ASNeG OpcUaWebServer Documentation

Kai Hübl, Aleksey Timin

Nov 27, 2019

Contents

ASNeG OPC UA Web Server is an open source OPC UA web server. It provides a simple way to connect modern Web applications with the OPC UA technology. Any process data can be displayed in realtime in Web applications using the ASNeG OPC UA Web Server.

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Overview

ASNeG OPC UA Web Server provides a simple way to connect modern Web applications with the OPC UA technology. Any process data can be displayed in realtime in Web applications using ASNeG OPC UA Web Server.

Structure of the OPC UA Web Server

The OpcUaWebServer contains the following components:

- HTTP Server
- Web Socket Server
- Web Gateway
- OPC UA Client

HTTP Server provides a simple interface for web pages. The provision of simple static web pages in HTML format does not require a separate web server for simple web applications with OPC UA access. The use of the HTTP Server component is optional.

WebSocket Server provides a JSON API via bidirectional WebSocket protocol for access to OPC UA variables. The WebSocket Server component is decoupled from the OPC UA server. Symbolic names are used to access OPC UA variables. A configuration in the WebSocket Server maps these variables to OPC UA variables and the assigned OPC UA server. The functionality of the interface is very simple but limited to a few functions. The disadvantage of the interface is the high configuration effort. The use of the WebSocket Server component is optional.

WebSocket Gateway provides a JSON API via bidirectional WebSocket protocol for access to OPC UA server services. The WebSocket Gateway component is not decoupled from the OPC UA server. All web application JSON packets are transferred from the WebGateway to binary OPC UA packets and sent directly to the assigned OPC UA server. A separate configuration for the WebGateway is not necessary. As an advantage, all OPC UA service functions

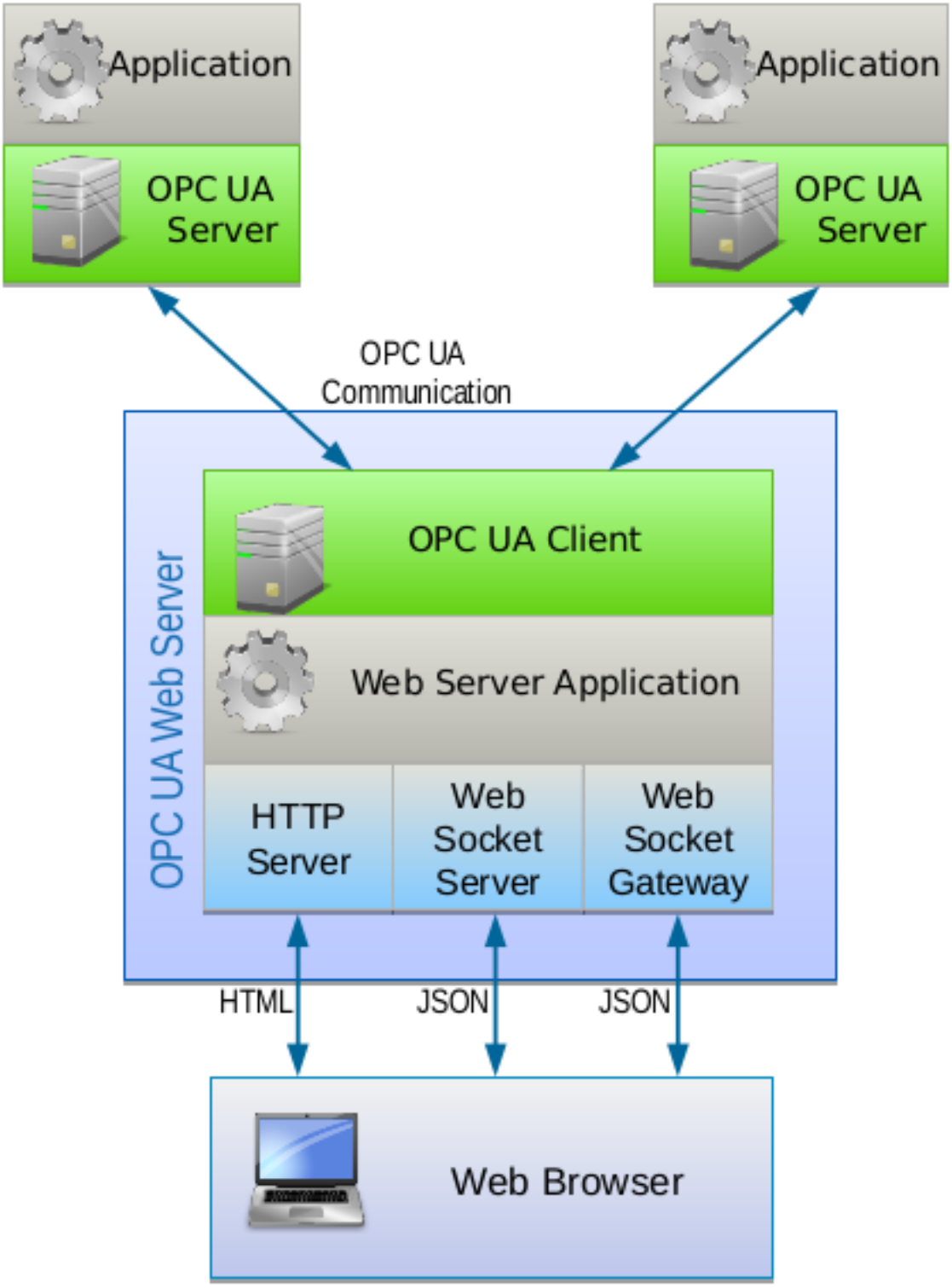

Fig. 1: **OpcUaWebServer components**

of an OPC UA server can be used directly from the web application. However, the JSON interface functions are a little bit complex than with the WebSocket Server model. The use of the Web Gateway component is optional.

OPC UA Client is used for communication with one or more OPC UA servers.

In addition, the Web Server provides **Web Panel** for visualization of the process data. **Web Panel** uses a JavaScript library of visual components which can be bound with OPC UA variables. You can use it as a very simple SCADA system configured by XML files. You can see how it work with our Demo Application.

WebSocket Server JSON API Reference

WebSocket Server has a simple JSON API with the limited functionality contains the following requests:

Name	Description
Value List	Returns all names of OPC UA variables processed by the server.
Value Info	Returns information about OPC UA variables (such as type, name etc.)
Read Value	Reads the value, status and timestamp of the given variable
Historical Read	Reads historical values of the given variables. **Not implemented.**
Write Value	Writes the value, status and timestamp of the given variable
Monitoring	Subscrubes to the given variable to receive its new values

For more information see *WebSocket Server JSON API*

WebSocket Gateway JSON API Reference

WebSocket Gateway supports all the OPC UA services that ASNeG OPC UA Stack covers. See its coverage tables for more information.

References

- ASNeG OPC UA Stack
- Demo Application
- *WebSocket Server JSON API*
- *WebSocket Gateway*

1.1.2 Installation

OpcUaWebServer provides different ways of installation.

Source Code

To compile and install the OpcUaWebServer from the source code, you should meet the following requirements:

- ASNeG OPC UA Stack >= 4.0.0
- ASNeG Demo Server >= 4.0.0 - Optional
- CMake
- C++ compiler with C++11 support

The ASNeG Demo Server is only used for testing the OpcUaWebServer. For this reason the installation of the ASNeG Demo Server is optional.

To install the requirements, follow the instructions in the following documents.

- Installation OpcUaStack.
- Installation ASNeG Demo Server.

Linux (Ubuntu or Debian)

To compile the source code and install OpcUaWebServer locally, you should run in the root directory of the sources:

```
$ sh build.sh -t local -s ASNEG_INSTALL_DIR
```

ASNEG_INSTALL_DIR is the path where ASNeG OPC UA Stack is installed.

By default, the server is installed into *~/.ASNeG* directory. You can change it with option *-i* (run *sh build.sh* for more details).

Now the OpcUaWebServer can be started.

```
$ export PATH=$PATH:~/.ASNeG/usr/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/.ASNeG/usr/lib/
$ OpcUaServer4 ~/.ASNeG/etc/OpcUaStack/OpcUaWebServer/OpcUaServer.xml
```

The OpcUaWebServer provides the following communication interfaces:

- HTTP interface on port 8080
- Web Socket Server on port 8081
- Web Socket Gateway on port 8082

Windows

To compile the project, you should install MSBuild Tools, then run in the environment which is suitable for your target platform (e.g., Native x86) the following command:

```
$ build.bat -t local
```

If you would like to build x64 version, you should set the suitable CMake generator for your environment with option *-vs*. For example:

```
$ build.bat -t local -vs "Visual Studio 15 2017 Win64"
```

By default, the server is installed into *C:\ASNeG* directory. You also can change it with option *-i*:

```
$ build.bat -t local -i C:\another\path\to\install
```

Now the OpcUaWebServer can be started.

```
$ set PATH=%PATH%;C:\ASNeG\bin
$ set PATH=%PATH%;C:\ASNeG\lib
$ OpcUaServer4 CONSOLE C:\ASNeG\etc\OpcUaStack\OpcUaWebServer\OpcUaServer.xml
```

This example is for the default case where the stack and the server are installed in *C:\ASNeG* directory.

DEP Packages

You can install OpcUaWebServer by using a DEP package, which you can download **'here <https://github.com/ASNeG/OpcUaWebServer/releases/'_** or build yourself by the following command:

```
$ sh build.sh -t deb -s ASNEG_INSTALL_DIR
```

Then OpcUaWebServer is available for installation by the command:

```
$ dpkg -i OpcUaWebServer-x.y.z-x86_64.deb
```

OpcUaWebServer will be installed as a service, and you can check its status typing:

```
$ service OpcUaWebServer status
```

RPM Packages

Users of RPM distributives can install OpcUaWebServer by using an RPM package, which can be downloaded **'here <https://github.com/ASNeG/OpcUaWebServer/releases/'_** or built by the following command:

```
$ sh build.sh -t rpm -s ASNEG_INSTALL_DIR
```

Then OpcUaWebServer is available for installing by the command:

```
$ rpm -i OpcUaWebServer-x.y.z-x86_64.rpm
```

MSI Packages

On Windows, OpcUaWebServer can be installed with MSI packages. The MSI packages belong to two kinds: a *usual* package without ASNeG OPC UA Stack and a *standalone* package including ASNeG OPC UA Stack and the all needed dependencies. The *usual* package is more suitable for a machine with several OPC UA applications and you would like to update the stack and the dependencies for all of them. The *standalone* package is easy for distribution, so you don't need to install anything except OpcUaWebServer.

You can download the packages **'here <https://github.com/ASNeG/OpcUaWebServer/releases/'_** or build them.

To build the *usual* package, use the command:

```
$ build.bat -t msi
```

For building the *standalone* one:

```
$ build.bat -t msi -S
```

Docker

You can use Docker image instead of installing the web server on you machine:

```
$ docker build -t OpcUaWebServer:latest .
$ docker run -d -p 8890:8890 -p 8080:8080 -p 8081:8081 -p 8082:8082␣
↪OpcUaWebServer:latest
```

If you want to run the server with the demo server as a data source, use *docker-compose*:

```
$ docker-compose run -d -p 8080:8080 -p 8081:8081 -p 8082:8082 webserver
```

Then open the link http://127.0.0.1:8080 with your web browser.

References

- ASNeG OPC UA Stack
- ASNeG Demo Server

1.1.3 Hello World

1.2 Configuration

ASNeG OPC UA Web Server has a flexible configuration in XML format. In this document, you'll learn how to configure HTTP and WebSocket servers and describe data sources.

The main configuration file is **OpcUaWebServerModel.xml**. You can find it in the sources:

```
src/OpcUaWebServer/Config/OpcUaWebServerModel.xml
```

or in the directory where the server was installed (**@CONF_DIR**)

```
path/to/directory/etc/OpcUaStack/OpcUaWebServer/OpcUaWebServerModel.xml
```

You can change the configuration in the sources if you want to distribute OpcUaWebServer with your configuration.

1.2.1 HTTP Server

The **HTTP Server** provides access to static web pages via HTTP protocol. Its configuration starts with tag *HttpServer* in **OpcUaWebServerModel.xml**.

Default Configuration

```
<HttpServer>
    <Address>0.0.0.0</Address>
    <Port>8080</Port>
    <WebDirectory>@CONF_DIR@/../../../var/www/OpcUaWebServer</WebDirectory>
    <RequestTimeout>5000</RequestTimeout>

    <IPLogger>
    <LogFileName>@CONF_DIR@/../../../var/log/OpcUaStack/OpcUaWebServer/access.log</
↪LogFileName>
    <MaxLogFileNumber>5</MaxLogFileNumber>
    <MaxLogFileSize>100000</MaxLogFileSize>
    <MaxIPAge>3600000</MaxIPAge>
  </IPLogger>

</HttpServer>
```

Configuration Settings

XML tag		Description
IP Address		IP address bound by the HTTP server
Port		Port bound by the HTTP server
WebDirectory		Root directory. It should have *index.html* file.
RequestTimeout		Time after the TCP connection establishment in milliseconds, that the server waits for the request from the client before closing the connection.
IPLogger		IP Logger registers all IP addresses of the clients that have connected with the server.
	LogFileName	Full file name of the log
	MaxLogFileSize	Max. count of the log files
	MaxLogFileSize	Max. size of the log files in bytes
	MaxIPAge	If the client connects to the server several times during this period in milliseconds, only one record is written in the log.

1.2.2 WebSocket Server

The **WebSocket Server** provides a JSON API via bidirectional WebSocket protocol for access to OPC UA variables. Its configuration starts with XML tag *WebSocketServer* in **OpcUaWebServerModel.xml**.

Default Configuration

```
<WebSocketServer>
    <Address>0.0.0.0</Address>
    <Port>8081</Port>
    <RequestTimeout>5000</RequestTimeout>
    <IdleTimeout>3600000</IdleTimeout>
</WebSocketServer>
```

Configuration Settings

XML tag	Description
IP Address	IP address bound by the WebSocket server
Port	Port bound by the WebSocket server
RequestTimeout	Time after the TCP connection establishment in milliseconds, that the server waits for the request from the client before closing the connection.
IdleTimeout	Time after the last message in milliseconds that the server waits before closing the connection.

OPC UA Client

In order to have access to OPC UA servers we have to include its communication settings and nodes in **OpcUaClient** part into **OpcUaWebServerModel.xml** file:

```
<OpcUaClient>
  <ClientConfigFile>@CONF_DIR@/OpcUaClient0.xml</ClientConfigFile>
  <ClientConfigFile>@CONF_DIR@/OpcUaClient1.xml</ClientConfigFile>
</OpcUaClient>
```

Example of OpcUaClient Configuration File

```
<?xml version="1.0" encoding="utf-8"?>
<OpcUaClient Name="ASNeG-Demo_0" xmlns="http://ASNeG/OpcUaClient.xsd">
  <Endpoint>
    <ServerUri>opc.tcp://127.0.0.1:8889</ServerUri>
    <SecurityMode>SignAndEncrypt</SecurityMode>
    <SecurityPolicyUri>http://opcfoundation.org/UA/SecurityPolicy#Basic128Rsa15</
↪SecurityPolicyUri>
    <UserAuth>
      <Type>UserName</Type>
      <UserName>user1</UserName>
      <Password>password1</Password>
      <SecurityPolicyUri>http://opcfoundation.org/UA/SecurityPolicy#Basic128Rsa15
↪<SecurityPolicyUri>
    </UserAuth>
  </Endpoint>
  <NamespaceUri>
    <Uri>http://ASNeG-Demo.de/Test-Server-Lib/</Uri>
  </NamespaceUri>
  <NodeList>
    <Node ValueName="TimerInterval" NodeId="ns=1;i=3" NodeType="UInt32">
      <MetaData>
        <DisplayName>TimerInterval</DisplayName>
      </MetaData>
    </Node>
    <Node ValueName="Boolean" NodeId="ns=1;i=220" NodeType="Boolean">
      <MetaData>
        <DisplayName>Switch</DisplayName>
        <Limits>
          <Min>0</Min>
          <Max>1</Max>
        </Limits>
      </MetaData>
    </Node>
  </NodeList>
</OpcUaClient>
```

Note: You need client configuration files only for **WebSocket Server**. **WebSocket Gateway** receives the information about communication and security through JSON API.

OpcUaClient Configuration Settings

XML tag			Description
Endpoint			Endpoint of the OPC UA Server, which the client connect to
	ServerUri		URI of the OPC UA Server
	SecurityMode		Security Mode can be "None", "Sign" and "SignAndCrypt". Default value is "None".
	SecurityPolicyUri		Security Policy URI used to encrypt OPC UA messages. See https://opcfoundation.org/UA/SecurityPolicy/
	UserAuth		Authentication settings, which the client use to connect with the OPC UA server
		Type	Type of the authentication can be "Anonymous" or "UserName"
		Username	Name of the authenticated user. Only for "Username" type.
		Password	Password of the authenticated user. Only for "Username" type.
		Security-Policy	Security Policy URI used to encrypt password. If it is empty the password is not encrypted.
NamespaceUri			List of Namespace URIs
	Uri		Namespace URI
NodeList			List of OPC UA Variables for access from *WebSocket Server*.
	Node		OPC UA Variables
	Attr	Value-Name	Name of the variable to access with JSON API
	Attr	NodeId	ID of the corresponding Variable in the OPCUA Server
	Attr	ValueType	Type of the OPC UA Variable. Use OPC UA names
	Attr	Array	Equals 1 if the variable is an array
		MetaData	Additional data that can be available through JSON API.

Note: *NodeId* has the format common for OPC UA standard (e.g. "i=208;ns=0"), but be careful! "ns" means the namsespace index in *NapespaceUri* list not the name space of the server.

1.2.3 WebSocket Gateway

The **WebSocket Gateway** provides a JSON API via bidirectional WebSocket protocol for access to OPC UA server Services. Its configuration starts with XML tag *WebSocketGateway* in **OpcUaWebServerModel.xml**.

Default Configuration

```
<WebGateway>
  <Address>0.0.0.0</Address>
    <Port>8082</Port>
</WebGateway>
```

Configuration Settings

XML tag	Description
IP Address	IP address bound by the WebSocket gateway
Port	Port bound by the WebSocket gateway

1.3 WebSocket Server JSON API

With **OpcUaWebServer** you can use a simple JSON API to access to OPC UA data via Internet.

1.3.1 Message Format

All the JSON messages have the following structures:

Field		Description
Header		
	MessageType	The type of the JSON message. It can have the following values:
		READ_REQUEST
		READ_RESPONSE
		WRITE_REQUEST
		WRITE_RESPONSE
		VALUELIST_REQUEST
		VALUELIST_RESPONSE
		VALUEINFO_REQUEST
		VALUEINFO_RESPONSE
		MONITORSTART_REQUEST
		MONITORSTART_RESPONSE
		MONITORSTOP_REQUEST
		MONITORSTOP_RESPONSE
		MONITORUPDATE_MESSAGE
		HISTORICALREAD_REQUEST
		HISTORICALREAD_RESPONSE
	ClientHandler	An identifier of the message which is set by the client in the request. The server copies its value to the corresponding response.
	[StatusCode]	Is sent if an error occurs by processing the request from the client.
Body		

All the JSON messages are described by using our *Notation*.

1.3.2 Value List

With *VALUELIST_REQUEST* request the client can get the list of OPC UA Variables described in *OpcUaClient Configuration Settings*.

Value List Request

Field		Description
Header		
	MessageType	Must be *VALUELIST_REQUEST*.
	ClientHandler	See *Message Format*.
Body		Empty.

Value List Response

Field		Description
Header		
	MessageType	Must be *VALUELIST_RESPONSE*.
	ClientHandler	See *Message Format*.
Body		
	@Variables	The list of the variable names.

Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.

Example in Python

```
import websocket
import json

msg = {
  'Header': {
    'MessageType':'VALUELIST_REQUEST',
    'ClientHandle':'1'
  },
  'Body': {}
}

ws = websocket.create_connection('ws://127.0.0.1:8081')
ws.send(json.dumps(msg))
resp = ws.recv()
json.loads(resp)   #=> {
                   # 'Header': {
                   #   'ClientHandle': '1',
                   #   'MessageType': 'VALUELIST_RESPONSE'
                   # },
                   #
                   # 'Body': {
                   #   'Variables': [
                   #     'Var1',
                   #     'Var2', ..
                   #   ]
                   # }
                   #}
```

1.3.3 Value Info

Knowing the variable names the client can get the information about the configuration of the variables by using *VALUEINFO_REQUEST* request.

Value Info Request

Field		Description
Header		
	MessageType	Must be *VALUEINFO_REQUEST*.
	ClientHandler	See *Message Format*.
Body		
	@Variables	The list of the variable names.

Value Info Response

Field			Description
Header			
	MessageType		Must be *VALUEINFO_RESPONSE*.
	ClientHandler		See *Message Format*.
Body			
	@Variables		The List of objects representing the configuration.
		[StatusCode]	The error occurs by getting the configuration.
		Variable	The name of the variable.
		Type	The type of the variable.
		IsArray	Equals 'true' if the variable is an array.
		MetaData	Additional information described in the configuration as metadata.

Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.
BadAttributeInvalid	The server failed decode the body of the message.
BadNodeIdUnknown	The variable name isn't found in the server configuration.

Example in Python

```
import websocket
import json

msg = {
    'Header': {
      'MessageType':'VALUEINFO_REQUEST',
      'ClientHandle':'1'
    },
    'Body': { 'Variables' : ['Boolean']}
 }


ws = websocket.create_connection('ws://127.0.0.1:8081')
ws.send(json.dumps(msg))
resp = ws.recv()
json.loads(resp)  #=> {
```

(continues on next page)

(continued from previous page)

```
                    # 'Header': {
                    #   'ClientHandle': '1',
                    #   'MessageType': 'VALUEINFO_RESPONSE'
                    # },
                    #
                    # 'Body': {
                    #   'Variables': [
                    #     {
                    #       'Variable': 'Boolean',
                    #       'Type': 'Boolean',
                    #       'IsArray': 'false',
                    #       'MetaData': {
                    #         'DisplayName': ' Switch',
                    #          'Limits': {'Max': ' 1', 'Min': ' 0'}
                    #          }
                    #     }
                    #   ]
                    # }
                    #}
```

1.3.4 Read Value

To read the value of a variable, the client must use *READ_REQUEST* request.

Read Request

Field		Description
Header		
	MessageType	Must be *READ_REQUEST*.
	ClientHandler	See *Message Format*.
Body		
	Variable	The variable to read.

Read Response

Field			Description
Header			
	MessageType		Must be *READ_RESPONSE*.
	ClientHandler		See *Message Format*.
Body			
	Value		
		Body	The value of the variable.
		Type	The type of the variable.
	[Status]		The OPC UA status of the variable if it is not *Success*.
	SourceTimestamp		The time of the value given by the source in ISO 8601 format. Example: “2015-09-06T09:03:21Z”
	ServerTimestamp		The time of the value given by the server in ISO 8601 format. Example: “2015-09-06T09:03:21Z”

Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.
BadAttributeInvalid	The server failed decode the body of the message.
BadNodeIdUnknown	The variable name isn't found in the server configuration.
BadSessionClosed	The connection with OPC UA server is lost.

Example in Python

```
import websocket
import json

msg = {
    'Header': {
      'MessageType':'READ_REQUEST',
      'ClientHandle':'1'
    },
    'Body': { 'Variable' : 'Boolean'}
 }

ws = websocket.create_connection('ws://127.0.0.1:8081')
ws.send(json.dumps(msg))
resp = ws.recv()
json.loads(resp)   #=> {
                   # "Header": {
                   #    "MessageType": "READ_RESPONSE",
                   #    "ClientHandle": "1"
                   # },
                   # "Body": {
                   #    "Value": {
                   #        "Type": 1,
                   #        "Body": true
                   #    },
                   #    "SourceTimestamp": "2019-07-26T11:10:20Z",
                   #    "ServerTimestamp": "2019-07-26T11:10:20Z"
                   # }
                   #}
```

1.3.5 Historical Read

To read the historical values of a variable, the client must use *HISTORICALREAD_REQUEST* request.

Historical Read Request

Field		Description
Header		
	MessageType	Must be *HISTORICALREAD_REQUEST*.
	ClientHandler	See *Message Format*.
Body		
	Variable	The variable to read
	StartTime	Beginning of period to read in ISO 8601 format. Example: "2015-09-06T09:03:21Z"
	Endtime	End of period to read in ISO 8601 format. Example: "2015-09-06T09:03:21Z"

Historical Read Response

Field			Description
Header			
	MessageType		Must be *HISTORICAL-READ_RESPONSE*.
	ClientHandler		See *Message Format*.
Body			
	@DataValues		The history data
		Value	
		Body	The value of the variable.
		Type	The type of the variable.
		[Status]	The OPC UA status of the variable if it is not *Success*.
		SourceTimestamp	The time of the value given by the source in ISO 8601 format. Example: "2015-09-06T09:03:21.237123"
		ServerTimestamp	The time of the value given by the server in ISO 8601 format. Example: "2015-09-06T09:03:21.237123"

Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.
BadAttributeInvalid	The server failed decode the body of the message.
BadNodeIdUnknown	The variable name isn't found in the server configuration.
BadSessionClosed	The connection with OPC UA server is lost.

Example in Python

```
import websocket
import json

msg = {
    'Header': {
      'MessageType':'HISTORICALREAD_REQUEST',
      'ClientHandle':'1'
    },
    'Body': { 'Variable' : 'Boolean'}
 }


ws = websocket.create_connection('ws://127.0.0.1:8081')
ws.send(json.dumps(msg))
resp = ws.recv()
json.loads(resp)   #=> {
                   # "Header": {
                   #    "MessageType": "READ_RESPONSE",
                   #    "ClientHandle": "1"
                   # },
                   # "Body": {
                   #    "Value": {
                   #        "Type": "Boolean",
                   #        "Body": true
                   #    },
                   #    "SourceTimestamp": "2019-07-26T11:10:20Z",
                   #    "ServerTimestamp": "2019-07-26T11:10:20Z"
                   # }
                   #}
```

1.3.6 Write Value

To write the value of a variable, the client should use *WRITE_REQUEST* request.

Write Request

Field				Description
Header				
	MessageType			Must be *WRITE_REQUEST*.
	ClientHandler			See *Message Format*.
Body				
	Variable			The name of the variable to write
	Value			
		Value		
			Body	The value of the variable.
			Type	The type ID of the variable.
		[Status]		The OPC UA status of the variable.
		[SourceTimestamp]		The time of the value given by the source in ISO 8601 format. Example: "2015-09-06T09:03:21Z"
		[ServerTimestamp]		The time of the value given by the server in ISO 8601 format. Example: "2015-09-06T09:03:21Z"

Write Response

Field		Description
Header		
	MessageType	Must be *WRITE_RESPONSE*.
	ClientHandler	See *Message Format*.
Body		
	[Status]	The OPC UA status of the variable if it is not *Success*.

Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.
BadAttributeInvalid	The server failed decode the body of the message.
BadNodeIdUnknown	The variable name isn't found in the server configuration.
BadSessionClosed	The connection with OPC UA server is lost.

Example in Python

```
import websocket
import json

msg = {
   'Header': {
    'ClientHandle': '1',
    'MessageType': 'WRITE_REQUEST'
   },
   'Body': {
     'Variable': 'Int32Test',
     'Value': {
```

(continues on next page)

(continued from previous page)

```
        'Value': {
          'Body': '555',
          'Type': '8'
        }
      }
    }
}

ws = websocket.create_connection('ws://127.0.0.1:8081')

resp = ws.recv()
json.loads(resp)   #=> {
                   # "Header": {
                   #     "MessageType": "WRITE_RESPONSE",
                   #     "ClientHandle": "1"
                   # },
                   # "Body": {}
                   #}
```

1.3.7 Monitoring

OpcUaWebServer provides a subscription model. The client can subscribe to a variable by using *MONITORSTART_REQUEST*. After that the server sends the data of the variable as *MONITORUPDATE_MESSAGE* only when it changes. Finally, the client must stop monitoring the value and send *MONITORSTOP_REQUEST* request.

Monitor Start Request

Field		Description
Header		
	MessageType	Must be *MONITORSTART_REQUEST*.
	ClientHandler	See *Message Format*.
Body		
	Variable	The variable to read.

Monitor Start Response

Field		Description
Header		
	MessageType	Must be *MONITORSTART_RESPONSE*.
	ClientHandler	See *Message Format*.
Body		
	[Status]	The OPC UA status if it is not *Success*.

Monitor Start Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.
BadAttributeInvalid	The server failed decode the body of the message.
BadNodeIdUnknown	The variable name isn't found in the server configuration.

Monitor Update Message

Field			Description
Header			
	MessageType		Must be *MONITORUPDATE_MESSAGE*.
	ClientHandler		See *Message Format*.
Body			
	Value		
		Body	The value of the variable.
		Type	The type of the variable.
	[Status]		The OPC UA status of the variable if it is not *Success*.
	SourceTimestamp		The time of the value given by the source in ISO 8601 format. Example: "2015-09-06T09:03:21Z"
	ServerTimestamp		The time of the value given by the server in ISO 8601 format. Example: "2015-09-06T09:03:21Z"

Monitor Stop Request

Field		Description
Header		
	MessageType	Must be *MONITORSTOP_REQUEST*.
	ClientHandler	See *Message Format*.
Body		
	Variable	The variable to read.

Monitor Stop Response

Field		Description
Header		
	MessageType	Must be *MONITORSTOP_RESPONSE*.
	ClientHandler	See *Message Format*.
Body		
	[Status]	The OPC UA status if it is not *Success*.

Monitor Stop Status Codes

Status Code	Description
BadInternalError	The server failed to process the request due to internal error.
BadAttributeInvalid	The server failed decode the body of the message.
BadNoEntryExists	The variable name isn't found in the server configuration.

Example in Python

```
import websocket
import json

msg = {
    'Header': {
      'MessageType':'MONITORSTART_REQUEST',
      'ClientHandle':'1'
    },
    'Body': { 'Variable' : 'Boolean'}
 }

ws = websocket.create_connection('ws://127.0.0.1:8081')
ws.send(json.dumps(msg))
resp = ws.recv()
json.loads(resp)   #=> {
                   # 'Header': {
                   #   'MessageType': 'MONITORSTART_RESPONSE',
                   #   'ClientHandle': '1'},
                   #  'Body': ''
                   # }

resp = ws.recv()
json.loads(resp)   #=> {
                   # "Header": {
                   #    "MessageType": "MONITORUPDATE_MESSAGE",
                   #    "ClientHandle": "1"
                   # },
                   # "Body": {
                   #    "Value": {
                   #        "Type": 1,
                   #        "Body": true
                   #    },
                   #    "SourceTimestamp": "2019-07-26T11:10:20Z",
                   #    "ServerTimestamp": "2019-07-26T11:10:20Z"
                   # }
                   #}
msg = {
    'Header': {
      'MessageType':'MONITORSTOP_REQUEST',
      'ClientHandle':'1'
    },
    'Body': { 'Variable' : 'Boolean'}
}

ws.send(json.dumps(msg))
resp = ws.recv()
```

(continues on next page)

(continued from previous page)

```
json.loads(resp)   #=> {
                   # 'Header': {
                   #   'MessageType': 'MONITORSTOP_RESPONSE',
                   #   'ClientHandle': '1'},
                   #  'Body': ''
                   # }
```

1.3.8 Notation

In this documentation we use the following notation to describe the JSON data:

Notation	Description
FieldName	The required scalar field in JSON with name *FieldName*
[OptionalFieldName]	The optional scalar field in JSON with name *OptionalFieldName*
@ArrayField	The array in JSON with name *ArrayField*

1.4 WebSocket Gateway

CHAPTER 2

Development Status

ASNeG OPC UA Web Server is in the development stage. Moreover it dependents on OpcUaStack 4, which is also being developed and not released.

CHAPTER 3

Contribution

Our goal is to let people use OPC UA technology easily and for free. As an open source project we can't reach the goal without a strong community. So we will appreciate any help to the project.

If you feel eager to help the project, take a look at Contributing to ASNeG and join us on Slack

CHAPTER 4

Indices and tables

- glossary