Nymi API C Interface Guide

Nymi Enterprise Edition
v2.0
2020-09-18

Contents

Preface

Nymi™ provides periodic revisions to Nymi Enterprise Edition™. Therefore, some functionality that is described in this document might not apply to all currently supported Nymi products. The product release notes provide the most up to date information.

Purpose

This document is part of the `Nymi Enterprise Edition` documentation suite.

This document provides Nymi developers with an alternative way to utilize the functionality of the Nymi SDK, over a WebSocket connection managed by a web-based or other applications.

Audience

This guide provides information to Developers.

Revision history

The following table outlines the revision history for this document.

Table 1: Revision history

Version	Date	Revision history
01	December 20, 2019	This guide is reissued due to document version update. There are no content changes since NEE 2.6.0.
02	September 18, 2020	Reissued to clarify behaviour for the presence operation, presence notification and bluetooth notifications.

Related documentation

- **Nymi Enterprise Server Deployment Guide**

  This document provides an overview of the components and the steps that are required to deploy the NES.
- **Nymi Enterprise Edition Administration Guide**

  This document provides information about how to use the `NES Administrator Console` to manage a `Nymi Enterprise Edition` system. This document describes how to set up, use and manage the Nymi Band™, and how to use the `Nymi Band Application`.

- **Nymi Enterprise Edition Release Notes**

  This document provides supplemental information about Nymi Enterprise Edition, including new features, limitations, and known issues with the Nymi Band, NES, the Nymi Band Application, and the NES Administrator Console.
- **Nymi Enterprise Edition Troubleshooting Guide**

  This document provides information about how to troubleshoot issues and the error messages that you might experience with the `NES Administrator Console`, the Nymi Enterprise Server deployment, the Nymi Band, and the `Nymi Band Application`.

How to get product help

If the Nymi software or hardware does not function as described in this document, contact your administrator for immediate support. Alternatively, you can submit a support ticket to Nymi, or email support@nymi.com

How to provide documentation feedback

Feedback helps Nymi to improve the accuracy, organization, and overall quality of the documentation suite. You can submit feedback by using support@nymi.com

Nymi API for C Interface Overview

The Nymi SDK provides Developers with libraries, APIs, sample code and documentation for building a Nymi-enabled Application (NEA). The Nymi API for C Interface architecture is part of the Nymi SDK.

Nymi SDK Overview

Nymi API exposes a very simple C interface that provides the following benefits:

- Minimizes the complexity of the integration and allows bidirectional communication by exchanging messages in JSON format.
- Supports the use of foreign function interfaces (FFIs), which enables developers to use the SDK with any language or environment that supports linking with C libraries.

The `Nymi API C Interface` contains the following components.

- Nymi Runtime—Facilitates communication between an NEA and Nymi Bands. Install the Nymi Runtime on the developer machine and on any machine where the NEA will run.
- Nymi API (NAPI)—Provides developers with the ability to interface with the Nymi Runtime and communicate with Nymi Bands.

SDK Package Contents

The Nymi SDK package contains the following artifacts:

- nymi_api.dll
- sample apps (C/C++ and C#)
- BleDriver_xx.msi
- Nymi Runtime Installer

Nymi API for C Interface Architecture

Nymi API supports enterprise integration and deployment of features that are available in `Nymi Enterprise Edition`.

Nymi has created APIs that supports the enterprise integration and development of features that are available in the Nymi Enterprise Edition software.

The `Nymi API C Interface` supports the `Nymi Bluetooth Endpoint` and also offers NFC support. For example, applications utilizing this functionality allow the Nymi Band to tap an NFC device, which sends an intent event message to the `Nymi Agent`. To enable NFC support, the NFC reader must be connected to a terminal with a compatible version of NBE installed.

For information about deploying in a Citrix or RDP environment, see the *Nymi Enterprise Edition Deployment Supplement Guide*.

Development Tools

To develop NEAs on a Windows platform, you can use one of the following tools.

- Any Microsoft-supported version of Visual Studio.
- Visual Studio Code (or any other code editor).

For C, C++, and C#, Nymi recommends that you use Visual Studio 2017.

Supported Platforms

`Nymi API C Interface` supports the following platforms.

- Microsoft Windows 10, 64-bit
- Microsoft Windows 7, 32-bit and 64-bit

Supported NFC Readers

The Nymi API C Interface supports multiple NFC readers. A list of supported NFC Readers is found in the Hardware requirements section of the Nymi Enterprise Edition Administration Guide 2.6.0.

The `Nymi Bluetooth Endpoint` monitors all attached and supported NFC readers and forwards events from all NFC readers (there is no preference between readers).

The NFC reader is connected to the user's terminal where the Nymi Bluetooth Endpoint is installed and are automatically detected by the `Nymi Bluetooth Endpoint`.

Nymi API for C Interface Sample Application

Nymi offers you a sample application that demonstrates some of the key functionality of the Nymi solution.

The sample applications are located within the package at: *nymi-sdk/windows/samplesApps*

Installing the Nymi API C Interface

The `Nymi API C Interface` development package includes:

- `Nymi API C Interface` package, which includes the API DLL and supporting documentation
- `Nymi Runtime` installer

Perform the following steps to set up the `Nymi API C Interface` and the `Nymi Runtime`.

1. Extract the `Nymi API C Interface` package to the development machine.
2. Copy the *nymi_api.dll* file from the *..\nymi-sdk\windows\x86_64* directory to the Visual Studio working directory.

   Note: In a remote environment where the NEA is running on a different machine than the runtime, Visual c++ 2013 and 2015 redistributables must be installed.
3. From the *..\nymi-sdk\windows\setup* folder, run the *Nymi Runtime Installer 5.4.x.y.exe* file.

   Where *x.y* is the version number of the `Nymi API C Interface` package.

   Note: `Nymi Runtime` upgrades might prompt you to reboot the machine to complete the upgrade process, but it is not necessary.

When the installation completes, the following services are started on the machine.

- `Nymi Agent`—Delegates requests and responses between the NEA and the Nymi Band.
- `Nymi Bluetooth Endpoint` —Establishes and secures the Bluetooth connection to the Nymi Band.

Nymi Component Configuration

In order to create an environment that can utilize the services contained in the `Nymi API C Interface`, specify the location of the `Nymi Agent` so that the `Nymi Bluetooth Endpoint` can connect to it.

The `Nymi API WebSocket Interface` is installed on each RDP client. The `Nymi API WebSocket Interface` service on each RDP client communicates with the `Nymi Agent` service, which is installed on a separate host, on websocket port 9120.

1. Navigate to the location where the *nbe.exe* file is installed.
2. Open the *C:\\Nymi\Bluetooth_Endpoint\nbe.toml* file.
3. Update the location of the `Nymi Agent`
   - agent_url = 'ws://<FQDN>:9120/socket/websocket'
4. Optionally, you can set the location of the `Nymi Bluetooth Endpoint` by configuring the *endpoint_id* parameter.
   - endpoint_id = "<unique ID>"

   By configuring the *endpoint_id* parameter, you need to use the subscribe operation. For more information about the subscribe operation, see the *Request Operations* section in this guide.

   Note: In centralized deployments where the `Nymi Agent` is running on a different computer from the NBE and NEA, the server running the `Nymi Agent` must be able to receive incoming WebSocket connections on TCP port 9120. Please ensure that port 9120 is open in the firewall on the server running the `Nymi Agent`.

Creating NEAs with Nymi API

Customer and partner developers can use the Nymi API C Interface to develop Nymi-enabled Applications (NEAs) in programning languages, such as Java or C#. The API is written in JSON. This chapter provides information about the supported operations.

To deploy an NEA, developers must install the `Nymi Runtime` on each terminal where the NEA runs. The Nymi Runtime includes the following components: `Nymi Bluetooth Endpoint`, and `Nymi Agent`.

Note: In this document, the use of device refers to the Nymi Band.

Overview of NAPI message handling

NAPI provides two function calls to handle messaging.

- *request()*—Used to send messages to NAPI.
- *update()*—Used to retrieve messages, such as response messages and system notifications from NAPI.

Call Concurrency

NAPI has two FIFO (First-In, First-Out) message queues.

- Device queues—One message queue exists for each Nymi Band. When NAPI receives a device-related message, NAPI dispatches the message to the appropriate device message queue, in the order that the message is received. NAPI might dispatch messages to a device before dispatching messages that have been queued longer, to another device.
- Non-device queue—One global message queue that stores messages that are not related to a device operation, for example, the response for an *init()* call. NAPI dispatches non-device related messages to the queue in the order that the messages are received.

Request and Response

The *request()* and *update()* calls are handled differently in memory.

- *request()*—NEA supplies the request message in a memory buffer. Before the call returns, Nymi API creates a copy of the message.
- *update()*—After the function returns, Nymi API expects the NEA to copy the response message out of the memory address provided by the *update()* call, before calling the *update()* function again.

Nymi API operations

A *request()* call submits a message to Nymi API. Nymi API performs the operation that is contained in the message. The result of the operation is a response message. Use the *update* function to retrieve the response message.

For information about how to use the *update* function, see *The update function* section.

Request Operations

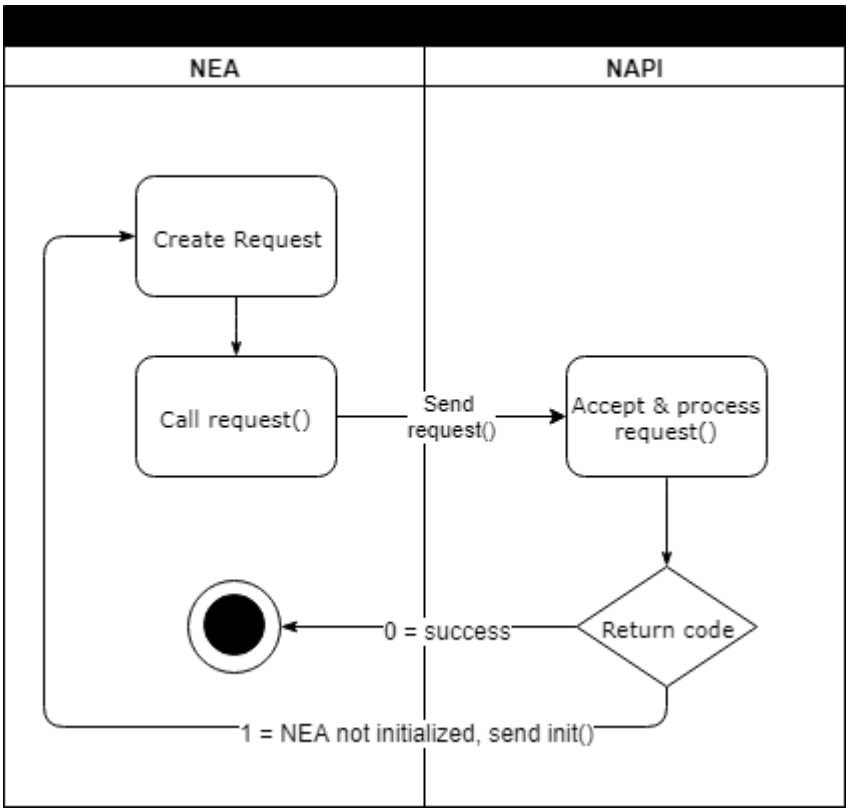
The `Nymi API C Interface` contains request operations.

A *request()* call submits a message to Nymi API. Nymi API performs the operation that is contained in the message. The result of the operation is a response message. Use the update function to retrieve the response message. For information about how to use the update function, see The *update function* section.

The declaration for the request function in C is as follows.

```
typedef int (*WINAPI REQUEST_FUNC_POINTER)(const char*);
REQUEST_FUNC_POINTER request = NULL;
```

The following diagram shows the *request()* call and message handling workflow for device operations.



`Figure 1: Request function workflow`

1. Create the request in a memory buffer and pass the request to Nymi API.
2. Nymi API creates a copy of the request message.

3. Nymi API initiates the requested operation.

   The *request()* call returns 0 when Nymi API accepts the message and returns a 1 when NEA the has not been initialized. The NEA must run the *init* operation before Nymi API can accept any messages other than *init*.

The request message is a null-terminated string containing a JSON object with the following key-value pairs:

```
{
        "operation": "operation_name",
        "exchange": "exchange_string",
           "payload": {
           "property_name": "property_value",
           "property_name1": "property_value1"
           …
           "property_nameX": "property_valueX"
           }
    }
```

where:

- *operation_name* defines the operation for Nymi API to perform. For example, *init*, *assert_identity*, and *lookup*.

subscribe operation

The `subscribe_endpoint` operation allows a Nymi-enabled Applicationn to change the Nymi Bluetooth Endpoint to which it is subscribed..

The subscribe_endpoint operation allows a Nymi-enabled Application to change the Nymi Bluetooth Endpoint to which it is subscribed.

By default, each Nymi-enabled Application is matched to it's local endpoint based on the IP address of the workstation. In most deployments, the Nymi-enabled Application and endpoint are correctly matched by default, and connect automatically.

`subscribe_endpoint` request operations appear in the following format:

In central deployments, certain network configurations, such as workstations that have multiple network interfaces, may interfere with the automatic matching of the NEA and NBE. In these cases, the subscribe operation must be used by the NEA to communication to which workstation it wants to connect.

```
{
"operation": "subscribe_endpoint",
"exchange":"exchange_value",
"payload": {
"endpoint_id": "bar"
}
}
```

where:

- *operation* is the subscribe_endpoint.
- *exchange* is any value and is used to match the response to the request.

payload:

- *endpoint_id* is based on the endpoint IP address.

The subscribe_endpoint operation returns status codes only, no errors are returned. The following table displays possible status codes:

```
{
“Operation”: “subscribe_endpoint”,
"exchange":"exchange_value",
“payload”: {}
“status”: 0,
“error”: {}
}
```

A Nymi-enabled Application can only be subscribed to one endpoint at any given time. When a subscribe operation is requested, the NEA is automatically unsubscribed from the endpoint it was previously subscribed to. If any Nymi Bands were present on that endpoint, they will become absent, and the NEA will receive corresponding presence update notifications. The NEA will then receive a Bluetooth status notification. If the requested NBE has connected successfully and is in a ready state, the NEA will receive a ble_ready notification, followed by presence update notifications for any present bands on that endpoint. Otherwise, the NEA will receive an error message. See *Bluetooth Notifications* for more information about possible error messages.

Note: The NEA will remain subscribed to the requested endpoint_id even if it is not able to connect to that NBE. If the NBE becomes ready at a later time (for example, that workstation is powered on), the NEA will receive a ble_ready message at that time.

init operation

The *init* operation initializes NAPI, configures communication channels between components, and performs certificate enrollment when required. Ensure that *init* is the first operation that is requested by the NEA. When the *init* operation succeeds, it is not necessary to call *init* again.

Initialization Options

There are three ways to call the *init* operation when initializing with certificate enrollment.

- nea_name
- nea_name + nes_url + token
- nea_name + nes_url + token + otp

JSON Object Format

Define the JSON payload for the *init* in the following format.

```
{
  "operation": "init",
  "exchange": "exchange_value",
  "payload": {
     "nea_name": "name_of_application",
     "nes_url": "https_url_to_nes",
     "token": "token",
     "otp": "one_time_password",
     "log_path": "path",
     "url": "ws://agent_server:9120/socket/websocket",
   }
}
```

where:

- *name_of_application* is the name that you assign to the NEA and is always required. The NES active group policy configuration influences the name that you can specify, in the following way:
  - When **`Manual OTP mode`** is enabled, you must specify the name as `NEAs`.
  - When **`Manual OTP mode`** is not enabled, you can assign any name to the NEA.

  Contact the NES Administrator to determine the active group policy configuration settings.
- *nes_url* field is the URL for the NES website application. You require this parameter in the first *init* call. The format of the URL is https_url_to_nes
- *token* is an HTTP Bearer token that NES uses to authenticate the NEA user or computer. This parameter is optional. If you will use this parameter, you must specify it in the first *init* call. Obtain the token as described in the *Appendix*.
- *one_time_password* is the OTP that provides the NEA with the ability to generate the NEA certificate. Include *one_time_password* in the payload when Manual OTP mode is configured in the active group policy in NES. You require this parameter in the first *init* call. When you define this parameter, you must also define the *https_url_to_nes* and *token* parameters.
- *path* is the log file path on the development machine. If you do not specify the path property, the NEA uses the default log path, which is your current working directory.
- *agent_server* specifies the hostname of the machine that runs the `Nymi Agent` service.

`Example`

The following code block provides an example of a JSON object that instructs NAPI to initialize the NEA that requires an OTP to retrieve a certificate.

```
{
   "operation": "init",
   "exchange": "rAndOm_IdeNtifyiNG_StrING_1211",
   "payload": {
        "nea_name": "NEAs",
        "nes_url": "https://server-2.nymi.lab/nes",
        "token": "eyJVc2VyVG9rZW5TdHJpbmciOiJMbk..",
        "otp": "4C82F6CF3ABED723",
```

```
                "url": "ws://agent.nymi.com:9120/socket/websocket"
            }
        }
```

Results

A successful *init* operation produces a response with the following properties.

```
{
   "operation": "init",
   "exchange": "rAndOm_IdeNtifyiNG_StrING_1211",
   "payload": {}
      "status": 0,
      "error": {}
}
```

An unsuccessful *init* operation generates a non-zero status.

The following table summarizes the status codes that can appear, and the payload properties that you require for a subsequent *init* call.

Table 2: Init Status Codes

Status code	Payload properties for subsequent *init* call
0	Operation completed successfully with the defined payload. The system is initialized. Additional calls to *init* are not required.
11xx	Operation completed successfully with the defined payload. When a request other than *init* is sent before the system is initialized, the system returns a status code 1100. If the system was already initialized, but a request for *init* was sent, the system returns a status code 1110.
8000	Payload is missing the *token* and *nes_url* property definitions. Call *init* again and include the *token* and *nes_url* properties, in addition to the *nea_name*.
8100	Payload includes the *token* and *nes_url* but is missing the OTP property definition. Call *init* again and include the *otp* in the payload, in addition to the *nea_name*, *token* and *nes_url* properties.
9000	There was an issue with the certificate from NES. Contact the NES Administrator for assistance.

The following flowchart provides an overview of how you can use NAPI responses to an *init* call, to determine the properties that you need to include in the payload file.

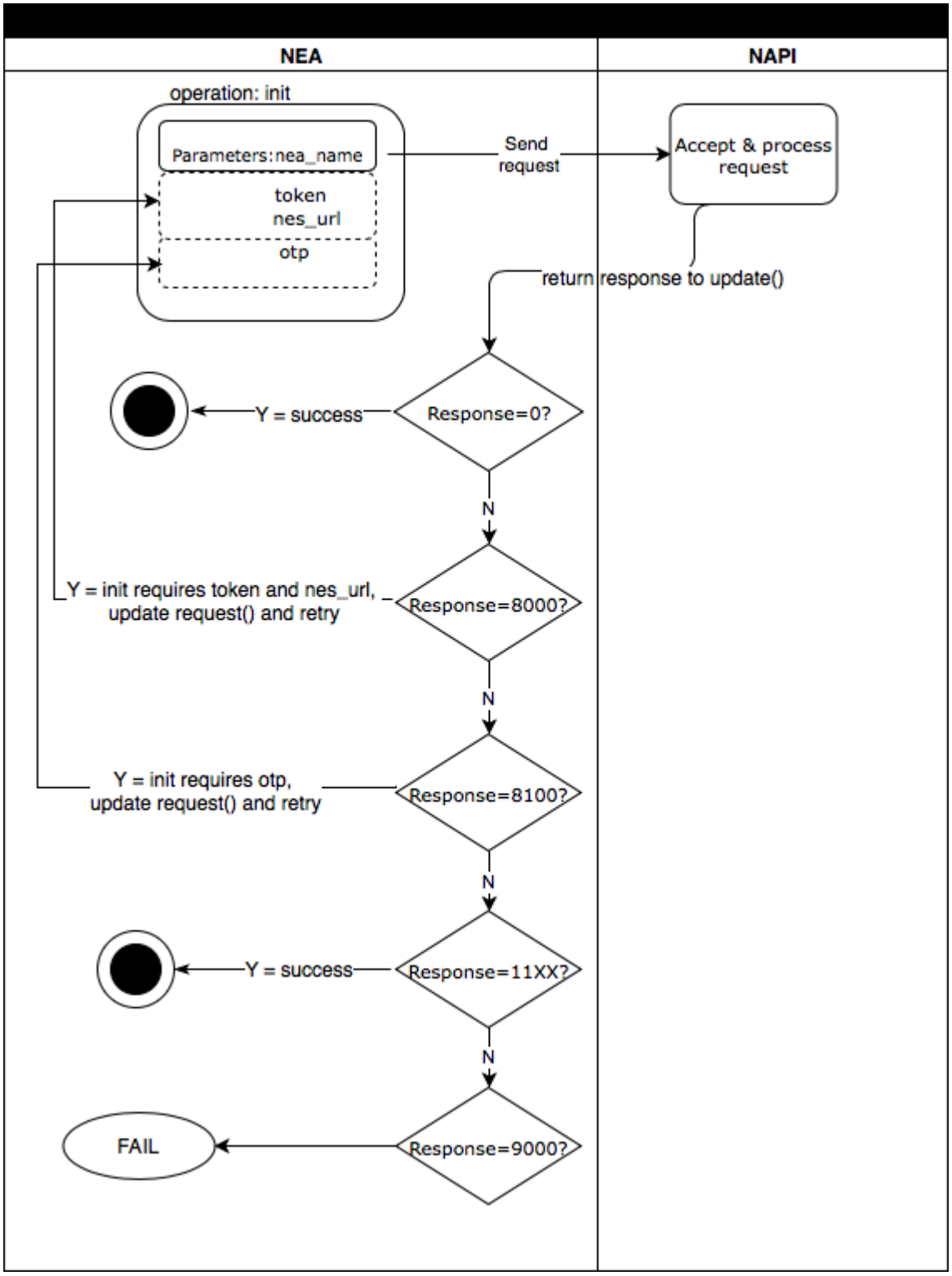


Figure 2: NAPI response calls to init

lookup

An NEA requires the device ID of a Nymi Band to communicate with the Nymi Band. You can retrieve the device ID of a Nymi Band from NES by using the *lookup* operation.

Use the *lookup* operation to determine the following values:

- Device ID (Device operations require that you specify the Nymi Band (or device) ID value that appears in the response.
- NfcUID of the Nymi Band
- Domain and name of the user.
- User status in Active Directory (AD). The AD status for a user appears in the response when user status check is enabled in NES. The following table summarizes the possible user statuses.

Table 3: AD user statuses

User Status	Definition
Active	User account is enabled.
NotExist	User account was deleted from AD.
Inactive	User account is disabled.
Locked	User account is locked. This status can appear with Active and Password Expired.
PasswordExpired	User account has an expired password. This status can appear with Active and Locked.

By default, NES disables support for user status checks in AD. Contact the NES administrator to enable AD user status checking, and optionally the checking interval in the .

JSON Object Format

Define the *payload* JSON object for the *lookup* command in the following format.

```
{
"operation": "lookup",
"exchange": "exchange_value",
   "payload":
   {
   "nes_url": "https_url_to_nes",
   "query": "query_JSON",
   "lookup_keys": "key_JSON"
   }
  }
```

where:

- *nes_url* field is optional if not provided it uses what is configured for the Nymi Agent. See the *Configuration Overview* .

- *query* field is a JSON object that defines the query values. Acceptable values include *NfcUID*, *Domain* and *Username*, and *NymiBandID*.
- *lookup_keys* field is a JSON array that contains a list of values that you want to appear in the response. Supported values include *NfcUID*, *Domain* and *Username*, *NymiBandID*, and *UserStatus*.

Note: The property names *Domain* and *Username* are case-sensitive.

Example 1

The following code block provides an example of a JSON object that instructs NAPI to provide the NfcUID of a device and the user status for a user named *JSmith* in the *MyCorpDomain* domain.

```
{
   "operation": "lookup",
   "exchange": "rAndOm_IdeNtifyiNG_StrING_1218",
   "payload": {
    "nes_url": "https://nes.nymi.com/nes/",
    "query": {
        "Domain":"MyCorpDomain",
        "Username": "JSmith"
     }
   "lookup_keys": ["NfcUID", "UserStatus"]
   }
}
```

Results 1

A successful *lookup* operation produces a response with the following properties.

In this example, the check user status in AD option is enabled in NES, as a result, the response includes the *UserStatus* property.

```
{
  "operation": "lookup",
  "exchange":"rAndOm_IdeNtifyiNG_StrING_1218",
  "payload": {
      "lookup_values":{"NfcUID": "1234xyz", "UserStatus":"Active|PasswordExpired"},
  },
  "status": "0",
  "error: {}
}
```

Example 2

The following code block provides an example of a JSON object that instructs NAPI to provide the NfcUID of a device with Nymi Band (or device) ID *"C2:FA:D7:F0:D7:96"*.

```
{
```

```
    "operation": "lookup",
    "exchange": "rAndOm_IdeNtifyiNG_StrING_1218",
    "payload": {
        "nes_url": "https://nes.nymi.com/nes/",
        "query": {
        "NymiBandID": "C2:FA:D7:F0:D7:96"
        }
        "lookup_keys": ["NfcUID"]
    }
}
```

Results 2

A successful *lookup* operation produces a response with the following properties.

```
{
  "operation": "lookup",
  "exchange":"rAndOm_IdeNtifyiNG_StrING_1218",
  "payload": {
      "lookup_values": {"NfcUID": "1234xyz"},
   },
  "status": "0",
  "error: {}
}
```

assert_identity

The *assert_identity* operation provides an NEA with the ability to confirm that a Nymi Band that is assigned to a specific user is authenticated and within Bluetooth range.

The *assert_identity* command completes a cryptographic handshake with the Nymi Band and verifies user/band identity.

Note: The Nymi Band must be in an authenticated state when you call the *assert_identity* operation.

Define the *assert_identity* JSON object in the following format.

```
{
  "operation": "assert_identity",
  "exchange": "exchange_value",
     "payload": {
         "nes_url": "https_url_to_nes",
         "device": "NymiBandID",
         "assert_type": "assert_user"
     }
}
```

where:

- *nes_url* field is optional if not provided it uses what is configured for the Nymi Agent. See the *Configuration Overview* .
- *NymiBandID* is the Nymi Band (or device) ID value that is returned in the *lookup* result.

Example

The following code block provides an example of a JSON object that instructs NAPI to assert the identity of the user with device ID *C2:FA:D7:F0:D7:96*.

```
{
  "operation": "assert_identity",
  "exchange": "rAndOm_IdeNtifyiNG_StrING_5555",
  "payload": {
      "nes_url": "http://nes.nymi.com/nes/",
      "device": "C2:FA:D7:F0:D7:96",
      "assert_type": " assert_user "
  }
}
```

assert_identity Response

The *UserStatus* property is an optional property. The *UserStatus* is stored in the Active Directory (AD).

If the *UserStatus* option is set in the NES console in the *Policies* > *Active Directory* page, the Active Directory status appears in the assert_identity response. If the option is not set, it does not return in the response.

The *UserStatus* option has the following possible values:

User Status	Definition
Active	User account is enabled.
NotExist	User account was deleted from AD.
Inactive	User account is disabled.
Locked	User account is locked. This status can appear with Active and Password Expired.
PasswordExpired	User account has an expired password. This status can appear with Active and Locked.

The last three properties can be combined into a coma separated list.

By default, NES disables support for user status checks in AD. Contact the NES administrator to enable AD user status checking, and optionally the checking interval in the .

A successful *assert_identity* operation produces a response with the following properties.

```
{
"operation": "assert_identity",
```

```
"exchange":"rAndOm_IdeNtifyiNG_StrING_5555",
"payload": {
"Username": "Jsmith",
"Domain": "Corp"
"UserStatus": "Active"

},
"status": "0",
"error: {}
}
```

presence update

Using the presence update request, you can retrieve the current state of the Nymi Band. Presence update requests are non transactional. The presence request has no response and a presence response is not tied to a specific request.

When a presence update request is sent, the system will replay the last presence update received. When a presence state changes you will receive automatic notifications. For information about these notifications, see *presence update notification.*

Presence is relative to an endpoint (the response indicates if the Nymi Band is in range of the NEA). A Nymi Band can be present on some endpoints, but absent on others. If the presence state is false the presence state returns as `absent`.

JSON Object Format

Define the *presence* request JSON object in the following format.

```
{
"operation": "presence",
"exchange":"exchange_value",
"payload": {
"device": device
}
}
```

Table 4: Presence Payload

Properties	Value	Description
Device	device	The Nymi Band MAC address.
State		The value named `state` has a string value.

Properties	Value	Description
	absent	The state is not detected. A Nymi Band that has not been reported on* should be considered the same as a Nymi Band that has most recently been reported `absent`. A Nymi Band that has an `absent` state may be unheard from for a certain length of time. **Note:** * reported on refers to a) The Nymi Band is connected via BLE and is present. b) It has sent a BLE advertisement to the endpoint within the last 30 seconds.
	unauthenticated	Nymi Band is not authenticated (may or may not have authenticators enrolled). A Nymi Band that is not authenticated may be on-body and `unauthenticated` or is being charged.
	weak	Nymi Band is in an authenticated state. The advertisement authentication code is not verified.

device version

Using the device_version request, you can retrieve hardware and firmware version of the Nymi Band. The Nymi Band can be in any state when the band label request is sent.

JSON Object Format

Define the *presence* request JSON object in the following format.

```
Request:
{
"operation":"get_device_version",
"payload":{
"device": "00:00:00:00:00:01"
},
"exchange":"ID"
}
```

Device Version Response

The).

Field	Definition
fw_version	U
hw_version	AD.

Field	Definition
exchange	U

Response Messages and Notifications

By default, a response message contains the `operation` value, the payload, and the `status` value of the request.

- Responses are messages that are generated as a result of the operations previously submitted to NAPI.
- Notifications are system-generated messages that provide information about state changes in the environment. Notifications are not generated in response to a request made by a function call.
- When the Presence of a Nymi Band changes, for example, when the `Nymi Agent` authenticates a Nymi Band.
- When a `Nymi Runtime` error occurs.

The *update* function retrieves the notifications and responses from memory. Before the response appears in the update queue, the system requires time to process the request and generate the response. Call the *update* function on a single thread, to maintain one centralized place that handles all *update* responses.

IMPORTANT: In large environments, call update frequently to avoid the loss of responses and notifications.

Response Messages and Notifications

- Responses are messages that are generated as a result of the operations previously submitted to NAPI.
- Notifications are system-generated messages that provide information about state changes in the environment. Notifications are not generated in response to a request made by a function call.
- When the Presence of a Nymi Band changes, for example, when the `Nymi Agent` authenticates a Nymi Band.
- When a `Nymi Runtime` error occurs.

The *update* function retrieves the notifications and responses from memory. Before the response appears in the update queue, the system requires time to process the request and generate the response. Call the *update* function on a single thread, to maintain one centralized place that handles all *update* responses.

IMPORTANT: In large environments, call update frequently to avoid the loss of responses and notifications.

Exchange Message

NAPI sends response messages and notifications to a memory buffer. There is only one response queue, and requests are not tracked against their original threads.

Define an exchange value in the *request_obj* to match the requests that are sent from various threads to the responses that are received on the *update* thread.

A response message appears in the following format:

```
{
"operation":"operation_value",
"payload": {
"property_name": "property_value",
"property_name1": "property_value1",
…
"property_nameX": "property_valueX"
}
"status": 0 or error_code,
"error": {
"error_description": "error_description",
"error_specifics": "specific error description"
}
}
```

Consider the following:

- *operation* always appears in the response and the value depends on the reason for the response.
  - For a request response, the *operation_value* matches the *operation_value* in the request.
  - For a notification response that is the result of an error, the *operation_value* is *error*.
- *payload* always appears in the response. If the *payload* does not contain properties or the response results in an error, the *payload* will appear empty. For example, `"payload": {}`.
- *status* is 0 when the operation is successful and an integer value that is greater than zero when the operation fails.
- *error* always appears in the response and the value depends on the reason for the response.
  - If the response is the result of a successful request, error is empty. For example, `"error": {}`.
  - If the response is the result of a failed request or error notification, status displays an error code, and error contains descriptive information about the failure. See *Error Handling* for more information.

update function

Use the *update* function to retrieve responses for requests and system notifications from NAPI.

The declaration for the update function is as follows:

```
typedef const char* (WINAPI* UPDATE_FUNC_POINTER)(int timeout_ms);
UPDATE_FUNC_POINTER update = NULL;
```

Where *timeout_ms* is an integer value that represents the number of milliseconds (ms) that the update function waits for a response before timing out.

Ensure that you do not call *update* simultaneously on two threads.

Results

The *update* function returns a pointer to a JSON message as an UTF-8 string. The string has one of the following values:

- Empty string, when a timeout occurs
- Valid JSON string

initialization error notifications

After initialization, NAPI might disconnect from the `Nymi Agent`, which results in update retrieving an error notification similar to the following example.

```
{
  "operation": "error",
  "exchange": null,
  "payload": {},
  "status": 4000,
  "error": {
      "error_description": "Nymi Agent missing.",
      "error_specifics":""
  }
}
```

When a disconnect occurs, NAPI automatically attempts to reconnect to `Nymi Agent`. Any requests that an NEA performs will fail until it retrieves a `reconnection` notification.

A reconnection notification appears similar to the following:

```
  {
  "operation": "reconnection",
  "exchange": "null",
  "payload": {},
  "status": 0,
  "error": {}
  }
```

Presence Notifications

When NAPI detects a change in Nymi Band presence, NAPI generates a presence notification.

The *update* calls that you perform after you perform the *init* operation retrieve a sequence of presence notifications, one for each present Nymi Band (if any Nymi Bands are present within range. Presence updates are non transactional. The system will return any changes to presence.

It is recommended that you develop a method for your application that tracks when the Nymi Bands come in and out of range.

Presence notifications appear in the following format:

```
{
"operation":"presence",
"exchange":null,
"status":0,"
payload":{
        "device": device,
        "proximity":"proximity value",
        "state":state
},
"error":{}
}
```

where:

- *proximity_value* is 0 when the Nymi Band is present and 4 when the Nymi Band is absent.
- *state_value* is one of the values in the following table.

If the payload contains only the device, no response is returned for this operation. A notification is returned, which is not tied to any request and does not contain any values.

Table 5: State values for presence notifications

State Value	Definition
Absent	Nymi Band is not reachable, and the `Nymi Agent` cannot communicate with it. There may be a delay in the Nymi Agent hearing from the Nymi Band for periods of time.
Unauthenticated	Nymi Band is enrolled and worn on the user's wrist but not authenticated.
Weak	Nymi Band is in an authenticated state.

Reasons for Nymi Band absence include:

- Nymi Band has been removed from the body.
- Nymi Band has not communicated with the `Nymi Agent` for at least 30 seconds.
- Nymi Band has not been within the range of the Bluetooth Adapter for at least 30 seconds.

bluetooth notifications

Nymi Bluetooth Endpoint is a client service that communicates with the Bluetooth Adapter. Bluetooth notifications for Bluetooth Adapter status are non transactional.

The Bluetooth Adapter communicates to the Nymi Band. Each time that a Bluetooth Adapter becomes available, the *update* function retrieves a notification in the following format.

```
{
  "operation": "ble_ready",
```

```
  "exchange": null,
  "status": 0,
  "payload": {},
  "error ": {}
}
```

If a Bluetooth Adapter becomes unavailable, the *update* function retrieves an error notification in the following format.

```
{
  "operation": "error",
  "exchange": null,
  "payload": {},
  "status": "error_code",
  "error": {
      "error_description":"error_description>",
      "error_specifics":"error_specifics"
   }
 }
```

where *error_code* is one of the following values: 5000, 5010, 5100.

For more information about error codes, see *Error Handling*.

Intent Notification

The intent notification is used to signal an intention to take an action. For example, an intent to provide an e-signature is generated when a user taps their authorized Nymi Band against an NFC reader.

A NES server must be specified in the `init` message in order for intent notifications to be received.

Intent notifications appear in the following format:

```
  {
  "operation": "intent",
  "exchange": null,
  "payload": {
  "device": "MAC address",
  "type": "see below",
  },
  "status": 0,
  "error": {}
  }
```

where *device* is the Nymi Band device ID.

Type is used to identify the manner in which the action was initiated.

type field	description
nfc	Nymi Band is in an authenticated state and tapped an NFC reader

Status Codes

A 2201 status code is reported when the NFC reader is unsuccessful at mapping the NFC ID to the enrolled Nymi Band.

A 2200 status code is reported when a NES communication error (for example, NES is offline) occurs.

Note: The 2201 and 2200 status codes do not contains a device ID in the payload.

assert_identity response

The assert_identity request returns *Username* and *Domain.* properties

assert_identity Results

The *UserStatus* property is an optional property. The UserStatus is stored in the Active Directory (AD).

If the UserStatus option is set in the NES console in the *Policies* > *Active Directory* page, the Active Directory status appears in the assert_identity response. If the option is not set, it does not return in the response.

The *UserStatus* option has the following possible values:

User Status	Definition
Active	User account is enabled.
NotExist	User account was deleted from AD.
Inactive	User account is disabled.
Locked	User account is locked. This status can appear with Active and Password Expired.
PasswordExpired	User account has an expired password. This status can appear with Active and Locked.

The last three properties can be combined into a coma separated list.

By default, NES disables support for user status checks in AD. Contact the NES administrator to enable AD user status checking, and optionally the checking interval in the .

A successful *assert_identity* operation produces a response with the following properties.

```
{
"operation": "assert_identity",
"exchange":"rAndOm_IdeNtifyiNG_StrING_5555",
"payload": {
```

```
"Username": "Jsmith",
"Domain": "Corp"
"UserStatus": "Active"

},
"status": "0",
"error: {}
}
```

Error Handling

The *update* function retrieves errors in the following scenarios.

- When a *request* operation fails, the response contains a non-zero "`status`" and *error* contains information about the failure. For example, when the *assert_identity* request was called with an incorrect *nes_url* value.
- When an *update* receives a notification response from NAPI as the result of a runtime error, the operation value is "`error`". For example, when the BLE adapter is removed from the USB port.

Notifications and response messages that result in an error appear in the following format:

```
{
  "operation": "operation_value",
  "exchange": "null" or "exchange_value",
  "payload": {}
    "status": status_code,
    "error": {
    "error_description": "general error description",
    "error_specifics": "specific error description"
  }
}
```

where:

- *operation_value* provides the operation value for the response or notification. For a response, the value is the same value that appeared with the request. For a notification, the value is `error`.
- *payload* does not contain any properties.
- *exchange* contains the user-defined exchange value, as it appeared in the request. If an exchange value was not specified in the request, the exchange value is `null`.
- *status_code* provides the status code that is associated with the error. See the *Status codes* table for more information
- *error_description* provides the description of the error that is associated with the status code.
- *error_specifics* provides additional information about the source of the error. For example, when a request specifies invalid parameters.

The following table summarizes the values that can appear in the *status_code* and *error_description*.

Status Code

Nymi provides you with status codes that assist you in solving SDK code-related issues and errors.

Table 6: Status codes

Status code	Error description
0	Operation completed successfully.
1000	Request made with invalid JSON.
1100	Request other than *init* sent before initialization.
1110	*init* request sent when already initialized.
1200	Cannot connect to NES. NES URL not specified in *init*.
2000	Request made with invalid parameters.
2102	Request made with device that does not exist. This is a permanent error, retries will fail.
2200	Problem occurred while communicating with NES.
2201	The requested query was not found on NES.
3000	Operation timed out. Retry the operation.
3010	Operation interrupted. For example, when the battery dies.
3100	Operation made during invalid band state.
4000	Connection to `Nymi Agent` lost. When you see this error, requests fail until *update* retrieves a reconnection notification.
4010	Request made while disconnected from `Nymi Agent`.
5000	Something went wrong with the Bluetooth Adapter.
5010	The Bluegiga BLED112 dongle is missing.
5100	`Nymi Bluetooth Endpoint` is missing or stopped.
6000	A temporary, recoverable error that indicates that the Nymi Band is currently not able to perform the operation, but the operation might succeed if the NEA tries the operation again.
7000	Error originating from the Nymi Band. Applies to device operations only.
8000	*init* payload requires the *token* and *nes_url* properties.
8001	NEA data is corrupt or not accessible.
8002	Missing organization name in the L1 certificate.
8100	*init* payload requires *otp* property.

Status code	Error description
9000	An error occurred. See *error_specifics* for more details.

Note: Status codes 1000 and 2000, should be considered the same as they indicate a messaging issue (for example, invalid JSON).

Troubleshooting

Nymi API writes information to log files that allow you to monitor and troubleshoot the NEA.

For additional assistance, visit the Support page on the Nymi website, or contact your Nymi Solution Consultant.

The following table summarizes the log files that are available for troubleshooting.

Table 7: Log file locations

Component	Log location	Files
Nymi API	By default, the current working directory.	*nymi_api.log*
`Nymi Agent`	*C:\Nymi\NymiAgent*	*nymi_agent.log*
`Nymi Bluetooth Endpoint`	*`NBE_INSTALL_DIR`\logs* where NBE_INSTALL_DIR is the path of the Nymi Bluetooth Endpoint.	*nymi_bluetooth_endpoint.log*

Enable debug mode

When testing NAPI and builds, set the *`NYMI_DEBUG`* environment variable to any value to enable debug logging, and the restart the `Nymi Agent` and `Nymi Bluetooth Endpoint` (NBE) services.

Information for C/C++ Developers

This section provides information that is specific to using C/C++ to develop an NEA

Preparing the C/C++ project to use NAPI

Before you can use NAPI, perform the following steps on the development machine, to load, initialize, and import the NAPI functions into the NEA project.

1. Create a new C/C++ project in Visual Studio.
2. Add *nymi_api.dll* to the project.
3. Define the following functions in the header file:

```
typedef int (WINAPI* REQUEST_FUNC_POINTER)(const char*);
typedef const char* (WINAPI* UPDATE_FUNC_POINTER)(int timeout_ms);
REQUEST_FUNC_POINTER request = NULL;
UPDATE_FUNC_POINTER update = NULL;
```

4. Create an *init* function in a C/C++ file with the following:

```
void init() {
   HINSTANCE hDll = LoadLibrary("Path to NAPI DLL folder");
   if (hDll) {
      request = (REQUEST_FUNC_POINTER) GetProcAddress(hDll, "request");
      update = (UPDATE_FUNC_POINTER) GetProcAddress(hDll, "update");
   }
}
```

5. Call the *init* function from your code.
6. Next verify the initialization was successful (request and update are not NULL).
7. Call the *request function* and *init* theNymi-enabled Application:

```
request ("{\"operation\": \"init\", \"payload\":{\"nea_name\": \"application_name\",
\"nes_url\": \"https://nes.server.com/NES\",\"token\": \"TokenBearerString\"}}");
```

   Note: If the request function successfully sends a message to NAPI, a value of 0 is returned. When a NAPI initialization has not occurred, and you send any request other than the *init* request, the request fails and returns a value of 1. Use the *update function* to retrieve details about the request. For more information about how to use the *update function*, see the *update Function* section in this guide.

8. Use the update function to retrieve details about the initialization status. A string is returned to you with and error or a **`ble_ready`** status.
   NAPI is now initialized and operations can be performed.

Information for C# Developers

This section provides information that is specific to using C# to develop an NEA.

Preparing the C# project to use NAPI

Before you can use NAPI, perform the following steps on the development machine, to load, initialize, and import the NAPI functions into the NEA project.

1. Open the NEA project in Visual Studio.
2. Add *nymi_api.dll* to your C# project and copy the *nymi_api.dll* file to the working directory.
3. Create a class to wrap the NAPI functions.
4. Define the name of the NAPI library (*nymi_api.dll*) in your class as follows.

```
private const string DllName = "nymi_api.dll";
```

5. Import the *request* and *update* functions from *nymi_api.dll* into your class as follows.

```
[DllImport(DllName, EntryPoint = "request")]
static extern Int32 request(string message);

[DllImport(DllName, EntryPoint = "update")]
static extern IntPtr update(int timeout_ms);
```

6. Initialize NAPI in your NEA project by calling the request function as follows.

```
request ('{"operation": "init", "payload":{"nea_name": "application_name", "nes_url": "https://
nes.server.com/NES", "token": "TokenBearerString"}}')
```

Note: If the *request* function successfully sends a message to NAPI, a value of 0 is returned. When a NAPI initialization has not occurred, and you send any request other than the *init* request, the request fails and returns a value of 1. Use the *update* function to retrieve details about the request. For more information about how to use the *update* function, see *The update Function*.

7. Use the *update* function to retrieve details about the initialization status.
8. Enter the following lines to import the *update* function from the *nymi_api.dll* and to declare the *update* function.

```
[DllImport(DllName, EntryPoint = "update")]
static extern IntPtr update(int timeout_ms);
```

The *update* function returns the responses to request operations, presence notifications, and error notifications.

9. Convert the pointer that is returned by the *update* function to a C# string:

```
var p = update(Timeout);
var s = Marshal.PtrToStringAnsi(p);
```

Appendix

Review this section for supplementary information about `Nymi API C Interface`.

Authentication requirements

An NEA and the Nymi Band establish trusted communication by using certificates. The first time that a user runs the NEA, the NEA retrieves a certificate from NES. The NEA certificate is stored in a keystore. Access to the keystore, by default, is enabled for all users

The NES Administrator can configure automatic or manual certificate retrieval. When the NES Administrator configures manual certificate retrieval, to initiate the retrieval process, you must specify a one-time password (OTP) in the *init* operation.

Acquire an Authentication Token

The first operation that the NEA must call is an *init* operation. If the *init* call results in a status code 8000, the (NEA must make an HTTP request to the NES REST API and acquire a token).

You can access NES by using one of the following endpoints to acquire the initial token:

- Basic Authentication
- Basic Authentication with cookies
- Negotiate

Basic Authentication (https://AS_url/api/BasicLoginWithToken)

This endpoint requires you to pass the user credentials in the authorization header.

A successful call:

- Returns one of the following outputs:
  - When Accept Header is set to application/xml or application/shtml+xml, the following xml output:

```
<LoginWithTokenResult xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://
schemas.datacontract.org/2004/07/Providers.Interfaces">
<Success>true</Success>
<Token>
…
</Token>
</LoginWithTokenResult>
```

  - When the Accept header is not defined, the following JSON string:

```
{"Success"="true", "Token"="<token>"}
```

- Passes the token in the WwwAuthenticate header.

Basic Authentication with Cookies (https://AS_url/api/BasicLoginWithCookies)

This endpoint requires you to pass the user credentials in the authorization header.

A successful call:

- Returns the following JSON string:

```
{"Success"="true", Cookies={"cookie1": "value1", "cookie2": "value2"}}
```

- Pushes the token as a *NymiAuth* cookie.

Nymi Inc. (Nymi) believes the information in this document is accurate as of its publication date. The information is subject to change without notice.

The information in this document is provided as-is and Nymi makes no representations or warranties of any kind. This document does not provide you with any legal rights to any intellectual property in any Nymi product. You may copy and use this document for your referential purposes.

This software or hardware is developed for general use in a variety of industries and Nymi assumes no liability as a result of their use or application.Nymi, Nymi Band, and other trademarks are the property of Nymi Inc. Other trademarks may be the property of their respective owners.

Published in Canada.
Nymi Inc.
Toronto, Ontario
www.nymi.com