clientRegistering: "Unexpected status, this seems to be a registration.",
 clientAuthenticating: "Authenticate in your mobile device.",
 failed: "Login failed.",
 succeeded: "Login successful.",
 unknown: "Unknown token.",
 networkError: "No connection to the backend.",
 noDispatchTargets: "No devices available for authentication."
}
```

12.3. AuthState Configuration Example

The JavaScript plug-in requires some DOM elements to be present in the HTML page to be activated. These elements can be specified in an AuthState using *GuiElements*. The following sample configuration defines an *AuthGeneric* AuthState, whose only role is to define the HTML elements to be rendered. This includes the following elements:

- The HTML form, named `oobloginform` as expected by the JavaScript plug-in.
- A hidden field containing the username, whose value is assumed to have been set previously by another AuthState in the `fidocredential.loginid` attribute of the session.
- The redeem URL value, which is required by the JavaScript to render the authentication QR code to be presented to the user.
- The submit form element (a button).

```
        <AuthState name="SubmitOutOfBandParameters" class=
"ch.nevis.esauth.auth.states.standard.AuthGeneric">
            <ResultCond name="default" next="DemoOutOfBandFidoUafAuthState" />
            <ResultCond name="error" next="AuthError" />
            <ResultCond name="failed" next="AuthError" />
            <Response value="AUTH_CONTINUE">
                <Gui name="oobloginform">
                    <GuiElem name="isiwebuserid" type="text" value=
"${sess:fidocredential.loginid}" />
                    <GuiElem name="redeemUrl"    type="hidden" value=
"https://siven.ch/token/redeem/authentication" />
                    <GuiElem name="submit"       type="button" label=
"continue.button.label" value="continue" />
                </Gui>
            </Response>
        </AuthState>
```

13. Auditing

nevisFIDO currently does not audit into a separate audit log file. In order to support auditing scenarios it is

recommended using the logback configuration with fine-grained log levels (for example *DEBUG*).

14. Known Limitations

1. nevisFIDO currently does not inform FIDO clients about deregistered credentials, if the deregistration process was not initiated by the client itself.
2. nevisFIDO does not support the *Registration Counter* (`RegCounter`) anti-fraud measure outlined in Anti-Fraud Signals.
3. With the current dispatch target model in nevisIDM, it is not possible to identify which (FIDO) credentials belong to which device.

Glossary

Throughout the document, several FIDO terms appear. This section explains those most often used.

You can find the complete FIDO Glossary here FIDO Glossary.

FIDO UAF Authenticator / FIDO Authenticator / authenticator

: According to the official FIDO documentation, the *FIDO UAF Authenticator* is "a secure entity, connected to or housed within the FIDO user devices, that can create key material associated to the Relying Party. The key can then be used to participate in FIDO UAF strong authentication protocols. For example, the FIDO UAF Authenticator can provide a response to a cryptographic challenge using the key material thus authenticating itself to the Relying Party".

Relying Party / relying party

: The *Relying Party* is "a website or other entity that uses a FIDO protocol to directly authenticate users". In the nevisFIDO context, the equivalent to the relying party will be nevisAuth.

Attestation

: In the FIDO context, an *Attestation* is "how Authenticators make claims to a Relying Party that the keys they generate, and/or certain measurements they report, originate from genuine devices with certified characteristics".

Authenticator Attestation ID / AAID

: The *Authenticator Attestation ID (AAID)* is a "unique identifier assigned to a model, class or batch of FIDO Authenticators that all share the same characteristics, and which a Relying Party can use to look up an Attestation Public Key and Authenticator Metadata for the device".

Appendix A: Crypto support

A.1. Supported Public Key Formats

nevisFIDO stores the public keys of the FIDO UAF authenticators as sent by the FIDO client during the FIDO UAF registration. nevisFIDO supports the following public key algorithms:

- `ALG_KEY_ECC_X962_DER`: ASN.1 DER [ITU-X690-2008] encoded ANSI X.9.62 formatted SubjectPublicKeyInfo [RFC5480].
- `ALG_KEY_ECC_X962_RAW`: Raw ANSI X9.62 formatted Elliptic Curve public key.
- `ALG_KEY_RSA_2048_DER`: ASN.1 DER [ITU-X690-2008] encoded 2048-bit RSA [RFC3447] public key [RFC4055].
- `ALG_KEY_RSA_2048_RAW`: Raw encoded 2048-bit RSA public key [RFC3447].

See the Public Key Representation Formats section of the FIDO UAF specification for details.

A.2. Supported Authentication Algorithms

During registration and authentication, the FIDO UAF client sends attestations to nevisFIDO. nevisFIDO supports the following algorithms when validating the signature of these attestations:

- `ALG_SIGN_SECP256R1_ECDSA_SHA256_DER`: DER [ITU-X690-2008] encoded ECDSA signature [RFC5480] on the NIST secp256r1 curve.
- `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW`: ECDSA-ANSI encoded ECDSA signature [RFC5480] on the NIST secp256r1 curve.
- `ALG_SIGN_SECP256K1_ECDSA_SHA256_DER`: DER [ITU-X690-2008] encoded ECDSA signature [RFC5480] on the secp256k1 curve.
- `ALG_SIGN_SECP256K1_ECDSA_SHA256_RAW`: ECDSA encoded ECDSA signature on the secp256k1 curve.
- `ALG_SIGN_RSASSA_PSS_SHA256_DER`: DER [ITU-X690-2008] encoded OCTET STRING containing the RSASSA-PSS [RFC3447] signature.
- `ALG_SIGN_RSASSA_PSS_SHA256_RAW`: RAW encoded RSASSA-PSS [RFC3447] signature [RFC4055] [RFC4056].
- `ALG_SIGN_RSA_EMSA_PKCS1_SHA256_DER`: DER [ITU-X690-2008] encoded OCTET STRING containing the EMSA-PKCS1-v1_5 signature [RFC3447].
- `ALG_SIGN_RSA_EMSA_PKCS1_SHA256_RAW`: RAW encoded EMSA-PKCS1-v1_5 signature [RFC3447].

See the Authentication Algorithms section of the FIDO UAF specification for details.

A.3. Supported Encryption Methods by the FCM (Firebase Cloud Messaging) Dispatcher

The FCM (Firebase Cloud Messaging) Dispatcher encrypts the tokens sent through the Firebase Cloud Messaging push service. The following encryption algorithms are supported:

- `RSA-OAEP-256` with encryption method `A256CBC_HS512`
  - RSA-OAEP-256: RSAES using Optimal Asymmetric Encryption Padding with SHA-256 hash function.
  - A256CBC_HS512: AES_256_CBC_HMAC_SHA_512 authenticated encryption using a 512 bit.
- `ECDH-ES+A256KW` with encryption method `A256CBC_HS512`
  - ECDH-ES+A256KW: Elliptic Curve Diffie-Hellman Ephemeral Static key agreement, where the agreed-upon key is used to wrap the Content Encryption Key (CEK) with the A256KW function.
  - A256CBC_HS512: AES_256_CBC_HMAC_SHA_512 authenticated encryption using a 512 bit.

See the JSON Web Algorithm (JWA) specification for details.

A.4. Supported Signature Methods to Modify Dispatch Targets

To modify dispatch targets (see the Modify Dispatch Target HTTP API), the client must sign the modification payload. nevisFIDO supports the following JWS algorithms:

- `RS256`: RSASSA-PKCS1-v1_5 using SHA-256.
- `RS384`: RSASSA-PKCS1-v1_5 using SHA-384.
- `RS512`: RSASSA-PKCS1-v1_5 using SHA-512.
- `PS256`: RSASSA-PSS using SHA-256 and MGF1 with SHA-256.
- `PS384`: RSASSA-PSS using SHA-384 and MGF1 with SHA-384.
- `PS512`: RSASSA-PSS using SHA-512 and MGF1 with SHA-512.
- `ES256`: ECDSA using P-256 and SHA-256
- `ES384`: ECDSA using P-384 and SHA-384
- `ES512`: ECDSA using P-521 and SHA-512

See the JSON Web Algorithm (JWA) specification for details.

[1] Check the Kubernetes documentation for more information on the configuration of the liveness and readiness probes.