Nymi SDK for C Developer's Guide

Nymi Connected Worker Platform
v1.0
2022-05-16

Contents

Nymi SDK Overview.................... 3

Development Tools.................... 4

Supported Platforms.................... 4

Supported NFC Readers.................... 4

NEA Certificates.................... 5

Sample Application.................... 5

Configure the Development Terminal.................... 7

SDK Package.................... 7

Importing the Root CA certificate.................... 7

Installing the Nymi Runtime.................... 9

Creating NEAs with NAPI.................... 11

Overview of NAPI.................... 11

Call Concurrency.................... 11

request() function.................... 11

update() function.................... 13

Response Messages and Notifications.................... 13

Error Handling.................... 14

Example: Workflow for Nymi Band Tap.................... 16

Preparing the C/C++ project to use NAPI.................... 18

Preparing the C# project to use NAPI.................... 19

Acquire an Authentication Token.................... 20

init operation.................... 21

Initialization error notifications.................... 24

Bluetooth notifications.................... 25

presence operation.................... 26

Presence Notifications.................... 27

Intent Notification.................... 29

assert_identity operation.................... 30

assert_identity response.................... 31

lookup.................... 32

device version.................... 34

subscribe operation.................... 35

Troubleshooting.................... 37

Enable debug mode.................... 37

Nymi SDK Overview

The Nymi SDK provides Developers with libraries, APIs, sample code and documentation to build a Nymi-enabled Application (NEA). The `Nymi API` (`NAPI`) architecture is part of the Nymi SDK.

The Nymi SDK package contains the following components:

- `Nymi Runtime` - Handles the primary functions of the Nymi Band communication and consists of the following components:
    - `Nymi Agent` - Provides BLE management, manages operations and message routing. Facilitates communication between NEAs and the Nymi Band, and maintains knowledge of the Nymi Band presence and authenticated states.

      You can install `Nymi Agent` on each workstation or install `Nymi Agent` in a central location, and then specify the location of the `Nymi Agent` in an `Nymi Bluetooth Endpoint` configuration file.
    - `Nymi Bluetooth Endpoint` - Provides an interface between the Bluegiga Dongle (BLE) and the Nymi Agent. You deploy `Nymi Bluetooth Endpoint` on individual workstations to provide local BLE communication with Nymi Bands through the Nymi-provided Bluegiga Adapter.
- `NAPI` - Library that acts as a universal converter plugin with a standard set of instructions to allow developers to create NEAs that access Nymi Band functionality and securely communicate with Nymi Bands.

  `NAPI` supports the use of Bluetooth and NFC to provide intent. For example, when a user taps their Nymi Band on an NFC device, that is connected to their user terminal, the `Nymi Bluetooth Endpoint` sends an intent event message to the `Nymi Agent`.

  `NAPI` 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 Nymi SDK with any language or environment that supports linking with C libraries.

Nymi SDK Package Contents

The Nymi SDK package contains the following artifacts:

- *nymi_api.dll* (`NAPI`)
- sample applications
- *BleDriver_xx.msi*
- `Nymi Runtime` installer

Nymi SDK Components and Communication

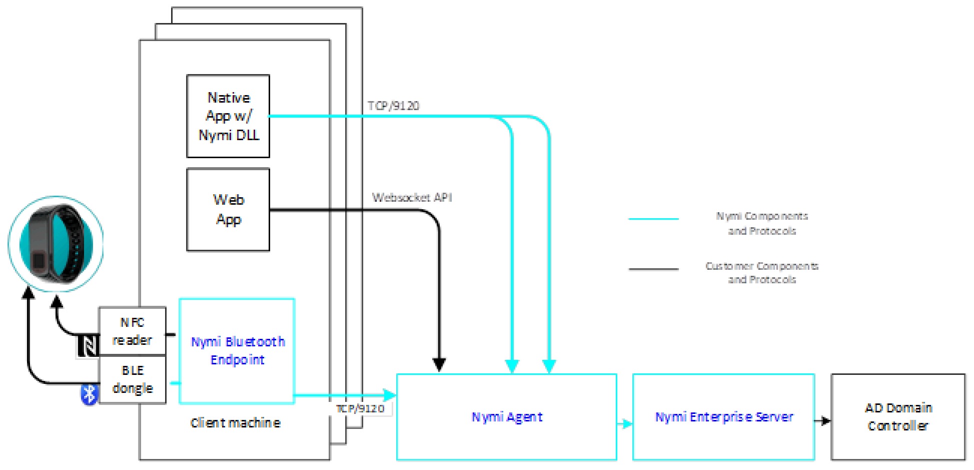


Figure 1: Nymi SDK Components and Communication

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).
- Any language that interfaces with a DLL, for example, Python

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

Supported Platforms

The `Nymi SDK` supports the following platforms.

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

Supported NFC Readers

When you connect a supported NFC reader to a user terminal where the Nymi Bluetooth Endpoint is installed, `Nymi Bluetooth Endpoint` automatically detects and monitors all attached NFC readers, and then forwards events from all NFC readers to the NEA through the `Nymi Agent`

A list of supported NFC Readers is found in the *Nymi Connected Worker Platform Administration Guide*.

NEA Certificates

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.

By default, the keystore is in the *%APPDATA%\Roaming\Nymi* directory.

Alternative locations include:

- *C:\Windows\ServiceProfiles\LocalService\AppData\Roaming\Nymi* for the Local Service account.
- *C:\Windows\system32\config\systemprofile\AppData\Roaming\Nymi* for the LOCAL SYSTEM (64-bit binary) account.
- *C:\Windows\SysWOW64\config\systemprofile\AppData\Roaming\Nymi* for the LOCAL SYSTEM (32-bit binary) account.

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* folder and contain applications that are developed in C# and C++ .

Sample Application for C++

The sample application for C++ is located in the *...\nymi-sdk\windows\samplesApps\cpp\sdkSample \sdkSample* folder.

Before you can use the sample application, modify the following content in the *sdkSample.cpp* file to reflect the configuration of your environment.

1. For

   ```
   const char* nes_url = "nes_url";
   ```

   replace `nes_url` with `https://nes_server/nes_service_name` where:

   - `nes_server` is the Fully Qualified Domain name of the NES host.
   - `nes_service_name` is the services mapping name of the NES web application. The default value is nes.

     For example, https://ev3-uat-srv1/ev3-uat-lab.local/nes

     Note: The service mapping name for NES was defined during deployment.
   - Close *regedit.exe*.
2. For

   ```
   const char* nes_directory_service_id = "NES_DS";
   ```

, replace NES_DS with the service mapping name that you provide in the previous step.

3. For

```
const char* username = "username_goes_here";
```

, replace `username_goes_here` with a username of an user that is valid in AD.

4. For

```
const char* password = "password_goes_here";
```

, replace `password_goes_here` with the password of a user that is valid in AD.

5. For

```
const char* nea_name = "NEA_name_goes_here";
```

, replace `NEA_name_goes_here` with a arbitrary name to provide the NEA.

Sample Application for C#

The sample application for C# is located in the *..\nymi-sdk\windows\csharp\sdkSample\SDK_Sample* folder. The application prompts you for the configuration parameters that are unique to your environment.

Configure the Development Terminal

On the development terminal, install the Nymi software and the required certificates.

SDK Package

The SDK package contains the following files

- *..\nymi-sdk\windows/i686* - Contains the NAPI dll file for i686 user terminals.
- *..\nymi-sdk\windows\sampleApps* - Contains sample Nymi-enabled Applications(NEAs).
- *..\nymi-sdk\windows\x86_64*- Contains the NAPI dll file for i686 user terminals.
- *..\nymi-sdk\windows\setup\BleDriver_x64.msi* - 64-bit Bluegiga driver installation file.
- *..\nymi-sdk\windows\setup\BleDriver_x86.msi* - 32-bit Bluegiga driver installation file.
- *..\nymi-sdk\windows\setup\NymiRuntime-5.9.1.8.msi* - `Nymi Runtime` MSI installation file.
- *..\nymi-sdk\windows\setup\Nymi Runtime installer.`version`.exe* - `Nymi Runtime` installation file.

Importing the Root CA certificate

Perform the following steps only if the Root CA issuing the NES TLS server certificate is not a Trusted Root CA (for example, if a self-signed TLS server certificate is used for NES). Install the Root CA on each user terminal to support the establishment of a connection with the NES host.

About this task

While logged into the user terminal as a local administrator, use the `certlm` application to import the root CA certificate into the Trusted Root Certification Authorities store. For example, on Windows 10, perform the following steps:

Procedure

1. In `Control Panel`, select **`Manage Computer Certificates`**.
2. In the `certlm` window, right-click **`Trusted Root Certification Authorities`**, and then select **`All Tasks > Import`**.

   The following figure shows the `certlm` window.

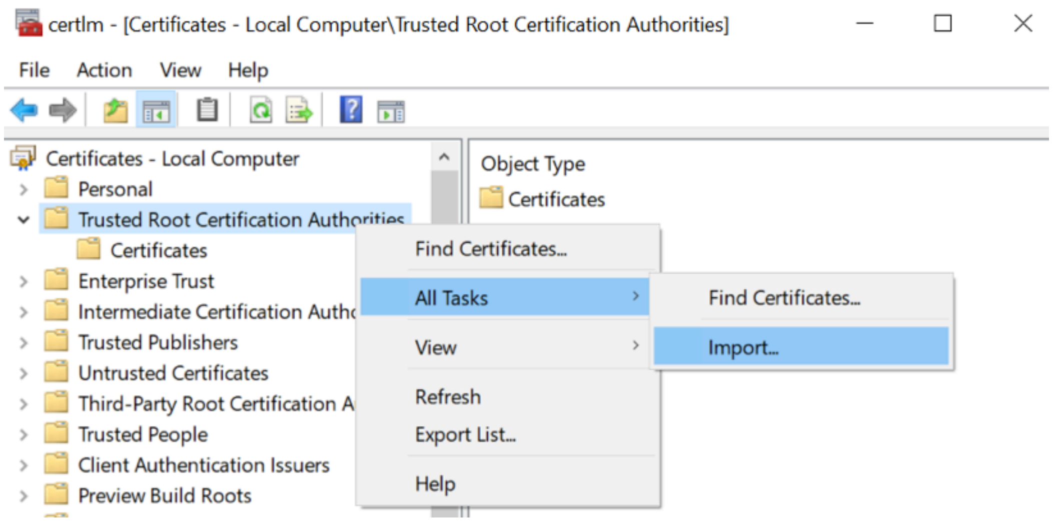
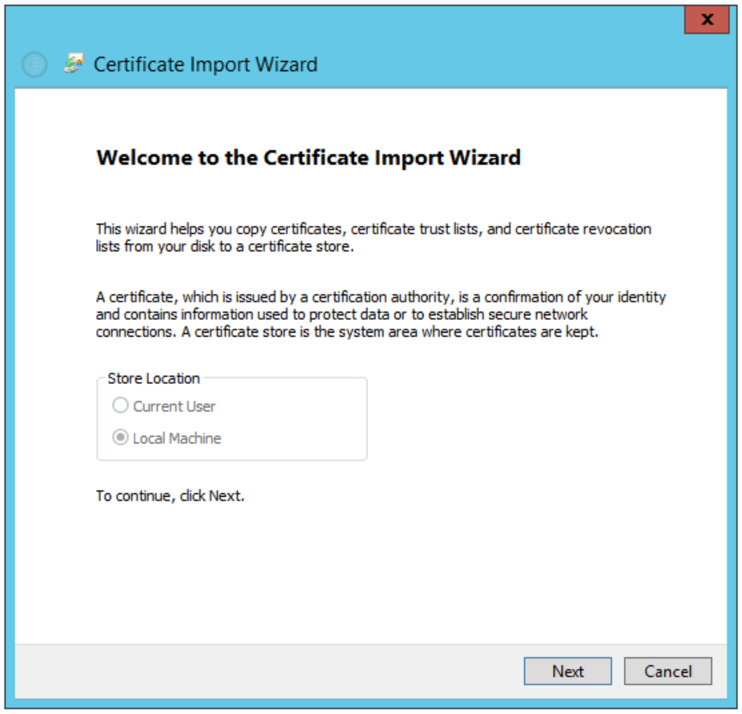

certlm - [Certificates - Local Computer\Trusted Root Certification Authorities]
File
Action
View
Help
Certificates - Local Computer
Personal
Trusted Root Certification Authorities
Certificates
Enterprise Trust
Intermediate Certification Autho
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification A
Trusted People
Client Authentication Issuers
Preview Build Roots
Object Type
Certificates
Find Certificates...
All Tasks
View
Refresh
Export List...
Help
Find Certificates...
Import...

Figure 2: certlm application on Windows 10

3. On the `Welcome to the Certificate Import Wizard` screen, click **`Next`**.

   The following figure shows the `Welcome to the Certificate Import Wizard` screen.



Certificate Import Wizard

Welcome to the Certificate Import Wizard

This wizard helps you copy certificates, certificate trust lists, and certificate revocation lists from your disk to a certificate store.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

Store Location

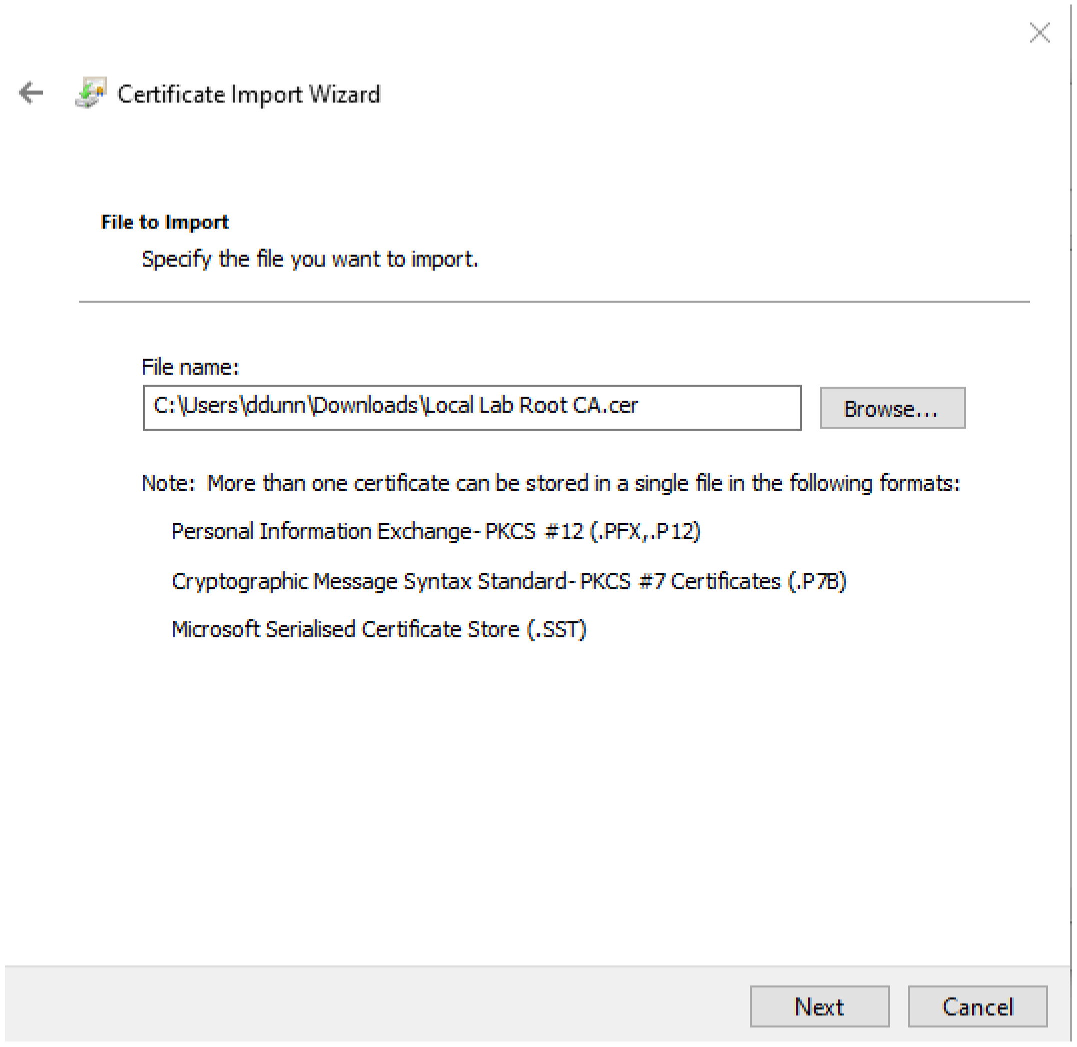
Current User [✓ Local Machine]

To continue, click Next.

Figure 3: Welcome to the Certificate Import Wizard screen

4. On the `File to Import` screen, click **`Browse`**, navigate to the folder that contains the root certificate file, select the file, and then click **`Open`**.
5. On the `File to Import` screen, click **`Next`**.

   The following figure shows the `File to Import` screen.



File to Import

Specify the file you want to import.

File name:

C:\Users\ddunn\Downloads\Local Lab Root CA.cer

Note: More than one certificate can be stored in a single file in the following formats:

Personal Information Exchange- PKCS #12 (.PFX,.P12)

Cryptographic Message Syntax Standard- PKCS #7 Certificates (.P7B)

Microsoft Serialised Certificate Store (.SST)

`Figure 4: File to Import screen`

6. On the `Certificate Store` screen, accept the default value **`Place all certificates in the following store`** with the value **`Trusted Root Certification Authorities`**, and then click **`Next`**.
7. On the `Completing the Certificate Import Wizard` screen, click **`Finish`**.

Installing the Nymi Runtime

Perform the following steps to install `Nymi Runtime`.

Procedure

1. Extract the `Nymi SDK` package to the development machine.
2. with c#, C, or C++ , 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 5.8.x.y.exe* file.

   Where *x.y* is the version number.

   Note: In a physical environment, when you install the `Nymi Runtime`, accept all the defaults. For a virtual environment, install the `Nymi Bluetooth Endpoint` component only on the development machine. In a virtual environment, install the `Nymi Runtime` on the machine designated as the centralized `Nymi Agent`, and only install the `Nymi Agent` component.

Creating NEAs with NAPI

Customer and partner developers can use the NAPI to develop Nymi-enabled Application (NEAs) in programming 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

NAPI makes use of the following components.

- *request()*—Function call that is used to send messages from the NEA to NAPI. NAPI performs the operation that is contained in the message. NEA supplies the request message in a memory buffer. Before the call returns, NAPI creates a copy of the message.
- *response()* - NAPI provides the results of the *request()* operation through a response.
- Notifications - System-generated messages that provide information about state changes in the environment. Notifications are not generated in response to a *request()*.
- *update()*— Function call that an NEA uses to retrieve *response()* messages and notifications from NAPI. After the function returns, NAPI expects the NEA to copy the response message out of the memory address provided by the *update()* call, before calling the *update()* function again.

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() function

Request messages are received by NAPI in JSON format as a null-terminated string argument to request().

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.

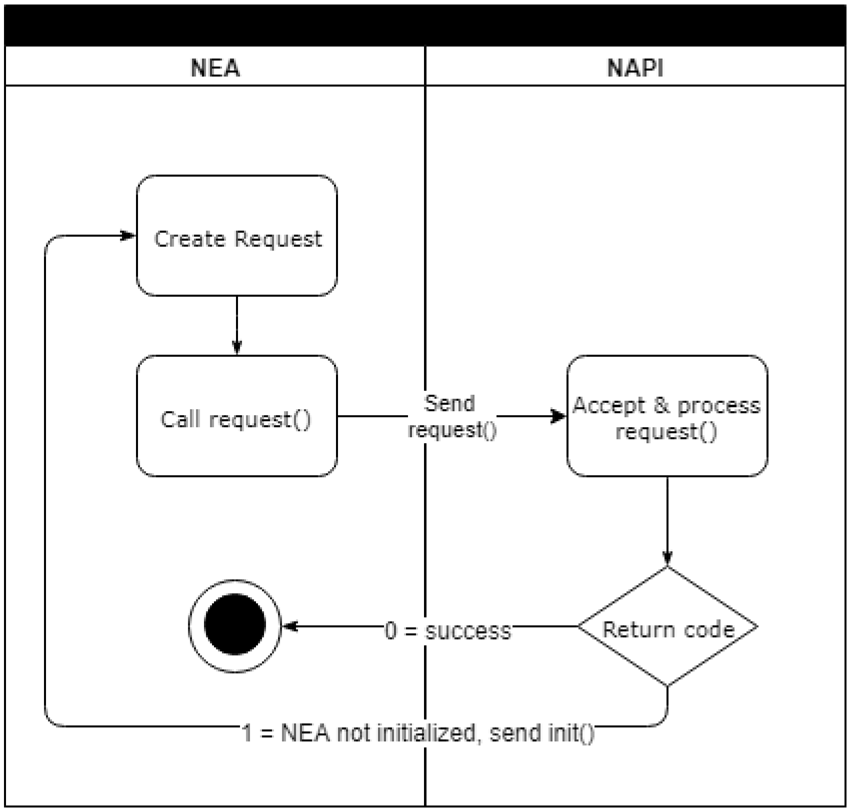


Figure 5: Request function workflow

1. Create the request in a memory buffer and pass the request to NAPI.
2. NAPI creates a copy of the request message.
3. NAPI initiates the requested operation.

   The *request()* call returns 0 when NAPI accepts the message and returns a 1 when the NEA has not been initialized. The NEA must run the *init* operation before NAPI 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 NAPI to perform. For example, *init*, *assert_identity*, and *lookup*.
- *Exchange_string*

  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.

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

Response Messages and Notifications

There are two types of responses.

- Responses are messages that are generated as a result of an request operation that was previously submitted to NAPI. Response messages include the same *operation*, *exchange*, and *status* values as the original request message.
- 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.

  Examples of notifications include:

  - 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.

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.

Error Handling

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

- When a *request* operation fails:
  - response contains a non-zero "*status*"
  - *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 1: Status codes

Status code	Error description
0	Applies to all operations to indicate success.
1000	Applies for all operations and indicates that the request operation was made with invalid JSON.
1100	Applies to any operation that is called before an *init* request. Indicates that request other than *init* request was sent before initialization.
1110	Appears for an *init* request and indicates that the*init* request was sent when the API has already been successfully initialized.
1200	Appears for an *init* request, when the NAPI cannot connect to NES, for example, when the NES URL was not specified in *init* request.
2000	Appears when a request operation was made with invalid parameters.
2102	Appears when a request and the Nymi Band ID value that is specified for the device property does not exist. This is a permanent error, retries will fail.

Status code	Error description
2200	Appears when a *lookup* and *assert_identity* request is made but NAPI cannot communicate with NES value that is specified in nes_url property.
2201	Appears when a *intent*, *lookup*, and *assert_identity* request is made but the requested query was not found on NES.
3000	Appears for any request operation and indicates that the operation timed out. Retry the operation.
3010	Appears for any request operation and indicates that the operation was interrupted. For example, when the battery dies.
3100	Appears for any request operation and indicates that the operation was made while the Nymi Band was in an invalid state.
4000	Notification to indicate that NAPI cannot connect to `Nymi Agent`. When you see this error, requests fail until *update* retrieves a reconnection notification.
4010	Appears for any request operation and indicates that the operation was made while NAPI is disconnected from `Nymi Agent`.
5000	Notification to indicate that NAPI cannot connect to the Bluetooth Adapter.
5010	Notification to indicate that the Bluetooth Adapter is missing.
5100	Notification to indicate that the `Nymi Bluetooth Endpoint` is missing or stopped.
6000	Appears for any request operation and indicates that a temporary, recoverable error has been generated by the Nymi Band. The Nymi Band cannot currently perform the operation, but the operation might succeed if the NEA tries the operation again.
7000	Appears for any operation and indicates that an error the Nymi Band generated an error. For example, when emory is full.
8000	Appears for an *init* request when the payload is missing the *token* and *nes_url* properties.
8001	Appears for an *init* request when the when certificates cannot be stored.
8002	Appears for an *init* request when the L1 certificate is missing the organization name.
8100	Appears for an *init* request when the payload is missing the *otp* property.
9000	Appears for an request when an error occurred. See the *error_specifics* property for more details.

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

Example: Workflow for Nymi Band Tap

The following image provides an overview of the calls and interactions between the NEA and `NAPI` when a Nymi Band user performs an NFC of BLE tap of the Nymi Band, while performing an authentication operation in the NEA.

Note: The workflow assumes that the NEA has already called *init()* and the response contained a status of 0.

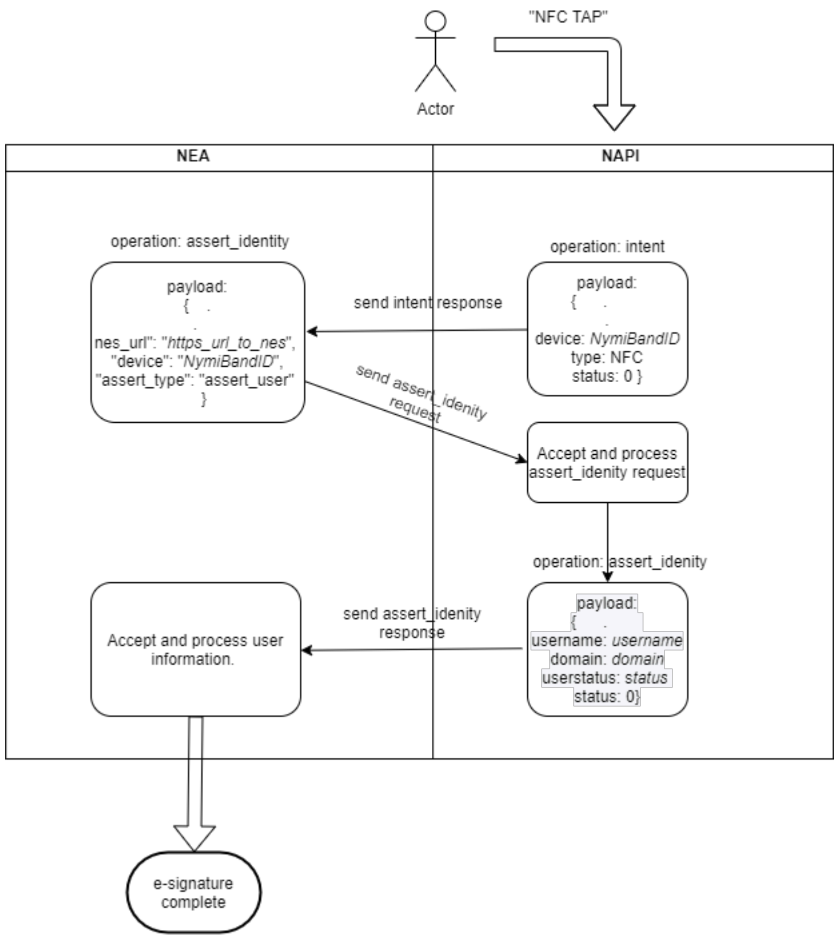

"NFC TAP"
Actor
NEA
NAPI
operation: assert_identity
payload: { . . nes_url": "https_url_to_nes", "device": "NymiBandID", "assert_type": "assert_user" }
send intent response
operation: intent
payload: { . . device: NymiBandID type: NFC status: 0 }
send assert_idenity request
Accept and process assert_idenity request
operation: assert_idenity
payload: { . username: username domain: domain userstatus: status status: 0}
send assert_idenity response
Accept and process user information.
e-signature complete

Figure 6: Workflow of operations during a tap

In this diagram, the following activities occur:

1. The Nymi Band user opens the NEA and performs a tap.
2. The update function in the NEA retrieves an intent notification. The payload of the notifcation contains the Nymi Band ID.
3. The NEA perform an *assert_identity* request and the device property in the payload specifies the Nymi Band ID that was in the intent notification.
4. The update function in the NEA retrieves an assert_identity notification.
   - If the the *assert_identity* request is successful (status is 0), the response contains the username and domain, and user status (if the Check User Status option is enabled in NES policy).
   - If the Nymi Band is not present or not authenticated the *assert_identity* request fails and the response contain a non-zero status value.
5. The NEA provides the appropriate result for the authentication task. For example, the e-signature completes.

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.

About this task

Procedure

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.

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.

About this task

Procedure

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);
```

Acquire an Authentication Token

The first operation that the NEA must call is an *init* operation with an authentication token, that you retrieve from NES.

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

- Basic Authentication
- Basic Authentication with cookies

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

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

A successful call performs the following two actions:

1. 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>"}
```

2. 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.

Negotiate Login with Token (https://AS_url/api/NegotiateTokenWithLogin

This endpoint does not require you to pass user credentials in the authorization header, but requires each user terminal and NES to access the same AD for centralized authentication. The method that you use is specific to the language that you use to develop the NEA.

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 two ways to call the *init* operation when initializing with certificate enrollment.

- nea_name
- nea_name + nes_url + token

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",
     "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.
- *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*.
- *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.
- url is required when you are using a centralized `Nymi Agent`, and *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.

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

Status code	Payload properties for subsequent *init* call
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*.
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.

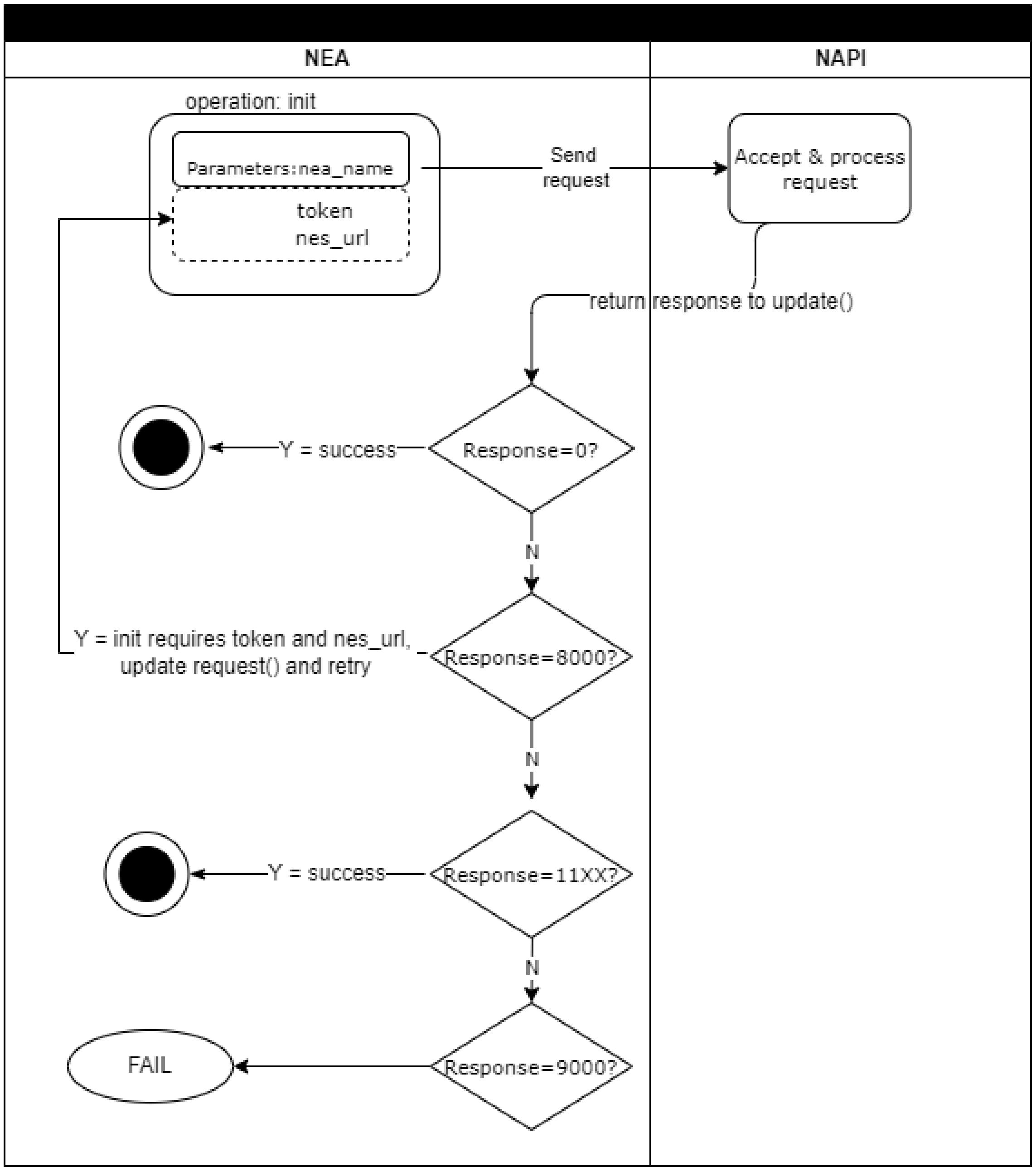


Figure 7: NAPI response calls to init

Initialization error notifications

After initialization of the API, and `init()` request results in a status of 0, 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 fails until the NEA retrieves a `reconnection` notification.

A reconnection notification with a status of zero. The follownig provides an example of a successful reconnection notification:

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

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*.

presence operation

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

When a presence 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 notifications*.

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": NymiBandID
      }
    }
```

Table 3: 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.

Presence Notifications

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

After *init()*, the *update* function retrieves a sequence of presence notifications, one for each Nymi Band that is present within range of the Bluetooth adapter. 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": "NymiBandID",
        "proximity" : "proximity value",
        "service_request_state" : "service request state",
        "state" : "state"
},
"error":{}
}
```

where:

- *proximity_value*: determined by the distance between the Nymi Band and the BLE adapter. The *proximity_value* will change when the Nymi Band moves closer or farther from the BLE adapter. The threshold (distance) for the *proximity_value* is determined in the *nbe.toml* file.

Note: To edit the *nbe.toml* file, refer to Editing the nbe.toml File.

- *state*: determined by the state of the Nymi Band; weak, absent, or unauthenticated.
- *service request state*: a flag that accompanies each presence notification and determines if there is a message in the Nymi Band that is ready to be downloaded. If the value of `service_request_state` is not zero, the Nymi Band has service level messages. If the value is '0', there are no messages

Note: 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 4: Proximity values for presence notifications

Proximity values	Definition	Example: Nymi Lock Control Behavior
4	The BLE adapter does not detect the Nymi Band.	For example, the user may be in another room. When the user enters the BLE adapter range, the *proximity_value* will go from 4 to 3. Nymi Lock Control does not perform any actions. When the user leaves the BLE adapter range, the *proximity_value* goes from 3 to 4. Nymi Lock Control does not perform any actions.
3	The BLE adapter detects the presence of the Nymi Band.	For example, the user is in the same room as their user terminal. When the user moves closer to the BLE adapter, the *proximity_value* will go from 3 to 2. Nymi Lock Control does not perform any actions. When the user moves further from the BLE adapter, the *proximity_value* goes from 2 to 3. Nymi Lock Control locks the user terminal if it is unlocked.
2	The BLE adapter is close to the Nymi Band.	For example, the user is near their user terminal. Nymi Lock Control keeps the user terminal unlocked while the user remains within this range (*proximity_value* is 2 or less). While Nymi Lock Control is enabled, the user may press the Enter key or the space bar on their keyboard to unlock their user terminal. When the user moves the Nymi Band closer to the BLE adapter, the *proximity_value* goes from 2 to 1. Nymi Lock Control will allow the user to access their user terminal without entering their credentials. When the user moves the Nymi Band further from the BLE adapter, the *proximity_value* goes from 1 to 2.

Proximity values	Definition	Example: Nymi Lock Control Behavior
1	The BLE adapter and the Nymi Band are in very close range.	For example, the user may be sitting at their user terminal. When the user moves the Nymi Band closer to the BLE adapter, the *proximity_value* goes from 1 to 0. This initiates a tap intent. When the user moves the Nymi Band away from the BLE adapter, the *proximity_value* goes from 0 to 1. This ends a tap intent.
0	The BLE adapter and the Nymi Band are adjacent (within 4 inches or 10 cm).	For example, the user places their Nymi Band on top of their BLE adapter. A tap intent is in progress and indicates a task.

Table 5: State values for presence notifications

State Value	Definition
Absent	The `Nymi Agent` cannot communicate with the Nymi Band. This state also applies when a user wears an unenrolled Nymi Band. 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 BLE Adapter for at least 30 seconds.
Unauthenticated	Nymi Band is enrolled and but not authenticated.
Weak	Nymi Band is in an authenticated state.

Intent Notification

An intent occurs when a user taps their authenticated Nymi Band next to an NFC reader or Bluetooth radio antenna, and is used to signal an intent 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.

To ensure that intent notifications are received, specify the NES server in the **`init`** message.

Intent notifications appear in the following format:

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

where *device* is the Nymi Band MAC address.

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

type field	description
ble	A user tapped an authenticated Nymi Band against a BLE device or is in close proximity to a BLE radio antenna, such as a BLE adapter.
nfc	A user tapped an authenticated Nymi Band against an NFC reader or is in close proximity to read range of the 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 contain a NymiBandID in the payload.

assert_identity operation

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* operation 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 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.
Active|Locked	User account is locked. This status can appear with Password Expired.
Active|PasswordExpired	User account has an expired password. This status can appear with Locked.

The last three properties can be combined into a comma 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 `NES Administrator Console`.

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: {}
}
```

lookup

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

- Device ID (MAC address) of the Nymi Band.

  Note: An intent notification includes the device ID or you can retrieve the device ID of a Nymi Band from NES by using the lookup operation.
- 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 6: AD user statuses

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

By default, NES is not configured to perform user status checks in AD. Contact the NES Administrator to enable AD user status checking, and optionally the checking interval in the `NES Administrator Console`.

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* the NES URL.
- *query* field is a JSON object that defines the values that are passed during the request to retrieve the response. Acceptable values include *NfcUID*, *Domain* and *Username*, and *NymiBandID*.

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

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: {}
}
```

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.

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

Device Version Response

Field	Definition
fw_version	U
hw_version	AD
exchange	U

subscribe operation

The `subscribe_endpoint` operation allows an NEA to change the Nymi Bluetooth Endpoint to which it is subscribed.

subscribe_endpoint request operations appear in the following format:

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

where:

- *operation* is *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. Required when the configuration uses a centralized `Nymi Agent`.

The *subscribe_endpoint* operation returns a status code only, no errors are returned.

```
{
“operation”: “subscribe_endpoint”,
"exchange":"exchange_value",
“payload”: {}
```

```
        “status”: 0,
        “error”: {}
        }
```

An NEA 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 Nymi Bluetooth Endpoint 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 Nymi Bluetooth Endpoint. If the Nymi Bluetooth Endpoint becomes ready at a later time (for example, that workstation is powered on), the NEA will receive a ble_ready message at that time.

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`	*C:\Nymi\Bluetooth_Endpoint \logs*	*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` services.

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