**Getting started - Nimbus setup**

The Nimbus WebServices use the built-in ActiveX Service and communicates using a single TCP-socket between the WebService and the Nimbus Alarm Server.

Enable Nimbus ActiveX services in the *Nimbus\_Server.ini* file, section *[ActiveX],* parameter *EnableActiveXServer=1*.

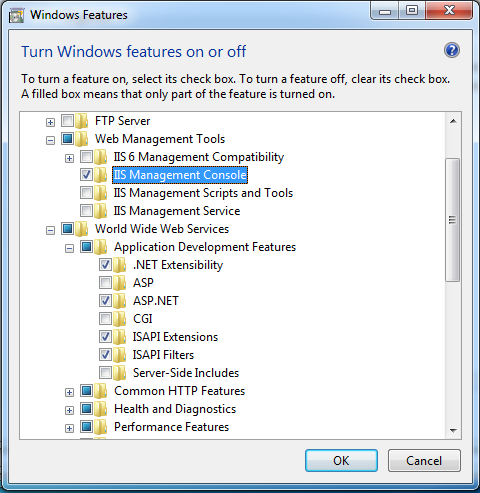
You can open the file explorer in the correct folder by right-clicking the projectpath down below in the Nimbus Explorer window and select 'Open Explorer at this location'.

Don't forget to restart Nimbus Alarm Server when the Nimbus\_Server.ini changes are saved.

You will need Nimbus Alarm Server version 3.00.11 to use the WebServices.

**Getting started - WebService setup**

Install IIS and the *ASP.NET* feature



Copy the *Nimbus Web Service* files to a folder in your website, ex *C:\inetpub\wwwroot\nimbusws*

*.\ NimbusWS.asmx  
.\* *Global.asax  
. \Web.config  
.\bin\NimbusWS.dll  
.\bin\NimbusWS.pdb*

Edit the *Web.Config* file and change the *ServerAddress* and *ServerPort* to match the settings in *Nimbus\_Server.ini*. It defaults to a local installation at 127.0.0.1.

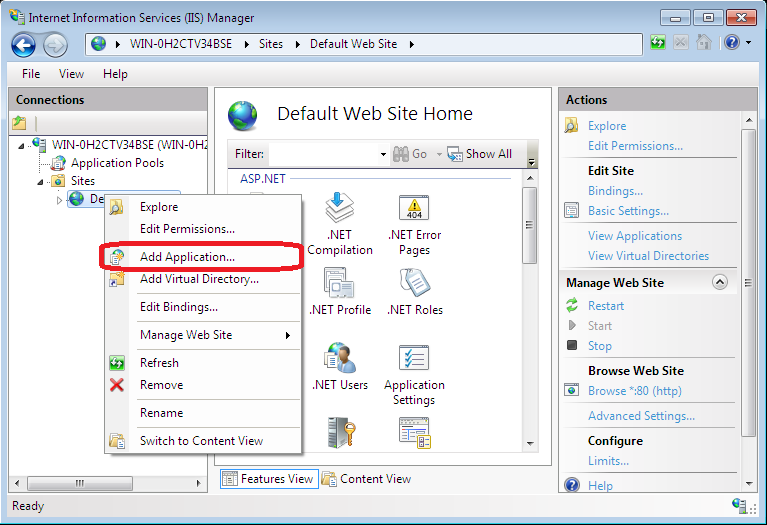
If it is a redundant Nimbus installation, both Nimbus Alarm Server's addresses should be set, ex *ServerAddress="192.168.123.82,192.168.123.83"* and *ServerPort="58658,58658"*

The DNS lookup name could be used instead of the IP

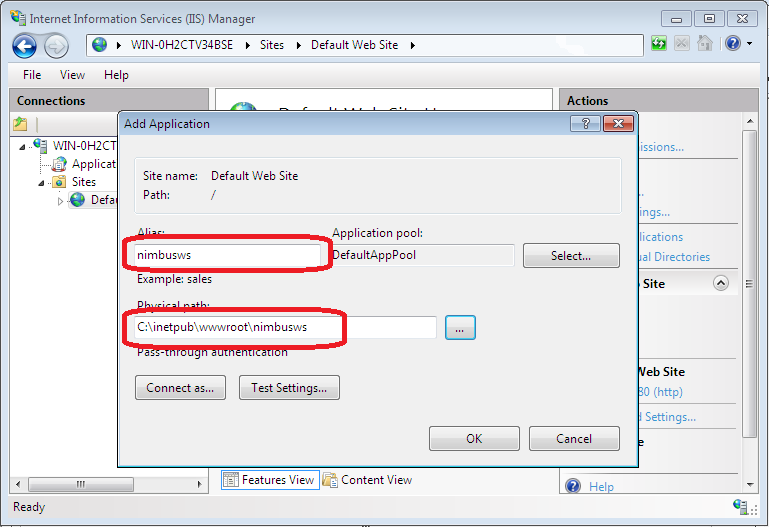
The WebService will automatically connect to the Nimbus Alarm Server in a redundant configuration that is the first to respond and will also automatically try the other server if the connection is broken.

All changes will automatically be replicated to the redundant partner no matter to which server the WebService is connected.

Open IIS Manager

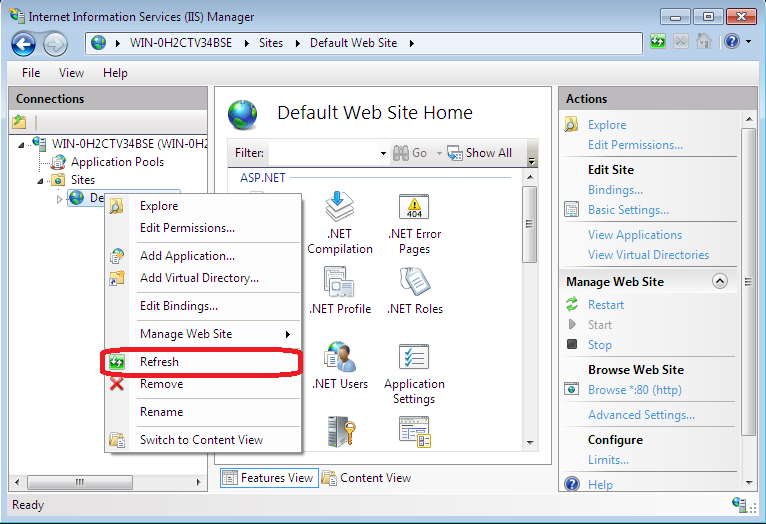


Right-click your web site (i.e. *Default Web Site*), select *Add Application*



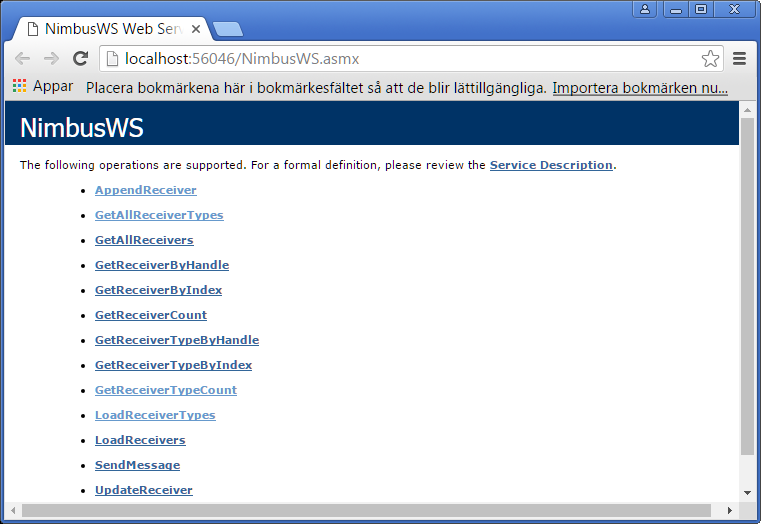
Fill in the info about where you put the files

Right-click your web site (i.e. *Default Web Site*), select *Refresh*



That's it

Open the webservice using *http://localhost/nimbusws/NimbusWS.asmx*



Try to invoke the methods (above example is not complete, methods are described later on)



Example of the *GetAllReceivers* method, which returns an array of structs containing all receivers



Example of the *GetAllReceiverTypes* method, which returns an array of structs containing all receivertypes

**Methods**

**IsConnectedToNimbusServer ()**

*Description:*Get the connection status of the Nimbus Alarm Server TCP socket. The status is only ok if the connection is established and all initial data has been retrieved properly

*Return value:*A bool indicating if we are connected and fine (true) or if we are not (false).

**GetEventLog ()**

*Description:*Enumerates the WebService eventlog.

*Return value:*An array of strings (last 100 messages)

**LoadReceivers()**

*Description:*Retrieves all Receivers into the WebService internal collection from Nimbus Alarm Server. The Receivers are automatically retrieved when the WebService is initialized first time, and will automatically be updated from Nimbus Alarm Services if something is changed.

*Return value:*An int containing the number of retrieved Receivers, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverCount ()**

*Description:*Returns the number of Receivers in the WebService collection

*Return value:*An int containing the number of Receivers in the WebService collection, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than *the GetReceiverCount()* value.

*Return value:*A *Receiver* struct

**GetReceiverByHandle (int Handle)**

*Description:*The handle value corresponds to the filename enumeration (ex *Receiver\_1234.dat*).

*Return value:*A *Receiver* struct

**GetReceiverByName (string ReceiverName)**

*Description:*Lookup a receiver using non-case sensitive search in the Receiver collection.

*Return value:*A *Receiver* struct

**GetAllReceivers ()**

*Description:*Enumerates all receivers.

*Return value:*A list of *Receiver* structs

**EnableOrDisableReceiver (int Handle, string ReceiverName, bool EnableReceiver)**

*Description:*Enable or disable a Receiver.

Only one of the two first parameters are needed. Primary the *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver.

*Return value:*A bool indicating success (true) or failure (false)

**AddReceiver (cReceiver Receiver)**

*Description:*Adds a new Receiver to the already WebService collection and will it to Nimbus Alarm Server.

The Receiver struct should have been filled properly, however the fields *Handle* and *FileDateTimeStamp* are ignored.

If the *ReceiverTypeHandle* field is set to -1 the field *ReceiverTypeName* must be set. If *ReceiverTypeHandle*>= 0 then the *ReceiverTypeName* field is ignored.

*Return value:*An int containing the handle of the newly added receiver. If the receiver could not be added (ex it already exists a receiver with the same name) the handle returned is -1

**UpdateReceiver (cReceiver Receiver)**

*Description:*Update a Receiver in the WebService collection and send it to Nimbus Alarm Server.

The Receiver struct should have been filled properly, however the field *FileDateTimeStamp* is ignored.

If the *ReceiverTypeHandle* field is set to -1 the field *ReceiverTypeName* must be set. If *ReceiverTypeHandle*>= 0 then the *ReceiverTypeName* field is also ignored.

The *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver. The receiver name may only be changed if *Handle* is valid.

*Return value:*A bool indicating success (true) or failure (false)

**DeleteReceiver (int Handle, string ReceiverName)**

*Description:*Delete a Receiver from the receivers collection in the WebService and send the update to Nimbus Alarm Server.

Only one of the parameters are needed. Primary the *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver.

*Return value:*A bool indicating success (true) or failure (false)

**CloneReceiver (int Handle, string ReceiverName, string NewReceiverName)**

*Description:*Create a new receiver based upon an old receiver and clone all its properties (beside *Handle* and *ReceiverName*).

Only one of the first two parameters are needed. Primary the *Handle* field will be used to find the Receiver to clone. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver to clone.

*Return value:*An int containing the handle of the newly added receiver. If the receiver could not be added (ex it already exists a receiver with the same name) the handle returned is -1

**LoadProfiles()**

*Description:*Retrieves all Profiles into the WebService internal collection from Nimbus Alarm Server. The Profiles are automatically retrieved when the WebService is initialized first time, and will automatically be updated from Nimbus Alarm Services if something is changed.

*Return value:*An int containing the number of retrieved Profiles, -1 if we are not connected to Nimbus Alarm Server

**GetProfileCount ()**

*Description:*Returns the number of Profiles in the WebService collection

*Return value:*An int containing the number of Profiles in the WebService collection, -1 if we are not connected to Nimbus Alarm Server

**GetProfileByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than *the GetProfileCount()* value.

*Return value:*A *Profile* struct

**GetProfileByHandle (int Handle)**

*Description:*The handle value corresponds to the filename enumeration (ex *Profile\_1234.dat*).

*Return value:*A *Profile* struct

**GetProfileByName (string ProfileName)**

*Description:*Lookup a profile using non-case sensitive search in the Profile collection.

*Return value:*A *Profile* struct

**GetAllProfiles ()**

*Description:*Enumerates all profiles.

*Return value:*A list of *Profile* structs

**EnableOrDisableProfile (int Handle, string ProfileName, bool EnableProfile)**

*Description:*Enable or disable a Profile.

Only one of the two first parameters are needed. Primary the *Handle* field will be used to find the Profile. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile.

*Return value:*A bool indicating success (true) or failure (false)

**AddProfile (cProfile Profile)**

*Description:*Adds a new Profile to the WebService collection and sends it to Nimbus Alarm Server.

The Profile struct should have been filled properly, however the fields *Handle* and *FileDateTimeStamp* are ignored.

Dayschedule only needs to be filled if dayschedules are used (UseTimeSchedule = true)

*Return value:*An int containing the handle of the newly added profile. If the profiles could not be added (ex it already exists a profiles with the same name) the handle returned is -1

**UpdateProfile (cProfile Profile)**

*Description:*Update a Profile in the WebService collection and send it to Nimbus Alarm Server.

The Profile struct should have been filled properly, however the field *FileDateTimeStamp* is ignored.

The *Handle* field will be used to find the Profile. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile. The profile name may only be changed if *Handle* is valid.

*Return value:*A bool indicating success (true) or failure (false)

**DeleteProfile (int Handle, string ProfileName)**

*Description:*Delete a Profile from the profiles collection in the WebService and send the update to Nimbus Alarm Server.

Only one of the parameters are needed. Primary the *Handle* field will be used to find the Profile. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile.

*Return value:*A bool indicating success (true) or failure (false)

**CloneProfile (int Handle, string ProfileName, string New ProfileName)**

*Description:*Create a new profile based upon an old profile and clone all its properties (beside *Handle* and *ProfileName*).

Only one of the first two parameters are needed. Primary the *Handle* field will be used to find the profile to clone. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile to clone.

*Return value:*An int containing the handle of the newly added profile. If the profile could not be added (ex it already exists a profile with the same name) the handle returned is -1

**LoadReceiverTypes()**

*Description:*Retrieves the ReceiverTypes from Nimbus Alarm Server into the WebService collection. The ReceiverTypes are automatically retireved when the WebService is initialized first time.

*Return value:*An int containing the number of ReceiverTypes in the collection, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverTypeCount ()**

*Description:*Returns the number of ReceiverTypes in the collection

*Return value:*An int containing the number of ReceiverTypes in the collection, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverTypeByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than *the GetReceiverTypeCount()* value.

*Return value:*A ReceiverType struct

**GetReceiverTypeByHandle (int Handle)**

*Description:*The handle value corresponds to the filename enumeration (ex *ReceiverType\_0021)*

*Return value:*A *ReceiverType* struct

**GetReceiverTypeByName (string ReceiverTypeName)**

*Description:*Lookup a receivertype using non-case sensitive search in the ReceiverType collection.

*Return value:*A *ReceiverType* struct

**GetAllReceiverTypes ()**

*Description:*Enumerates all ReceiverTypes.

*Return value:*A list of *ReceiverType* structs

**LoadSCADASystems()**

*Description:*Supported SCADA Systems are automatically retrieved from Nimbus Alarm Server when the WebService is initialized.

*Return value:*An int containing the number of SCADASystems in the collection, -1 if we are not connected to Nimbus Alarm Server

**GetSCADASystemCount ()**

*Description:*Returns the number of SCADASystems in the collection

*Return value:*An int containing the number of loaded SCADASystems, -1 if we are not connected to Nimbus Alarm Server

**GetSCADASystemByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than the *GetSCADASystemCount()* value.

*Return value:*A *SCADASystem* struct

**GetSCADASystemByHandle (int Handle)**

*Description:*The handle value for the SCADASystem. The Index value and the *Handle* value are mostly pretty close (or actually the same for some SCADASystems) but they should not be considered to be equal.

*Handle* must be used when a Profile is updated or created (if *ScanSpecificSystems* member andthe *System* listmembers are used).

*Return value:*A *SCADASystem* struct

**GetAllSCADASystems ()**

*Description:*Enumerates all SCADASystems.

*Return value:*A list of *SCADASystem* structs

**SendMessage (int ReceiverHandle, string ReceiverName, string Message, int DestinationServer)**

*Description:*Place a message in the alarm event queue, destined for the Receiver in *ReceiverHandle* or *ReceiverName*

Only one of the parameters *ReceiverHandle* or *ReceiverName* are needed. Primary the *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver.

The *DestinationServer* parameter is only used in a redundant configuration:  
  
*DestinationServer* = 0 -> Send from the primary Nimbus Server only  
*DestinationServer* = 1 -> Send from the secondary Nimbus Server only  
*DestinationServer* = 2 -> Send from the active Nimbus Server only  
*DestinationServer* = 3 -> Send from both Nimbus Servers  
  
In a single server configuration, the parameter should be set to 0

*(As for now (June 2017 Nimbus release 3.00.11) the DestinationServer has no function)*

*Return value:*An int indicating the message id used internally in Nimbus Alarm Server or -1 if it failed. This message id (event id) also is used in the Nimbus log files. The return value only reflects if the message was successfully placed in the queue or not, it does not indicate the result of the sending process.

**QueueAlarm (cAlarmEvent AlarmEvent, int DestinationServer)**

*Description:*Place an alarm event in the alarm event queue. The AlarmEvent will be treated as an alarm arriving from the SCADA system specified in the *SCADASystemHandle* parameter.

The SCADA System handles are sort of static and will probably never be changed, for example *Generic File* will always have handle *36*.

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (June 2017 Nimbus release 3.00.11) the DestinationServer has no function)*

*Return value:*A bool indicating success (true) or failure (false). The return value only reflects if the alarm was successfully placed in the queue or not. It does not indicate the result of the sending process.

**AbortAll (int DestinationServer)**

*Description:*Abort all queued events waiting to be sent. In a redundant system the abort function is performed only on the primary server.

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (June 2017 Nimbus release 3.00.11) the DestinationServer has no function)*

*Return value:*A bool indicating success (true) or failure (false).

**Structs and classes**

**ReceiverType**

public class cReceiverType

{

publicstring ReceiverTypeName;

publicint Handle;

publicstring Algorithm;// Type of algoithm to be used when sending

publicint Limit;// Max number of chars in messages

publicstring SettingFileName;

publicstring TemplateFileName;

publicstring Country;

}

**Receiver**

public class cReceiver

{

publicconstint MAX\_RECEIVERSETTINGS = 4;

public string ReceiverName;

public string BackupReceiver;

public int Handle;

public bool Enabled;

public DateTime FileDateTimeStamp;// Timestamp of last change, read only

public int KeepAliveInterval;

public int ReceiverTypeHandle

public string ReceiverTypeName;

public string[] Setting; // Setting fields depends of the receiver type

}

**Profile**

public class cProfile

{

public const int MAX\_CONDITIONS = 6;

public const int MAX\_SYSTEMTYPES = 64;

public string ProfileName;

public int Handle;

public bool Enabled;

publi cbool IncludeInactiveAlarms;

publi cbool IncludeActiveAlarms;

publi cbool IncludeAckedAlarms;

// If true also timeschedule is used as filter criteria

public bool UseTimeSchedule;

// List of parallell receivers

public string[] ParallellReceivers;

// List of sequential receivers

public cSequentialReceiver[] SequentialReceivers;

// The filter criterias (Tag, Area, Name etc)

// Max 6 items (usually named t0..t5), unallocated items are set to ’\*’

public string[] Conditions;

public bool TimeSchedule\_ViewDayType;

public bool Calendar\_ViewTimeAxis;

public bool UseExternalTag;

public string ExternalTagName;

public bool IgnoreExternalTagInOtherProfiles;

public bool UseExternalFile;

public string ExternalFileName;

public string ExternalFileText;

public bool UseAutoBlock;

public int AutoBlockTresholdEvents;

public int AutoBlockTresholdTime;

public int AutoBlockTime;

// true if this profile should be applied to specific SCADA systems

// If true the specific SCADA system handles are specified in the System

// list

public bool ScanSpecificSystems;

// List of SCADA System handles this profile will be used for if

// ScanSpecificSystems is true. Order of the handles are irrelevant

public List<int> System;

// 0 – Throw alarm (standard behaviour)

// 1 – Delay sending until on duty time is passed

// 2 – As 1 but throw queued event if it becomes inactive

// 3 – As 1 but throw queued event if it becomes acked

// 4 – As 1 but throw queued event if it becomes inactive or acked

public int OffDutyAlarmBehaviour;

// Last profile update timestamp (read only)

public DateTime FileDateTimeStamp;

// Max 9 days may be defined (Weekdays + Holiday evening

// + Holiday)

public List<cDaySchedule> DaySchedule;

// If calender should be used or not (global calender cannot be

// changed using webservices

public bool UseCalendar;

}

public struct cSequentialReceiver

{

// Receiver name

public string ReceiverName;

// true if this receiver is needed to manually confirm alarm

// reception

public bool RequestConfirmation;

}

public struct cDaySchedule

{

// If days schedule is to be used. The schedule 0..6 (mo..su)

// in DaySchedule are always enabled

public bool InUse;

public bool Grouped;

// Start/stop periods

public List<cOnOff>Periods;

}

public struct cOnOff

{

// Start/stop times in DaySchedule[]

public int StartTime;// 0-1440 equals 00:00..24:00 (minutes since midnigth)

public int StopTime;// 0-1440 equals 00:00..24:00 (minutes since midnigth)

}

**SCADASystem**

public class cSCADASystem

{

public string Name;

public int Handle;

public string Description;

}

**AlarmEvent**

publicstructcAlarmEvent

{

// 0 - Inactive (Normal), 1 - Active (Alarm), 2 – Acked

publicint EventType;

// Alarm date in format YYYY-MM-DD, if omitted it is set to now

publicstring Date;

// Alarm date in format HH:NN:SS, if omitted it is set to now

publicstring Time;

publicstring Tag; // [t0]

publicstring Area; // [t1]

publicstring Category; // [t2]

publicstring Name; // [t3]

publicstring Description; // [t4]

publicstring Status; // [t5]

publicint SCADASystemHandle; // Simulated SCADA system

}