NOVA README

v1.0.2

This readme covers general instructions for using NOVA. You should also receive a companion document titled **NOVA Manual** which covers sending messages to NOVA through the HTTP API.

A changelog is included at the end of this readme.

For support, please direct emails to **support@nebland.com**

Build Instructions

See `readme-windows-build.md` or `readme-linux-build.md`.

Configuring NOVA

NOVA is executed from the command line and has two required parameters: a path to a json formatted schema file (which describes the NOVA config), and the path to a NOVA config file, like this:

```
nova <path/to/schema.json> <path/to/config.json>
```

Settings in the config file can be overridden from the command line by passing additional parameters after the configuration file parameter. The config override options have the following format:

```
--object.path.in.config=value
```

Here's an example of setting the `vtnUrl` and `venName` from the command line:

```
nova <path/to/schema.json> <path/to/config.json> --
oadr.vtnUrl=http://localhost:3000 --oadr.venName=newVenName
```

A sample NOVA configuration can be found in `nova-source-root/nova/config/config.json.sample`. Refer to the **Sample HTTP Plugin** section for instructions to configure the plugin.

Using the default configuration file, NOVA won't be configured to connect to a VTN. At a minimum, the *venName* and *vtnUrl* will need to be modified.

Here's an explanation of the NOVA configuration parameters:

- **startPollThread** (boolean): Start the poll thread on startup. If `true`
- **ignoreEventRandomization** (boolean): NOVA will ignore event randomization if `true`. If `false`, NOVA will randomly start and end events when randomization is present.
- **persistIds** (boolean): Reuse VEN ID and Registration ID received from the VTN on subsequent registrations, and persist to the config file
- **statusCallback**
  - **enable** (boolean): Enable/disable the status callback. This function serves as a heartbeat to the external system, and can be used to collect status data for status reports.
  - **callbackIntervalSeconds** (uint): callback interval
- **oadr** The following parameters are related to the oadr communication

    - **venName** (string): VEN name supplied by VTN maintainer.
    - **venId** (string): VEN ID supplied by VTN maintainer. If included, this VEN ID will always be present when registering with the VTN.
    - **registrationId** (string): Registration ID supplied by the VTN maintainer. If inlcuded, this registration ID will be present when registering with the VTN.
    - **vtnId** (string): Not used; ignore this parameter.
    - **vtnUrl** (string): HTTP path to the VTN. This string must begin with http or https.
    - **tls**
        - **enable** (boolean): Set to `true` to enable TLS security. If enabled, all of the following fields should have values and the VTN URL should begin with https.
        - **capath** (string): File system path to the CA file in PEM format.
        - **keypath** (string): File system path to the key file in PEM format.
        - **certpath** (string): File system path the certificate file in PEM format.
        - **verifyHostname** (boolean): If `true`, the TLS library will validate the expiration date of the server certificate, and that the common name in the certificate matches the host part of the VTN URL parameter.
        - **cipherList** (string): List of supported cipher suites. Under most circumstances this field should be left to the default value of: `AES128-SHA256:ECDHE-ECDSA-AES128-SHA256`.
          If curl is built with an SSL library other than OpenSSL, this list may need to be updated accordingly.
        - **tlsVersion** (string): Selects the TLS version. Under most circumstances, this field should be left at the default value of `TLSv1_2`. Other supported values are: TLSv1, SSLv2, SSLv3, TLSv1_0, TLSv1_1, and TLSv1_2
    - **marketContexts** (array of RegEx strings): Lists the MarketContexts the VEN matches against. Market contexts which don't match the list will be rejected.
    - **signals** (array of RegEx strings): Lists the signalName.signalType the VEN matches against. Signals not matched by the list will be rejected by the VEN. When searching for a match, the signal name and type from the event are concatenated with a period separator. A signal with name **SIMPLE** and type **x-Custom** becomes: **SIMPLE.x-Custom**. Regular expressions should match that format. The signals supported by OpenADR are included by default.

- **plugin** NOVA is extended through plugins which implement an interface that accepts callbacks from NOVA
    - **pluginPath** (string): Path to the plugin shared library (.so on Linux, .dll on Windows)
    - **configPath** (string): Path to the plugin config file. This path is passed to the plugin. The plugin is responsible for parsing the config file.

- **listener** The listener listens for API requests. Messages exchanged over the listener are formatted as json. WARNING: there is no security placed on the listener: care should be taken in securing access to this port.
    - **enabled** (boolean): If `true`, the listener will be started on the port and ip addres listed below.
    - **port** (integer): If running as a regular user, the port must be a value greater than 1024.
    - **ip** (string): Set to `127.0.0.1` to listen on localhost only, or to `0.0.0.0` to listen on all interfaces.

- **log** Log file settings
    - **stdout** (boolean): If `true`, messages will be logged to stdout.
    - **filename** (string): File system path to the log file.
    - **flushIntervalInSeconds** (integer): Time in seconds before log buffer is flushed to file.
    - **rotate** Settings to limit log size and rotate log files
        - **enable** (boolean): Enable or disable log rotate settings
        - **maxSizeInBytes** (uint): Number of bytes to limit log to
        - **fileCount** (uint): Number of log files to use in rotate

- **timeouts** Timeouts in seconds
    - **connect** (integer): Sets the curl CURLOPT_CONNECTTIMEOUT option
    - **request** (integer): Sets the curl CURLOPT_TIMEOUT option

- **retries**
  - **retryIntervalInSeconds** (integer): Time interval in seconds to wait before retrying a failed job. Callbacks which throw an exception will be retried at this interval.

Connecting to a VTN with TLS

OpenADR requires connections to use TLS 1.2 with client side certificates. Both the client (the VEN, NOVA in this case) and the server (VTN) must have a certificate.

Generating Test Certificates

Certificates are generated by a certificate authority. The OpenADR alliance has selected Kyrio as the certificate provider. Test certificates can be generated for free through the Kyrio website. Users can create an account and generate certificates here: https://portal.kyrio.com/.

When generating certificates, select the *VEN* option and the *PEM* format. The website allows the *common name* field to be filled in free form or as a MAC address: either form can be used.

After clicking the **Genreate Certificate** button, the user is presented with a zip file to download. The zip file contains a number of files in whatever format was selected (PEM, DER, or PKCS12). Here's an image of the contents from a file generated from the Kyrio website:

Name	Size	Type	Modified
TEST_OpenADR_RSA_RCA0002_Cert.pem	2.1 kB	X.50...	30 November 2047, 00:00
TEST_OpenADR_RSA_MCA0002_Cert.pem	2.0 kB	X.50...	30 November 2047, 00:00
TEST_RSA_VEN_181217192038_privkey.pem	1.7 kB	X.50...	30 November 2047, 00:00
TEST_RSA_VEN_181217192038_cert.pem	1.5 kB	X.50...	30 November 2047, 00:00
TEST_RSA_VEN_181217192038_cert_Fingerprint.txt	29 b...	plain...	30 November 2047, 00:00

Here's a description of each file:

- **TEST_OpenADR_RSA_RCA0002_Cert.pem**: CA certificate file
- **TEST_OpenADR_RSA_MCA0002_Cert.pem**: CA certificate file
- **TEST_RSA_VEN_1234_privkey.pem**: Certificate private key
- **TEST_RSA_VEN_1234_cert.pem**: Certificate file
- **TEST_RSA_VEN_1234_cert_Fingerprint.txt**: Certificate fingerprint (this fingerprint isn't correct and isn't used)

The first two files are certificate authority files and are the same in every certificate bundle generated from Kyrio. The next two files - the private key and certificate file - contain the RSA private/public key pair and some information that was signed by Kyrio. The certificate file can be verified using the two CA certificate files.

Before these files can be used by NOVA, a little prep work must be done.

Preparing the Certificates

When connecting over TLS, NOVA requires 3 files:

1. CA certificate file
2. Private key file
3. Certificate chain file

CA certificate files are provided in the cacerts directory. There's a file for test and a file for production.

The *Private key* file from the certificate bundle download from Kyrio can be used as is.

The *Certificate chain* file is created by concatenating the following files in the order given:

- TEST_RSA_VEN_1234_cert.pem
- TEST_OpenADR_RSA_MCA0002_Cert.pem
- TEST_OpenADR_RSA_RCA0002_Cert.pem

The NOVA config file contains paths to each of these files. The parameters are *capath*, *keypath*, and *certpath* under *oadr->tls*.

Sample HTTP Plugin

NOVA communicates to the external system through plugins. Source for the sample plugin can be found in the following path: `nova-source-root/pluginimpl/plugin/notifier/http/`.
Configuration for the plugin can be found in `nova-source-root/pluginimpl/plugin/notifier/http/config/config.json.sample`.

The config file consists of a series of http paths which will receive a json message when the corresponding OpenADR event occurs. See the readme file found in the sample plugin path directory for more information.

Creating a plugin

Details for creating a plugin are coming. For now, use the sample HTTP plugin as a starting point. Refer to `nova-source-root/nova/deps/plugin/pluginapi/notifier/INovaNotifierPlugin.h` for a description of the API.

A pointer to an *IVENManager* is passed to the plugin's initialize function.
This object shouldn't be needed under most circumstances as messages sent through the http API (see the HTTP API section below) are forwarded to the VENManager. This object is offerred as a convenience in the event that NOVA lacks functionality required for integration.

FUNCTIONS ON the IVENManager OJBECT SHOULD ONLY BE CALLED WHEN INITIATED FROM THE HTTP API. THEY SHOULD NOT BE CALLED FROM ANY OF THE CALLBACKS.

The function `OnRegisteredMessage` in the sample plugin demonstrates calling the forEach event function on IVENManager.

Plugin Implementation Error Handling Tips

NOVA executes as a series of jobs which follow two simple rules:

1. Only one job may execute at a time
2. If a job fails, retry it at some interval

Each job is a series of steps. If a failure is detected in any step, the entire job will be retried. Failures mainly occur at the boundaries: when communicating to external systems. There are two external systems involved: the VTN and the customer system. Jobs that fail when communicating to the customer system will be retried at whatever retry interval is set in the config file. Jobs that fail when communicating to the VTN use an exponential back-off algorithm to determine when the next retry will occur.

Retries in a micro-service are an essential component of proper operation. Even when everything is up and running as expected, unexpected failures can occur at the network layer. Rebooting systems and services can also cause issues. Failures which occur from these and other circumstances are mitigated with retries.

Retries are most useful for coping with short outages. When long outages occur, the best strategy may be to restart NOVA. It is always safe to restart.

The following sections do not cover the intention of the plugin callbacks. See the header file `INovaNotifierPlugin.h` for an up-to-date description of the callbacks. Instead, the following sections provide tips for handling error conditions.

Idempotent Functions

Any function that can be retried must be designed to be idempotent. A function is idempotent if it can be executed multiple times with the same information and the system is left in a correct state.

Let's take a customer order system as an example. If the same order is charged to a customer account multiple times, the customer should only be charged once.

When failures occur, NOVA may call callback functions with the same information. The callback function and any functions called on the external system must be idempotent.

Error Handling in the Plugin

NOVA automatically handles retries to the VTN and may also handle retries to the external system. This section discusses when and how this retry mechanism can be used.

The custom plugin communicates to the customer system through a library which implements some protocol such as http, message queues, database, etc. When a failure occurs, the client code detects the failure in 1 of 3 ways:

1. An error code returned from a function call
2. An exception thrown from a function call
3. A negative response in the message from the external system

Assuming the micro-service code and API code on the external system are correct, all 3 errors are an indication that the system is temporarily down and the message should be retried at some point in the future. The 3rd error type might also indicate an error in the micro-service or the external system so it's good practice to log the 3rd error type and send a notification.

Error handling in the plugin can be done in two ways:

1. Catch the errors and handle retries in the plugin. The sample plugin does this. This strategy works OK for event handling, but does not work for report handling.
2. Don't catch exceptions and let NOVA do the retries. If an error is detected, throw a `std::runtime_error("With a useful message")` and let NOVA retry the job.

NOVA detects failures through C++ exceptions. As long as the exception can be caught with `std::exception`, NOVA will catch the error and retry the job at whatever interval is specified in the config. When an exception occurs, NOVA calls the OnException callback with details of the job that failed and the exception message.

You are not required to choose one strategy over the other. Choose whichever strategy works best for the given callback.

Some jobs communicate to both the VTN and to the customer system. In these instances, failures in the customer system affect what data is sent to the VTN. The following discussions on Event and Report handling in the plugin offer options for handling failures specific to those callbacks.

These discussions assume some familiarity of the callbacks. Before digging into the details below, it will help to read through and start implementing the callbacks.

Events

The plugin callbacks `OnDistributeEventStart`, `OnDistributeEventComplete`, `OnEvent`, and `OnEventCancel` all occur while NOVA is collecting data that will be sent to the VTN. These callbacks allow the plugin to opt in or out of events and that information is relayed back to the VTN. If an error is thrown from one of these functions, NOVA won't have the information needed to respond to the VTN and will delay sending the message to the VTN.

Here are some options for handling errors in these functions:

1. **Catch errors and default the opt status**: The default opt status should probably be optOut to the let the VTN

know the VEN is not available for control. Defaulting to optIn is also an option. Under this strategy, the external system would presumably not have event information stored in the customer system so a strategy is needed to sync the event information when the external system comes back online. Options to sync event information include restarting NOVA and writing a custom HTTP API function to retrieve the information. If the opt status of an event needs to be changed, the external system will also need to make calls to the http API function `Created Event` and/or `Opt Event`. IF nova is restarted, the opt status will automatically be updated since NOVA will retrieve the list of events and redo the event callbacks.

2. **Throw errors**: This is the easier strategy to implement. NOVA will keep retrying event processing and when the job completes successfully, NOVA and the external system will be in sync so there's no need for a strategy to resync event information once the external system is available again.

The other `OnEvent*` callbacks occur between NOVA and the plugin only - the VTN does not receive any communication from these callbacks. Since the VTN isn't involved, the plugin can handle retries (as the sample plugin does). The plugin can also detect problems and throw errors. Throwing errors is the easier strategy.

Reports

The callbacks `OnPeriodicReportStart` and `OnPeriodicReportComplete` do not properly handle exceptions. If an error occurs while processing these callbacks, NOVA must be restarted to correct the error. Failures in this callback should be extremely rare. We are working on a fix.

The `OnGeneratePeriodicReport` callback does properly handle exceptions. The plugin should track what the last upload date was for each `reportRequestID` and use that date to "widen" the report window when an error occurs. The sample plugin tracks this information in the `QueryIntervals` object. The only option for handling retries with this callback is to throw errors: retries cannot be handled within the plugin.

Registration

The registration process is a single job but it hits many callbacks for events and reports. If a failure occurs anywhere in the process, the entire job is restarted.

Event Handling Corner Cases

Two features of events which often cause implementation issues are:

1. Zero length events
2. Event randomization

Compliance testing tests these features and also tests canceling a randomized event. If the NOVA `OnEventStart`, `OnEventEnd`, `OnEventCancel` callbacks are used, the following corner cases are handled automatically. If these called backs aren't used, the following corner cases must be handled by the external system.

Regardless of whether or not your plugin uses the NOVA callbacks, EVERYONE should read these corner cases carefully and implement accordingly. Most notably, canceled randomized events.

1. **Zero length events**: events with a duration of 0 stay active until modified with a real duration or canceled.
2. **Randomized events**: NOVA has two modes for randomized events: one where NOVA picks the randomized start time and one where the external system picks the randomized time. If NOVA picks the start time, the external system is notified of the start and end times correctly, and the times in the callbacks are the randomized times.
   If NOVA randomization is disabled, the external system is responsible for parsing the randomization parameter and starting control at a randomized time. NOVA will call the start event callback at the unrandomized start time, but the external system should not start the event when the message is received, but should start the event after a randomized period.
3. **Canceled randomized events**: if a randomized event is canceled while active, the event should not end immediately - it should wait the randomized minutes before ending. In this situation, NOVA will send a canceled event message, followed by an end event message when the event is complete. The end event message will arrive

after a randomized amount of time.

Which Event Types to Support

The OpenADR profiles define a number of different event types such as SIMPLE/level and ENERGY_PRICE/price. Though some event types might not make sense for your device, implementations must support all event types. Implementations aren't required to have special logic for each event type, but each event type must be supported. See page 31 of the OpenADR 2.0b profile specification for a list of event types.

All implementations must support simple/level events. Simple/level events have a payload of 0, 1, 2, or 3, meaning there's only 4 control options when working with simple/level events. According to the OpenADR profiles, these payload values correspond to the following:

- 0 **normal**: run normal, don't perform any control
- 1 **moderate**: perform a moderate reduction in energy use
- 2 **high**: perform a high reduction in energy use
- 3 **special**: program defined

In addition to implementing simple/level events, select other event types that make sense for your device type and implement specific logic to handle those event types. For all other event types, use default logic for the event.

Following these tips will get you through compliance testing but the implementation will likely need to be modified to meet utility requirements. Furthermore, each utility will likely have different requirements for control for a given event type. This means special control logic may need to be written for each utility.

Implementing for Multiple Utilities

Each running instance of NOVA connects to a single VTN. Therefore, to work with multiple utilities, each utility require it's own instance of NOVA.

The NOVA plugin should be designed to work with any utility though some options in the config file will change (such as vtn url) for each utility. This is true whether NOVA is running in the cloud or on device. As much as possible, control logic should be implemented in the external system with pluggable control logic.

When running multiple instances of NOVA on the same machine, each instance will need it's own log file and listening port (if the HTTP API is enabled).

NOVA can be executed from the same location in two ways:

1. Use different config files for each instance
2. Use the same config file with different command line parameters to override the parameters that differ.

NOVA HTTP API

NOVA exposes an HTTP endpoint. This endpoint is not secured so care should be taken to control access to this listener. This port should be protected inside your firewall and should not exposed outside your firewall.

The listener can be disabled by setting the `listner -> enabled` parameter to `false`.
Under normal circumstances, the listener can be disabled. Alternatively, the listener can be configured to listen on `localhost` behind a secure proxy.

The HTTP endpoint exposes a json API. Most of the functions were implemented to aid in compliance testing, but the API can also be used for tighter integration with the external system.

The NOVA HTTP API supports the following commands:

- **Query Registration**: Send a query registration message.

- **Create Party Registration**: Initiate the registration process between NOVA and the VTN.
- **Cancel Party Registration**: Initiate cancel registration phase between NOVA and the VTN.
- **Clear Registration**: Clear NOVA's tracking of its registration status - no message is sent to the VTN.
- **Clear Reports**: Clear any pending report requests - no message is sent to the vtn.
- **Start Ven**: Start the Ven communication loop in NOVA. Upon starting, if NOVA isn't in a registered state, it will continually attempt to register with the VTN.
- **Stop Ven**: Stop the communication loop.
- **Request Event**: Issue a requestEvents command to the VTN.
- **Clear Events**: Clear events stored in NOVA. No messages are issued to the VTN.
  This function is primarily for testing purposes and would not normally be used in production.
- **Set VenID**: Set a preconfigured VenID.
- **Create Opt Schedule**: Creates an opt schedule.
- **Cancel Opt Schedule**: Cancels a previously created opt schedule.
- **Opt Event**: OptIn or optOut of an event with the createOpt request.
- **Created Event**: OptIn or optOut of an event with the createdEvent request.
- **Register Reports**: Complete report registration.
- **VEN Status**: Retrieve information regarding the current status of NOVA.
- **Registered Namespace**: This function executes code in the custom plugin. The plugin is responsible for parsing the parameters in this message.

API Request/Reply Format

Each API request has the following format:

```
{
    "namespace": "ven.functionName",
    "parameters": {
        "param1" : "parm1 value",
        "param2" : 200,
        "param3" : true
    }
}
```

If NOVA successfully parses the request, the response will have the following:

```
{
    "status": "OK"
}
```

The OK status response only indicates NOVA understood and processed the request. It does not indicate the success or failure of the execution of the command.

If there was an error parsing the request, the response will be as follows:

```
{
    "status": "ERROR processing request: key 'eventId' not found"
}
```

If an exception is thrown processing the request, the output is as follows:

```
{
    "status": "OK",
    "exception": "Exception message"
}
```

If the status is OK and no exception exists, further information will be available in the response body depending on the function that was executed. Functions that send a request to the VTN will report the OpenADR EiResponse status as

follows:

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

A response code of 200 indicates the VTN received and processed the message successfully. A 4xx error indicates an invalid ID was sent to the VTN and the message could not be processed. A 5xx error or any other response indicates the server could not process the message and the function call can be retried at some later time.

Here's are rules to use when processing a response:

1. Check for `status` of OK. If the status is not OK, process the error and exit.
2. Check for the existence of `exception`. If there's an exception object, process the exception message and exit.
3. Process the response according to the function that was called

API Function Definitions

Each request and response for each functions is covered below.

Query Registration

Send a queryRegistration message.
Request

```
{
  "namespace": "ven.queryRegistration",
  "parameters": {}
}
```

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Create Party Registration

Initiate the registration process between NOVA and the VTN.

Request

```
{
    "namespace": "ven.createPartyRegistration",
    "parameters": {
        "includeIds" : false
    }
}
```

- **inlcudeIDs** (bool): If true, include `registrationId` and `venId` in the registration request.

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Cancel Party Registration

Initiate cancel registration phase between NOVA and the VTN.

Request

```
{
    "namespace": "ven.cancelPartyRegistration",
    "parameters": {}
}
```

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Clear Registration

Clear NOVA's tracking of its registration status - no message is sent to the VTN. NOVA will detect this as being in an unregistered state and start the registration process.

Request

```
{
    "namespace": "ven.clearRegistration",
    "parameters": {}
}
```

Response

```
{
    "status": "OK"
}
```

Clear reports

Clear all pending report requests - no message is sent to the VTN.

Request

```
{
    "namespace": "ven.clearReports",
    "parameters": {}
}
```

Response

```
{
    "status": "OK"
}
```

Start Ven

Start the VEN communication loop in NOVA. On start, if NOVA isn't in a registered state, it will continually attempt to register with the VTN. Once registration succeeds, NOVA enters the poll state.

Request

```
{
    "namespace": "ven.start",
    "parameters": {}
}
```

Response

```
{
    "status": "OK"
}
```

Stop Ven

Stop the communication loop.

Request

```
{
    "namespace": "ven.stop",
    "parameters": {}
}
```

Response

```
{
    "status": "OK"
}
```

Request Event

Issue a *requestEvent* command to the VTN

Request

```
{
    "namespace": "ven.requestEvent",
    "parameters": {}
}
```

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Clear Events

Clear events stored in NOVA. No messages are issued to the VTN.
This function is primarily for testing purposes and should not be used in production. This function does not send a message to the VTN.

Request

```
{
    "namespace": "ven.clearEvents",
    "parameters": {}
}
```

Response

```
{
    "status": "OK"
}
```

Set VenID

Set a preconfigured VenID. The VEN ID will be used in all future registration attempts. This function does not send a message to the VTN.

Request

```
{
    "namespace": "ven.setVenId",
    "parameters": {
        "venId" : "012345678901234567890123456789012345678901234567890123456789"
    }
}
```

- **venId** (string): OpenADR VEN ID. The VEN ID can be configured ahead of time in the NOVA config file, or the VTN will issue a VEN ID on registration.

Response

```
{
    "status": "OK"
}
```

Create Opt Schedule

Creates an opt schedule on. Opt schedules inform the VTN of periods of availability or unavailability for control.

Request

```
{
    "namespace": "ven.createOptSchedule",
    "parameters": {
        "optId" : "optIndentifaction",
        "optType" : "optOut",
        "optReason" : "economic",
        "marketContext" : "http://MarketContext1",
        "resourceId" : "resource1",
        "endDeviceAsset" : "Thermostat",
        "availability" : [
            { "dtstart": 1475696028, "duration" : 6},
            { "dtstart": 1475868828, "duration" : 8}
        ]
    }
```

```
}
```

- **optId** (string): Opt message identifier
- **optType**: (string): optIn, optOut
- **optReason**: (string): economic, emergency, mustRun, notParticipating, outageRunStatus overrideStatus, participating, x_schedule
- **marketContext** (string): Market context should match the list of market contexts the VEN is participating in. NOVA does not validate this field.
- **resourceId** (string): ID of the resource the opt schedule pertains to. This is an optional field. If blank, the schedule pertains to the entire VEN and all of its resources.
- **availability** (array): Array of dtstart/duration values.
  - **dtstart** (time_t): Unix epoch start time of the period
  - **duration** (integer): Duration of the period in hours

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Cancel Opt Schedule

Cancels an existing opt schedule

Request

```
{
    "namespace": "ven.cancelOptSchedule",
    "parameters": {
        "optId" : "optIndentifaction"
    }
}
```

- **optId** (string): ID of an existing opt schedule to cancel

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Opt Event

OptIn or optOut of an event with the createOpt message. This function is similar to the createdEvent which is also use to opt in or out of an event. This function offers contains a reason field and a resource ID field. The resource ID field is useful if the VEN is controlling multiple devices but only a subset of devices will be participating in the event.

Request

```
{
    "namespace": "ven.eventOpt",
```

```
    "parameters": {
        "optId" : "optId",
        "eventId" : "465159917efa511ede0f",
        "optType" : "optOut",
        "optReason" : "economic",
        "resourceId": "resourceId",
        "requestId": "requestId"
    }
}
```

- **optId** (string): Opt message identifier
- **eventId** (string): ID of the event to opt in/out of
- **optType**: (string): optIn, optOut
- **optReason**: (string): economic, emergency, mustRun, notParticipating, outageRunStatus overrideStatus, participating, x_schedule
- **resourceId** (string): ID of the resource the opt schedule pertains to. This is an optional field. If blank, the schedule pertains to the entire VEN and all of its resources.
- **requestId** (string): OpenADR request ID

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Created Event

OptIn or optOut of an event with the createdEvent request.

Request

```
{
    "namespace": "ven.createdEvent",
    "parameters": {
        "eventId" : "5cabbf1371f930c46af5",
        "optType" : "optOut",
        "scheduleEvent" : true
    }
}
```

- **eventId** (string): ID of the event to opt in/out of
- **optType**: (string): optIn, optOut
- **scheduleEvent** (bool): Set to false if NOVA should NOT schedule the event. If the event isn't scheduled, no callbacks will occur on the plugin. Cloud systems or other installations where NOVA is controlling multiple resources, this parameter will likely always be set to true. If NOVA is controlling a single resource and there's no need to receive event callbacks in the plugin for an event that has been opted out of, set this value to FALSE on optOut but true on optIn.

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

Register Reports

Perform report registration.

Request

```
{
    "namespace": "ven.registerReports",
    "parameters": {}
}
```

Response

```
{
    "response": {
        "code": 200,
        "message": "OK"
    },
    "status": "OK"
}
```

VEN Status

Retrieve information regarding the current status of NOVA. This function does not send a message to thee VTN.

Request

```
{
    "namespace": "ven.status",
    "parameters": {

    }
}
```

Response

```
{
    "endpointStatus": null,
    "events": {
        "31aef7bd0ad4cef12936": {
            "dtstart": "2019-02-25 17:11:00",
            "duration": "PT5M",
            "eventId": "31aef7bd0ad4cef12936",
            "optType": "optIn",
            "status": "completed"
        },
        "5cabbf1371f930c46af5": {
            "dtstart": "2019-02-25 17:29:00",
            "duration": "PT5M",
            "eventId": "5cabbf1371f930c46af5",
            "optType": "optIn",
            "status": "completed"
        }
    },
    "log": [
        "[ 2019-02-26 14:20:36 ] new event received: 5cabbf1371f930c46af5",
        "[ 2019-02-26 14:20:36 ] new event received: 31aef7bd0ad4cef12936",
    ],
    "polling": {
        "enabled": true
    },
    "registration": {
        "defaultRegistrationId": "",
        "defaultVenId": "",
        "isRegistered": true,
```

```
        "pollFreqquency": "PT10S",
        "registrationId": "b552b5225451f4a45191",
        "venId": "2423f6ed2d9599a8f6b5",
        "vtnId": "NEBLAND_VTN"
    },
    "status": "OK",
    "version": "1.2.0 SHA: HASH"
}
```

Registered Namespace

This is a custom function which is forwarded to the custom plugin. The namespace value that is forwarded to the plugin is set in the plugin initialize function. The sample plugin sets name to `my.custom.namespace`, doesn't take any parameters, and returns information regarding the events that have been received.

Request

```
{
    "namespace": "my.custom.namespace",
    "parameters": {
    }
}
```

Response

```
{
    "events": [
        {
            "dtstart": "2019-02-25 17:11:00",
            "eventID": "31aef7bd0ad4cef12936",
            "status": "completed"
        },
        {
            "dtstart": "2019-02-25 17:29:00",
            "eventID": "5cabbf1371f930c46af5",
            "status": "completed"
        },
        {
            "dtstart": "2019-02-25 16:53:00",
            "eventID": "88720fdaf57f768d08d9",
            "status": "completed"
        },
        {
            "dtstart": "2019-02-25 17:16:00",
            "eventID": "b9767e11626a20728b7d",
            "status": "completed"
        },
        {
            "dtstart": "2019-02-25 17:21:00",
            "eventID": "d6a993aee2aa3ea71d0d",
            "status": "completed"
        }
    ],
    "note": "This is an example of processing a message sent using a registered
namespace"
}
```

Connecting to the NOVA API with Boomerang

Boomerang is a chrome plugin that can send messages to HTTP endpoints. A Boomerang config file is contained in `nova-source-root/test/maual/boomerangNOVA API_Boomerang.json`.
This file can be loaded into the Chrome **Boomerang** plugin (https://chrome.google.com/webstore/detail/boomerang-soap-rest-clien/eipdnjedkpcnlmmdfdkgfpljanehloah?hl=en). The Boomerang config file contains sample messages for all of the NOVA API messages mentioned in the previous section.

By default, each request in the Boomerang file is pointed at `http://localhost:8080`.

Changelog

1.0.3-3

- Parse +/- preceding duration string

1.0.3-2

- Fixed interval start time calculation in sample plugin

1.0.3-1

- Fixed a few compliance issues

1.0.3

- Windows build updates
- Compliance testing bug fixes
  - Two event tests fail in 1.0.2 - users must upgrade to 1.0.3
- Convert http keys in headers to lowercase

1.0.2

- Retry failures from plugin
- Upgrade SPD logger to v1.3.1
- Add log flushing. Expose config parameter to flush the log an a specified interval
- Added CA certificates for test and production
- Added OpenADR response code and message to the HTTP API functions which communicate to the VTN
- Schedule/Unschedule event callbacks on opt in/out
- Override config file options from the command line
- Implement max log size and log rotation. Expose parameters in the config
- Implement last report date tracking in the sample plugin. If the OnGeneratePeriodicReport function throws an error, the function will be retried at the retry interval, and will use the last report date as a starting point for collecting data.
- Implement curl rewind. This fixes occasional errors that occur when the remote has closed the socket but curl has not yet detected the socket is closed.

1.0.1

- Windows support

1.0.0

- Populate optional UID field in Interval object
- Expose ReadingType in EiSpecifierPayload
- Handle report generation when no intervals are added to report
- Wait for thread pool threads to exit before shutting down
- Use async IO to execute on a single thread
- Updated Event callbacks (see nova/deps/plugin/pluginapi/notifier/INovaNotifierPlugin.h)
  - EventFailedValidation callback: notifies plugin when an event is rejected
  - ValidateEvent callback: let's plugin accept or reject an event
- Compliance test aganist the 1.1.2 Quality Logic test harness
- Report generation API improvements
- Wrap OpenADR objects passed to plugin in interfaces

- Add retries to sample plugin
- Pass logger to plugin
- Pass exceptions to plugin
- Pass sent and received XML messages to plugin
- Single threaded with ASIO

0.9.7-4

- Add signal matcher to config

0.9.7-3

- Add report generation helper to sample plugin

0.9.7-3

- Add http code and message to receive message callback
- Expose connect and request timeouts

0.9.7-2

- Reregister on 452/463 errors
- Added `persistIds` flag: if set, reuses the registration ID and VEN ID supplied by the VTN during registration, and persists the IDs in the config file
- Add OnRegister callback, called when the VEN successfully registers
- Add `dtsart` to startEvent callback

0.9.7-1

- Allow 0 duration report intervals to be scheduled
- Fix memory management issues in EventManager and ReportManager

0.9.7

- Allow regular expressions for market context matching
- Add complete event callbacks for active events that are canceled

0.9.6

- Add content type header option to HttCurl constructor
- Schedule event callbacks to allow `forEach` to be called from more callbacks
- Change event callbacks to use full event duration, not remainingDuration
- Use TLS strings (instead of integers) in NOVA config

0.9.5

- Opt out per resource in callbacks
- Enable or disable status callback and set the callback interval from the config
- Status Functon calls go to the plugin `OnStatus` callback
- Enable or disable event randomization from the config. If disabled, the external system is responsible for adhering to the randomization set in the event.
- Event visitor function added to the IVenManager API. IVenManager exposed to plugins
- Messages received on HTTP API that match a plugin supplied namespace are forwarded to the plugin function `OnRegisteredMessage`.

0.9.4

- Initial Release