



Client Development Guide & Tutorial

Tested? You want it?

[License](#) [Model](#) [Prices](#) [Quotation](#) [Order](#) [Now](#)

C# |VB|

The Client Frame

A Client for OPC UA

1. Add reference to the **Opc.UaFx.Advanced** Client Namespace:

```
using Opc.UaFx.Client;
```

2. Create an instance of the OpcClient class with the address of the Server:

```
var client = new OpcClient("opc.tcp://localhost:4840/");
```

3. Build a connection to the Server and start a session:

```
client.Connect();
```

4. Your code to interact with the Server:

```
// Your code to interact with the server.
```

5. Close all sessions before closing the application:

```
client.Disconnect();
```

6. Using the using block this looks as follows:

```
using (var client = new OpcClient("opc.tcp://localhost:4840/")) {
    client.Connect();
    // Your code to interact with the server.
}
```

7. Are the Nodes/NodeIds of the Server ...

- **unknown:** Find out which Nodes the Server have
- **known:** Read Values of your Nodes or Write Values of your Nodes

A Client for OPC Classic

1. Add reference to the **Opc.UaFx.Advanced** Classic and Client Namespaces:

```
using Opc.UaFx.Client;
using Opc.UaFx.Client.Classic;
```

2. Create an instance of the OpcClient class with the address of the Server:

```
var client = new OpcClient("opc.com://localhost:48410/<progId>/<classId>");
```

3. Build a connection to the Server and start a session:

```
client.Connect();
```

4. Your code to interact with the Server:

```
// Your code to interact with the server.
```

5. Close all sessions before closing the application:

```
client.Disconnect();
```

6. Using the using block this looks as follows:

```
using (var client = new OpcClient("opc.com://localhost:48410/<progId>/<classId>")) {
    client.Connect();
    // Your code to interact with the server.
}
```

7. Are the Nodes/NodeIds of the Server ...

- **unknown:** Find out which Nodes the Server have
- **known:** Read Values of your Nodes or Write Values of your Nodes

What describes the address of the server opc.com://localhost:48410/<progId>/<classId> ?

- **opc.com** indicates that a connection to an *OPC Classic Server* is to be used.
- **localhost** stands for the name or the IP address of the computer on which the *OPC Classic Server* is running.
- **48410** the optional port number of the *OPC UA Wrapper Server* ¹⁾. If this is missing, a port number will be generated based on the <classId>.
- **<progId>** is a wildcard, replace it with the ProgId of the **DCOM Application Information** of the *OPC Classic Server*,
e.g. 'OPCManager.DA.XML-DA.Server.DA'.
- **<classId>** is a wildcard, replace it with the ClassId (also known as CLSID or AppID) of the **DCOM Application Information** of the *OPC Classic Server*,
e.g. '{E4EBF7FA-CCAC-4125-A611-EAC4981C00EA}'.

How can I determine the <classId> or <progId> of the server?

1. Method: OpcClassicDiscoveryClient

```
using (var discoveryClient = new OpcClassicDiscoveryClient("<host>")) {
    var servers = discoveryClient.DiscoverServers();

    foreach (var server in servers) {
        Console.WriteLine(
            "- {0}, ClassId={1}, ProgId={2}",
            server.Name,
            server.ClassId,
            server.ProgId);
    }
}
```

2. Method: Systeminformation

Perform the following steps on the *OPC Classic Server* computer:

1. *AppID* using the Component Services
 - Control Panel
 - Administrative Tools
 - Component Services
 - expand 'Component Services'
 - expand 'Computers'
 - expand 'My Computer'
 - expand 'DCOM Config'
 - select '<OPC Classic Server>'
 - open 'Properties'

- first tab 'General'
- copy the 'Application ID:' (= *AppID*)
- 2. *ProgID* using the System Registry
 - 'Windows-Key' + 'R'
 - enter 'regedit'
 - click 'Run'
 - expand the key 'Computer'
 - then the key 'HKEY_LOCAL_MACHINE'
 - expand 'SOFTWARE'
 - expand 'SYSWOW6432Node' (only on 64 Bit Systems else continue)
 - expand 'Classes'
 - expand 'CLSID'
 - expand '<classId>'
 - select 'ProgID'
 - copy the '(Default)' value (= *ProgID*) in the 'Data' column

Values of Node(s)

Reading Values

The following types are used: [OpcClient](#), [OpcNodeId](#), [OpcValue](#), [OpcAttribute](#) and [OpcReadNode](#).

The **OpcNodeid** of a Node decides on which Node is to be read. When a Node value is read the current value of the *value attribute* is read by default. The hereby determined **OpcValue** consists, additionally to the actual value, of a time stamp at which the value was identified at the source of the value (**SourceTimestamp**) and of a second time stamp at which the value was registered by the Server (**ServerTimestamp**). If another attribute of the Node shall be read the according **OpcAttribute** has to be **mentioned** at the call of **ReadNode** and the concerning **OpcReadNode** instance.

- Read the value of the value attribute of a single Node:

```
OpcValue isRunning = client.ReadNode("ns=2;s=Machine/IsRunning");
```

- Read the values of the value attribute of several Nodes:

```
OpcReadNode[] commands = new OpcReadNode[] {
    new OpcReadNode("ns=2;s=Machine/Job/Number"),
    new OpcReadNode("ns=2;s=Machine/Job/Name"),
    new OpcReadNode("ns=2;s=Machine/Job/Speed")
};

IEnumerable<OpcValue> job = client.ReadNodes(commands);
```

- Read the value of the DisplayName attribute of a single Node:

```
OpcValue isRunningDisplayName = client.ReadNode("ns=2;s=Machine/IsRunning",
OpcAttribute.DisplayName);
```

- Read the values of the DisplayName attribute of several Nodes:

```
OpcReadNode[] commands = new OpcReadNode[] {
    new OpcReadNode("ns=2;s=Machine/Job/Number", OpcAttribute.DisplayName),
    new OpcReadNode("ns=2;s=Machine/Job/Name", OpcAttribute.DisplayName),
    new OpcReadNode("ns=2;s=Machine/Job/Speed", OpcAttribute.DisplayName)
};

IEnumerable<OpcValue> jobDisplayNames = client.ReadNodes(commands);
```

Writing Values

The following types are used: [OpcClient](#), [OpcNodeId](#), [OpcValue](#), [OpcAttribute](#), [OpcStatus](#), [OpcStatusCollection](#) and [OpcWriteNode](#).

The **OpcNodeId** of a Node decides which Node to write. When a Node value is written, the current value of the *value attribute* is written by default. The hereby set **OpcValue** automatically receives the latest time stamp as the time stamp of the source (**SourceTimestamp**). If another attribute of the Node shall be written the according **OpcAttribute** has to be mentioned at the call of the **WriteNode** or at the concerning **OpcWriteNode** instance.

- Write the value of a single Node:

```
OpcStatus result = client.WriteNode("ns=2;s=Machine/Job/Cancel", true);
```

- Write the values of several Nodes:

```
OpcWriteNode[] commands = new OpcWriteNode[] {
    new OpcWriteNode("ns=2;s=Machine/Job/Number", "0002"),
    new OpcWriteNode("ns=2;s=Machine/Job/Name", "MAN_F01_78910"),
    new OpcWriteNode("ns=2;s=Machine/Job/Speed", 1220.5)
};

OpcStatusCollection results = client.WriteNodes(commands);
```

- Write the value of the DisplayName attribute of a single Node:

```
client.WriteNode("ns=2;s=Machine/IsRunning", OpcAttribute.DisplayName, "IsActive");
```

- Write the values of the DisplayName attribute of several Nodes:

```
OpcWriteNode[] commands = new OpcWriteNode[] {
    new OpcWriteNode("ns=2;s=Machine/Job/Number", OpcAttribute.DisplayName, "Serial"),
    new OpcWriteNode("ns=2;s=Machine/Job/Name", OpcAttribute.DisplayName,
        "Description"),
    new OpcWriteNode("ns=2;s=Machine/Job/Speed", OpcAttribute.DisplayName, "Rotations
per Second")
};

OpcStatusCollection results = client.WriteNodes(commands);
```

Processing Values

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcValue](#), [OpcStatus](#) and [OpcStatusCollection](#).

The **ReadNode** methods always provide an **OpcValue** instance, while the **ReadNodes** methods provide a list of **OpcValue** instances (one **OpcValue** per read Node). The actual value read is in the *Value* property

of the **OpcValue** instances. The result of the read request can be checked via the *Status* property. The timestamp at which the read value has been detected at the source can be retrieved via the *SourceTimestamp* property. Correspondingly, the timestamp at which the read value was detected by the Server can be retrieved via the *ServerTimestamp* property.

- Read the value of a single Node:

```
OpcValue value = client.ReadNode("ns=2;s=Machine/Job/Speed");
```

- Check the result of the read request:

```
if (value.Status.IsGood) {
    // Your code to operate on the value.
}
```

- Retrieve the scalar value of the **OpcValue** instance:

```
int intValue = (int)value.Value;
```

- Retrieve the array value of the **OpcValue** instance:

```
int[] intValues = (int[])value.Value;
```

The **WriteNode** methods always provide an **OpcStatus** instance, while the **WriteNodes** methods provide an **OpcStatusCollection** instance (that contains an **OpcStatus** for every written Node). The result of the write request can thereby be checked via the properties of the **OpcStatus** instance(s).

- Write the scalar value of a single Node:

```
OpcStatus status = client.WriteNode("ns=2;s=Machine/Job/Speed", 1200);
```

- Write the array value of a single Node:

```
int[] values = new int[3] { 1200, 1350, 1780 };
OpcStatus status = client.WriteNode("ns=2;s=Machine/Job/Speeds", values);
```

- Check the result of a write request:

```
if (!status.IsGood) {
    // Your code to handle a failed write operation.
}
```

By using the individual steps to prepare the processing of scalar values and array values, the array value of a Variable Node can be modified as follows:

```
using (var client = new OpcClient("opc.tcp://localhost:4840")) {
    client.Connect();
    OpcValue arrayValue = client.ReadNode("ns=2;s=Machine/Job/Speeds");

    if (arrayValue.Status.IsGood) {
        int[] intArrayValue = (int[])arrayValue.Value;

        intArrayValue[2] = 100;
        intArrayValue[4] = 200;
        intArrayValue[9] = 300;

        OpcStatus status = client.WriteNode("ns=2;s=Machine/Job/Speeds", intArrayValue);

        if (!status.IsGood)
            Console.WriteLine("Failed to write array value!");
    }
}
```

Browsing Nodes

Which Nodes does the Server have?

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcNodeInfo](#), [OpcAttribute](#), [OpcAttributeInfo](#) and [OpcObjectTypes](#).

Starting from a Server whose Address-Space (all available nodes) are still (almost) unknown, it is advisable to inspect the provided Nodes of the Server. Either a graphical OPC UA client like the

OPC Watch

can be used or the Address-Space of the Server can be examined manually.

When using the class **OpcObjectTypes** already predefined Server-Nodes can also be examined by browsing. The "root" of all Nodes of the Server represents the Default-Node called "ObjectsFolder". When browsing is started at the "ObjectsFolder"-Node, the entire Address-Space of the Server can be determined:

```
var node = client.BrowseNode(OpcObjectTypes.ObjectsFolder);
Browse(node);
...
private void Browse(OpcNodeInfo node, int level = )
{
    Console.WriteLine("{0}{1}({2})",
        new string('.', level * 4),
        node.Attribute(OpcAttribute.DisplayName).Value,
        node.NodeId);

    level++;

    foreach (var childNode in node.Children())
        Browse(childNode, level);
}
```

The output via the code snippet shown contains, among other things (in parentheses) the Node-IDs of the

Nodes of the Server. Based on these NodeId's the Nodes of the Server can be addressed directly. Simply pass the NodeId as a string (in double quotes) to the corresponding method of the **OpcClient** class. For example, what it looks like when reading a node value is shown in the section [Reading Values](#).

Inspecting Node by Node

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcNodeInfo](#), [OpcAttribute](#), [OpcAttributeInfo](#), [OpcMethodNodeInfo](#), [OpcArgumentInfo](#) and [OpcObjectTypes](#).

Browsing in the OPC UA can be compared to .NET Reflections. Through browsing it is therefore possible to dynamically determine and examine the entire "Address Space" of a Server. This includes all Nodes, their references towards each other and their Node Types. Browsing is introduced through the **OpcNodeId** of the Node on which the browsing procedure shall be started. Coming from the hereby retrieved **OpcNodeInfo**, browsing can be continued on Child, Parent or Attribute Level.

If a Node is a method-depicting Node, then browsing provides an **OpcMethodNodeInfo** instance by the help of which Input / Output arguments of the method can be examined.

1. Determine the OpcNodeInfo of the desired Node:

```
OpcNodeInfo machineNode = client.BrowseNode("ns=2;s=Machine");
```

2. Use the **Child** or **Children** method to browse the Child-Node:

```
OpcNodeInfo jobNode = machineNode.Child("Job");

foreach (var childNode in machineNode.Children()) {
    // Your code to operate on each child node.
}
```

3. Use the **Attribute** or **Attributes** method to browse the attributes:

```
OpcAttributeInfo displayName = machineNode.Attribute(OpcAttribute.DisplayName);

foreach (var attribute in machineNode.Attributes()) {
    // Your code to operate on each attribute.
}
```

4. In case a Node depicts a method, the Node can be visualized and examined like this:

```
if (childNode.Category == OpcNodeCategory.Method) {
    var methodNode = (OpcMethodNodeInfo)childNode;

    foreach (var argument in methodNode.GetInputArguments()) {
        // Your code to operate on each argument.
    }
}
```

High-Speed Browsing

The following types are used here: [OpcNodeInfo](#), [OpcNodeId](#), [OpcBrowseNode](#), [OpcReferenceTypeOpcBrowseNodeDegree](#) and [OpcBrowseOptions](#).

If a Server provides a very extensive "Address Space", browsing through all nodes is often a bit slow, especially if the large amount of node information also has to be transported via a quite weak hardware or network configuration. The additional load on the Server also leads to a loss of performance if Clients go

through the entire node tree node by node.

To optimize this situation, the SDK offers the option of defining how many nodes and node levels should be examined at the same time. The `OpcBrowseNodeDegree` enumeration is used for this. Depending on the number of nodes required at the same time, the corresponding value of the enumeration is to be specified via the `Degree` property or in the constructor of the `OpcBrowseNode` class. There exists also the option of reducing the user data to a minimum or only to the really relevant data when browsing. The `OpcBrowseOptions` enumeration offers various options for this. It should be noted that at least the `ReferenceTypeId` must be part of the node information. If you want to receive additional node information, such as the `DisplayName`, then the `DisplayName` must also be included (when using the `Options` property).

Browsing can be parameterized and thus further optimized using the following properties:

```
// Create a browse command to browse all hierarchical references.
var browse = new OpcBrowseNode(
    nodeId: OpcNodeId.Parse("ns=2;s=Machine"),
    degree: OpcBrowseNodeDegree.Generation);

// Create a browse command to browse specific types of references.
var browse = new OpcBrowseNode(
    nodeId: OpcNodeId.Parse("ns=2;s=Machine"),
    degree: OpcBrowseNodeDegree.Generation,
    referenceTypes: new[] {
        OpcReferenceType.Organizes,
        OpcReferenceType.HasComponent,
        OpcReferenceType.HasProperty
    });

// Reduce browsing to the smallest possible amount of data.
browse.Options = OpcBrowseOptions.IncludeReferenceTypeId
    | OpcBrowseOptions.IncludeBrowseName;

var node = client.BrowseNode(browse);

foreach (var childNode in node.Children()) {
    // Continue recursively...
}
```

It should also be noted that if nodes that only have a certain relationship to one another (which is expressed using `ReferenceTypes`) are to be examined, browsing can also be further optimized here. The node references relevant for browsing can also be used as parameters for the browse operation (see the code snippet above) so that only the nodes that use one of the specified `ReferenceTypes` are visited.

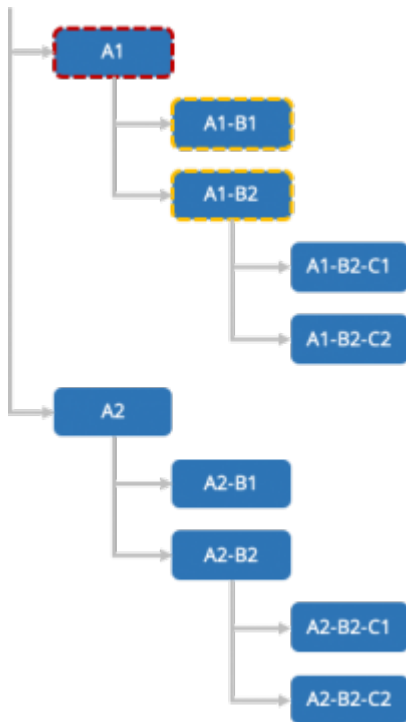
High-Speed Browsing - Details

If a Server provides a very extensive "Address Space", browsing through all nodes is often a bit slow, especially if the large amount of node information also has to be transported via a quite weak hardware or network configuration. The additional load on the Server also leads to a loss of performance if Clients go through the entire node tree node by node.

In such cases it is important to keep the **workload** for the Server and the entire **communication** between Client and Server as **lean** and **as short as possible**. For this purpose, OPC UA offers various mechanisms with which a Client can signal to the Server that many more nodes than just one level are being examined. The Server then has the option to prepare the information and communicate it to the Client in packets. This therefore reduces further internal reprocessing of the node tree on the Server side, always on request

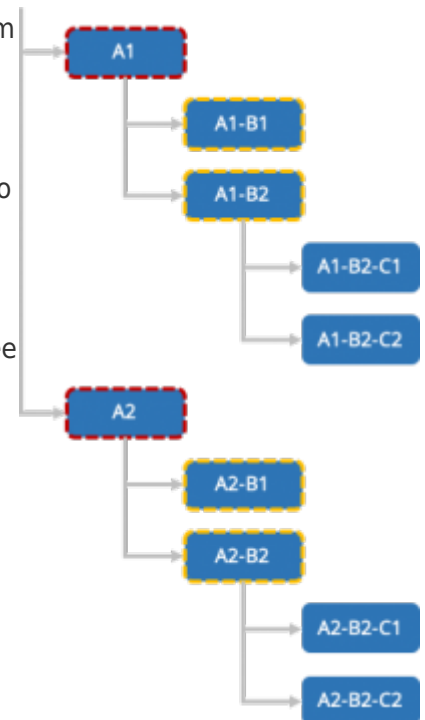
from the Client. In addition, a Client can, with the appropriate logic, query further nodes in advance from the Server, of which it is known that their node information and that of their children and their children-children (and so on) would also be processed.

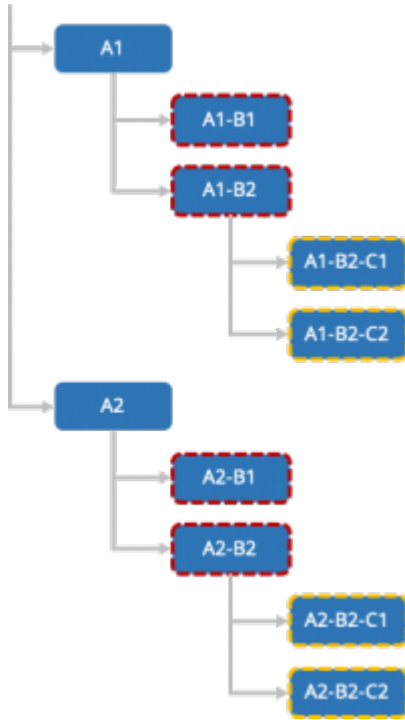
The **SDK offers various parameters** for optimizing the browse operations, which can be used to determine the behavior when browsing. Depending on the browsing process, the SDK uses the parameters to decide which nodes with which information should be prepared in advance on the Server side for the current browse process. The SDK then automatically retrieves these nodes during further browsing, stores them in the memory in advance for the further browsing process and then delivers them from the internal cache instead from the Server.



A value of the `OpcBrowseNodeDegree` enumeration is used to control which nodes are to be automatically requested by the Server at the same time. The value `Self` corresponds to the standard behavior in which only the children of the current node are examined. In the tree on the left, browsing is started at node `A1`. As a result, the Server only delivers the nodes directly below node `A1`, i.e. the nodes `A1-B1` and `A1-B2`. The nodes of all other subtrees, including those of the nodes next to the node `A1`, at which the browse operation was started, are not part of the browse operation.

With the value `Sibling`, all children of the sibling nodes are retrieved from the Server at the same time when the child nodes of the current node are examined. In this case, when the browse operation starts at node `A1`, the SDK not only determines the direct child nodes of node `A1`, but also those of all nodes that are siblings of `A1`. This means that the subtrees from `A2` to `An` are also determined during the browse operation. The SDK temporarily stores the node information obtained in this way and (again) retrieves it from the memory instead from the Server as soon as the developer requires the subtree of e.g. Node `A2`. In the picture on the right you can see that a browse operation - starting from node `A1` - behaves as if you were browsing nodes `A1` to `An` at the same time.





If the value **Generation** is used, all nodes of the same generation are examined at the same time, which means that all nodes of the same level depth are examined. Specifically, this means, as shown in the tree on the left, that if a browse operation is started at the node **A1-B1**, not only its subtree (as when browsing with **OpcBrowseNodeDegree.Self**), but also the subtrees of his siblings (as when browsing with **OpcBrowseNodeDegree.Sibling**) as well as the subtrees of his cousins of the nth degree can also be called up. In this way, browsing behaves as if you were browsing the nodes **A1-B1**, **A1-B2**, **A2-B1** and **A2-B2** at the same time. Here, too, the node information of the subtrees is cached and made available to the developer if required.

It should be noted that the use of the individual **OpcBrowseNodeDegree** values is always associated with a little caution. If, for example, a Server provides a relatively “flat address space” that contains a large number of nodes per subtree, the “Generation” option can lead to relatively long browse operations, which in turn can be very memory-intensive. The same applies to the **Self** option, especially if one has a large number of small subtrees on the same level, which in turn also contain many other child nodes.

The general **recommendation** is therefore that the developer either knows the complexity of the Server in advance and selects the level of browse operations based on this, or in the case of unknown/changing Servers by choosing the level **Sibling** or **Generation** the node information shall be reduced via the Options property. Any further necessary node information can then be called up by the developer using a separate browse operation.

Subscriptions

Subscription Creation

The following types are used here: **OpcClient**, **OpcNodeId**, **OpcSubscribeDataChange**, **OpcSubscribeEvent**, **OpcSubscription**, **OpcMonitoredItem**, **OpcMonitoredItemCollection**, **OpcDataChangeReceivedEventHandler**, **OpcDataChangeReceivedEventArgs**, **OpcEventReceivedEventHandler**, **OpcEventReceivedEventArgs** and **OpcNotification**.

Subscriptions in the OPC UA can be compared to subscriptions to one or more journals in the bundle. Instead of magazines, however, Node Events (= in the OPC UA: Monitored Item) or changes to the Node Values (= in the OPC UA: Monitored Item) are subscribed. For each subscription, it is possible to specify in which interval the notifications (**OpcNotification** instances) of the monitored items should be published (**OpcSubscription.PublishingInterval**). Which Node to subscribe to is determined by the **OpcNodeId** of the Node. By default, the **Value-Attribute** is monitored. If another attribute of the Node is to be monitored, the corresponding **OpcAttribute** have to be specified when calling **SubscribeDataChange** or when using a respective **OpcSubscribeDataChange** instance.

- Subscribe to notifications about changes to the Node Value:

```
OpcSubscription subscription = client.SubscribeDataChange(
    "ns=2;s=Machine/IsRunning",
    HandleDataChanged);
```

- Subscribe to notifications about changes to multiple Node Values:

```
OpcSubscribeDataChange[] commands = new OpcSubscribeDataChange[] {
    new OpcSubscribeDataChange("ns=2;s=Machine/IsRunning", HandleDataChanged),
    new OpcSubscribeDataChange("ns=2;s=Machine/Job/Speed", HandleDataChanged)
};

OpcSubscription subscription = client.SubscribeNodes(commands);
```

- Handle notifications of changes to Node Values:

```
private static void HandleDataChanged(
    object sender,
    OpcDataChangeReceivedEventArgs e)
{
    // Your code to execute on each data change.
    // The 'sender' variable contains the OpcMonitoredItem with the NodeId.
    OpcMonitoredItem item = (OpcMonitoredItem)sender;

    Console.WriteLine(
        "Data Change from NodeId '{0}': {1}",
        item.NodeId,
        e.Item.Value);
}
```

- Subscribe to notifications about Node Events:

```
OpcSubscription subscription = client.SubscribeEvent(
    "ns=2;s=Machine",
    HandleEvent);
```

- Subscribe to notifications about Node Events of multiple Nodes:

```
OpcSubscribeEvent[] commands = new OpcSubscribeEvent[] {
    new OpcSubscribeEvent("ns=2;s=Machine", HandleEvent),
    new OpcSubscribeEvent("ns=2;s=Machine/Job", HandleEvent)
};

OpcSubscription subscription = client.SubscribeNodes(commands);
```

- Handle notifications about Node Events:

```
private void HandleEvent(object sender, OpcEventReceivedEventArgs e)
{
    // Your code to execute on each event raise.
}
```

- Configuration of the **OpcSubscription**:

```
subscription.PublishingInterval = 2000;

// Always call apply changes after modifying the subscription; otherwise
// the server will not know the new subscription configuration.
subscription.ApplyChanges();
```

- Subscribe to notifications of changes to multiple Node Values through a single subscription and set custom values of the Tag property of the **OpcMonitoredItem**:

```
string[] nodeIds = {
    "ns=2;s=Machine/IsRunning",
    "ns=2;s=Machine/Job/Speed",
    "ns=2;s=Machine/Diagnostics"
};

// Create an (empty) subscription to which we will add OpcMonitoredItems.
OpcSubscription subscription = client.SubscribeNodes();

for (int index = 0; index < nodeIds.Length; index++) {
    // Create an OpcMonitoredItem for the NodeId.
    var item = new OpcMonitoredItem(nodeIds[index], OpcAttribute.Value);
    item.DataChangeReceived += HandleDataChanged;

    // You can set your own values on the "Tag" property
    // that allows you to identify the source later.
    item.Tag = index;

    // Set a custom sampling interval on the
    // monitored item.
    item.SamplingInterval = 200;

    // Add the item to the subscription.
    subscription.AddMonitoredItem(item);
}

// After adding the items (or configuring the subscription), apply the changes.
subscription.ApplyChanges();
```

Handler:

```
private static void HandleDataChanged(
    object sender,
    OpcDataChangeReceivedEventArgs e)
{
    // The tag property contains the previously set value.
    OpcMonitoredItem item = (OpcMonitoredItem)sender;

    Console.WriteLine(
        "Data Change from Index {0}: {1}",
        item.Tag,
        e.Item.Value);
}
```

Subscription Filtering

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcSubscribeDataChange](#), [OpcDataChangeFilter](#), [OpcDataChangeTrigger](#), [OpcSubscription](#), [OpcMonitoredItem](#), [OpcMonitoredItemCollection](#), [OpcDataChangeReceivedEventHandler](#), [OpcDataChangeReceivedEventArgs](#) and [OpcNotification](#).

A Client application is notified through a once created subscription only about changes of the Status or of the Value of a Node, by default. However, for a Server to notify the Client about changes to the Node according to certain policies, it is possible to setup a trigger of a notification to evaluate. For this, the desired Trigger is determined via a value of the **OpcDataChangeTrigger** enumeration. The following

examples subscribe to notifications about changes in the Status or the Value or the Timestamp of a Node.

- Subscribe to notifications about changes to the Node Value and Node Timestamps:

```
OpcSubscription subscription = client.SubscribeDataChange(
    "ns=2;s=Machine/IsRunning",
    OpcDataChangeTrigger.StatusValueTimestamp,
    HandleDataChanged);
```

- Subscribe to notifications about changes to multiple Node Values and Node Timestamps:

```
OpcSubscribeDataChange[] commands = new OpcSubscribeDataChange[] {
    new OpcSubscribeDataChange(
        "ns=2;s=Machine/IsRunning",
        OpcDataChangeTrigger.StatusValueTimestamp,
        HandleDataChanged),
    new OpcSubscribeDataChange(
        "ns=2;s=Machine/Job/Speed",
        OpcDataChangeTrigger.StatusValueTimestamp,
        HandleDataChanged)
};

OpcSubscription subscription = client.SubscribeNodes(commands);
```

- Handle notifications of changes to Node Values and Node Timestamps:

```
private static void HandleDataChanged(
    object sender,
    OpcDataChangeReceivedEventArgs e)
{
    // Your code to execute on each data change.
    // The 'sender' variable contains the OpcMonitoredItem with the NodeId.
    OpcMonitoredItem item = (OpcMonitoredItem)sender;

    Console.WriteLine(
        "Data Change from NodeId '{0}': {1} at {2}",
        item.NodeId,
        e.Item.Value,
        e.Item.Value.SourceTimestamp);
}
```

The used **OpcDataChangeTrigger**-Value of the preceding samples can be defined using an instance of the **OpcDataChangeFilter** class as well. Doing so enables the reuse of the once defined filter within multiple subscriptions and monitored items:

```
OpcDataChangeFilter filter = new OpcDataChangeFilter();
filter.Trigger = OpcDataChangeTrigger.StatusValueTimestamp;

OpcSubscription subscriptionA = client.SubscribeDataChange(
    "ns=2;s=Machine/IsRunning",
    filter,
    HandleDataChanged);

// or

OpcSubscribeDataChange[] commands = new OpcSubscribeDataChange[] {
    new OpcSubscribeDataChange(
        "ns=2;s=Machine/IsRunning",
        filter,
        HandleDataChanged),
    new OpcSubscribeDataChange(
        "ns=2;s=Machine/Job/Speed",
        filter,
        HandleDataChanged)
};

OpcSubscription subscriptionB = client.SubscribeNodes(commands);
```

Structured Data

In the following sections **it is assumed** that the server provides a node (via the NodeId “ns=2;s=Machine/Operator”), which makes use of the **(fictitious) data type “StaffType”** , The structure of the data type is defined as follows:

```
StaffType
.Name : string
.ID : long
.Shift : ShiftInfoType
    .Name : string
    .Elapsed : DateTime
    .Remaining : int
```

Simplest Access

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcValue](#) and [OpcDataObject](#).

For easy access to values of variable nodes with structured data, the framework supports the use of the keyword **dynamic**. Accesses to variables that are declared using **dynamic** are evaluated by .NET at runtime. This means that the data of a structured data type can be accessed without prior explicit implementation of a .NET type. Such an access could look like this:

```

client.UseDynamic = true;
client.Connect();

dynamic staff = client.ReadNode("ns=2;s=Machine/Operator").Value;

// Access the 'Name' and 'ID' field of the data without to declare the data type itself.
// Just use the field names known as they would be defined in a .NET Type.
Console.WriteLine("Name: {0}", staff.Name);
Console.WriteLine("Staff ID: {0}", staff.ID);

// Continue accessing subsequently used data types.
Console.WriteLine("Shift: {0}", staff.Shift.Name);
Console.WriteLine("- Time Elapsed: {0}", staff.Shift.Elapsed);
Console.WriteLine("- Jobs Remaining: {0}", staff.Shift.Remaining);

// Change Shift
staff.Name = "John";
staff.ID = 4242;
staff.Shift.Name = "Swing Shift";

client.WriteNode("ns=2;s=Machine/Operator", staff);

```

Name-based Access

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcValue](#), [OpcDataObject](#) and [OpcDataField](#).

For name-based access to values of variable nodes with structured data, the **OpcDataObject** class can be used directly. If the names of the fields of the data type are known, these can be accessed in the following way:

```

client.UseDynamic = true;
client.Connect();

OpcDataObject staff = client.ReadNode("ns=2;s=Machine/Operator").As<OpcDataObject>();

// Access the 'Name' and 'ID' field of the data without to declare the data type itself.
// Just use the field names known as the 'key' to access the according field value.
Console.WriteLine("Name: {0}", staff["Name"].Value);
Console.WriteLine("Staff ID: {0}", staff["ID"].Value);

// Continue accessing subsequently used data types using the OpcDataObject as before.
OpcDataObject shift = (OpcDataObject)staff["Shift"].Value;

Console.WriteLine("Shift: {0}", shift["Name"].Value);
Console.WriteLine("- Time Elapsed: {0}", shift["Elapsed"].Value);
Console.WriteLine("- Jobs Remaining: {0}", shift["Remaining"].Value);

// Change Shift
staff["Name"].Value = "John";
staff["ID"].Value = 4242;
shift["Name"].Value = "Swing Shift";

client.WriteNode("ns=2;s=Machine/Operator", staff);

```

Dynamic Access

The following types are used here: [OpcClient](#), [OpcNodeSet](#), [OpcAutomatism](#) and [OpcDataTypeSystem](#).

For the “simples access” (see [Simplest Access](#)) via **dynamic** (in Visual Basic not typed using **Dim**) as well as the name-based access (see [Name-based Access](#)) the **OpcClient** needs information about the data types provided by the server. This information determines the **OpcClient** class automatically during *Connect()* from the server if the property *UseDynamic* either on the **OpcClient** or the static class **OpcAutomatism** is set to the value **true** before connecting.

By enabling the feature, the **OpcClient** will load the necessary type information from the server so that it can be accessed following the call to *Connect()* without explicit coding or shown in the previous sections.

If a **UANodeSet.xml** or a XML file with the description of the server is available (an XML file whose content starts with *UANodeSet*), this file can also be loaded into the client application. The **OpcClient** can then retrieve the information from the NodeSet and does not need to retrieve the type information when connecting. This could then look like this:

```
using (OpcClient client = new OpcClient("opc.tcp://localhost:4840")) {
    client.NodeSet = OpcNodeSet.Load(@"..\Resources\MyServersNodeSet.xml");
    client.UseDynamic = true;

    client.Connect();
    dynamic staff = client.ReadNode("ns=2;s=Machine/Operator").Value;

    Console.WriteLine("Name: {0}", staff.Name);
    Console.WriteLine("Staff ID: {0}", staff.ID);
}
```

Note that the **OpcClient** class when calling *GetDataTypeSystem()* after setting a NodeSet provides an instance of the **OpcDataTypeSystem** class that describes the type system described in the NodeSet. If, on the other hand, no NodeSet is set using the *NodeSet* property of the **OpcClient** class, calling *GetDataTypeSystem()* returns a **OpcDataTypeSystem** instance representing the server's type system which was determined when calling *Connect ()*.

Typed Access

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcDataTypeAttribute](#) and [OpcDataTypeEncodingAttribute](#).

For typed access, .NET types are defined in the client application as they are defined in the server. All required metadata is provided via attributes. Definition of .NET types for structured data types:

```
[OpcDataType("ns=2;s=StaffType")]
[OpcDataTypeEncoding("ns=2;s=StaffType-Binary")]
public class Staff
{
    public string Name { get; set; }
    public int ID { get; set; }
    public ShiftInfo Shift { get; set; }
}

[OpcDataType("ns=2;s=ShiftInfoType")]
[OpcDataTypeEncoding("ns=2;s=ShiftInfoType-Binary")]
public class ShiftInfo
{
    public string Name { get; set; }
    public DateTime Elapsed { get; set; }
    public byte Remaining { get; set; }
}
```

After the definition of the .NET type for a structured data type (on the client-side) defined by the server, this can be used as follows:

```
client.Connect();
Staff staff = client.ReadNode("ns=2;s=Machine/Operator").As<Staff>();

// Access the 'Name' and 'ID' field of the data with the declared the data type.
Console.WriteLine("Name: {0}", staff.Name);
Console.WriteLine("Staff ID: {0}", staff.ID);

// Continue accessing subsequently used data types.
Console.WriteLine("Shift: {0}", staff.Shift.Name);
Console.WriteLine("- Time Elapsed: {0}", staff.Shift.Elapsed);
Console.WriteLine("- Jobs Remaining: {0}", staff.Shift.Remaining);

// Change Shift
staff.Name = "John";
staff.ID = 4242;
staff.Shift.Name = "Swing Shift";

client.WriteNode("ns=2;s=Machine/Operator", staff);
```

Generate Data Types

If the typed access (see [Typed Access](#)) is to be used, there is the option to generate either only certain or all data types of an OPC UA server via the [OPC Watch](#).

Generation using a Server

To generate a single data type implemented in .NET, perform the following steps:

1. Open **OPC Watch**
2. Create a new connection (+) and configure it (if necessary)
3. Connect to the server (plug icon)
4. Select a node...
 1. either a variable-node with a structured data type as its value
 2. or a DataType-node under /Types/DataTypes/BaseDataType/Structure or /Enumeration
5. Right click on this node

6. Click "Generate DataType"
7. Paste the code into the application, done!

All data types can be generated either in a code file (*.cs) or assembly file (*.dll). To do this, follow these steps:

1. Open **OPC Watch**
2. Create a new connection (+) and configure it (if necessary)
3. Connect to the server (plug icon)
4. Select the Server-node "opc.tcp://..."
5. Right click on this node
6. Click "Generate Models"
7. Select the desired file type in the dialog
8. Click "Save"
9. Add the file to the project, done!

Generation using a NodeSet

1. Open **OPC Watch**
2. Click the first icon in the upper right corner of the application
3. Open the NodeSet file (usually a XML file of a Companion Specification)
4. NodeSet is loaded and then a "Save as ..." dialog is displayed
5. Select the desired file type in the dialog
6. Click "Save"
7. Add the file to the project, done!

Define Data Types

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcData](#), [OpcDataTypeAttribute](#), [OpcDataTypeEncodingAttribute](#), [OpcDataTypeSystem](#) and [OpcDataTypeInfo](#).

As an alternative to generating the data type as a .NET code or .NET assembly (see [Generate Data Types](#)), these can also be defined manually. To do this, implement the type as defined by the server. This means that the .NET Type (regardless of structure or class) must provide the fields of the structured data type - in terms of their type - in exactly the same order. All other metadata is provided via corresponding attributes:

```
[OpcDataType("<NodeId of DataType Node>")]
[OpcDataTypeEncoding(
    "<NodeId of Binary Encoding Node>",
    NamespaceUri = "<NamespaceUri.Value of binary Dictionary-Node>")]
internal struct MyDataType
{
    public short FieldA;
    public int FieldB;
    public string FieldC;
    ...
}
```

The information required for the definition can be obtained either via the OPC UA Server manual, the responsible PLC developer or via the **OpcClient** class. To get the necessary information via the **OpcClient** class, you can examine the variable node - which uses the structured data types - as follows:

```
OpcNodeInfo node = client.BrowseNode("ns=2;s=Machine/Operator");

if (node is OpcVariableNodeInfo variableNode) {
    OpcNodeId dataTypeId = variableNode.DataTypeId;
    OpcDataTypeInfo dataType = client.GetDataTypeInfo().GetType(dataTypeId);

    Console.WriteLine(dataType.TypeId);
    Console.WriteLine(dataType.Encoding);

    Console.WriteLine(dataType.Name);

    foreach (OpcDataFieldInfo field in dataType.GetFields())
        Console.WriteLine("{0} : {1}", field.Name, field.FieldType);

    Console.WriteLine();
    Console.WriteLine("Data Type Attributes:");
    Console.WriteLine(
        "\t[OpcDataType(\"{0}\")] ",
        dataType.TypeId.ToString(OpcNodeIdFormat.Foundation));
    Console.WriteLine(
        "\t[OpcDataTypeEncoding(\"{0}\", NamespaceUri = \"{1}\")] ",
        dataType.Encoding.Id.ToString(OpcNodeIdFormat.Foundation),
        dataType.Encoding.Namespace.Value);
}
```

Data Types with optional Fields

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcData](#), [OpcDataTypeAttribute](#), [OpcDataTypeEncodingAttribute](#), [OpcDataTypeEncodingMaskAttribute](#) and [OpcDataTypeMemberSwitchAttribute](#).

In order to reduce the amount of data transported, it is possible to “mark” certain fields of a structured data type as existent or missing in the data stream depending on certain conditions or on the value of another field within the data structure. Thus, to “mark” that a field is available, either the value of another field or a single bit within the EncodingMask (a field coded as a preamble before the data in the stream) is used. The size of the EncodingMask is specified in number of bytes in the 'OpcDataTypeEncodingMaskAttribute'; if the 'Size' property is not explicitly set, its value (if 'OpcEncodingMaskKind.Auto' is used) will be the smallest number of bytes needed (based on the number of optional fields).

The following options are available for defining the optional fields:

```
[OpcDataType("<NodeId of DataType Node>")]
[OpcDataTypeEncoding(
    "<NodeId of Binary Encoding Node>",
    NamespaceUri = "<NamespaceUri.Value of binary Dictionary-Node>")]
[OpcDataTypeEncodingMask(OpcEncodingMaskKind.Auto)]
internal struct MyDataTypeWithOptionalFields
{
    public short FieldA;
    public int FieldB;
    public string FieldC;

    // Nullables are treat as optional fields by default.
    // Existence-Indicator-Bit is Bit0 in the encoding mask.
    public uint? OptionalField1;

    // Existence-Indicator-Bit is Bit1 (the next unused bit) in the encoding mask.
    [OpcDataTypeMemberSwitch]
    public int OptionalField2;

    // Existence-Indicator-Bit is Bit3 (bit 2 is unused) in the encoding mask.
    [OpcDataTypeMemberSwitch(bit: 3)]
    public byte OptionalField3;

    public bool FieldD;

    // 'OptionalField4' exists only if the value of 'FieldD' is equals 'true'.
    [OpcDataTypeMemberSwitch("FieldD")]
    public string OptionalField4;

    public int FieldE;

    // 'OptionalField5' exists only if the value of 'FieldE' is greater than '42'.
    [OpcDataTypeMemberSwitch("FieldE", value: 42, @operator:
OpcMemberSwitchOperator.GreaterThan)]
    public string OptionalField5;
}
```

Historical Data

The following types are used here: [OpcClient](#), [OpcNodeId](#), [IOpcNodeHistoryNavigator](#), [OpcHistoryValue](#) and [OpcAggregateType](#).

According to OPC UA specification every Node of the category **Variable** supports the historical recording of the values from its *Value Attribute*. Hereby the new value is saved together with the time stamp of the *Value Attribute* at every change of value of the *Value Attribute*. These **pairs consisting of value and timestamp** are called **historical data**. The Server itself decides on where to save the data. However, the Client can detect via the *IsHistorizing Attribute* of the Node, if the Server provides historical data for a Node and / or historically saves value changes. The **OpcNodeId** of the Node determines from which Node the historical data shall be accessed. Hereby the Client can read, update, replace, delete and create historical data. Mostly, historical data is read by the Client. Processing all historical values, independent from reading with or without navigator, is not necessary.

In order to read historical data the Client can:

- read all values within an **open time frame** (= non-defined StartTime or EndTime)

- read all values **from a particular timestamp forward** (= StartTime):

```
var startTime = new DateTime(2017, 2, 16, 10, , );
var history = client.ReadNodeHistory(
    startTime, null, "ns=2;s=Machine/Job/Speed");
```

- read all values **up until a particular timestamp** (= EndTime):

```
var endTime = new DateTime(2017, 2, 16, 15, , );
var history = client.ReadNodeHistory(
    null, endTime, "ns=2;s=Machine/Job/Speed");
```

- read all values within a **closed time window** (= defined StartTime and EndTime):

```
var startTime = new DateTime(2017, 2, 16, 10, , );
var endTime = new DateTime(2017, 2, 16, 15, , );

var history = client.ReadNodeHistory(
    startTime, endTime, "ns=2;s=Machine/Job/Speed");
```

- The values are being processed via an instance that implements the **IEnumerable** interface:

```
foreach (var value in history) {
    Console.WriteLine(
        "{0}: {1}",
        value.Timestamp,
        value);
}
```

In order to read historical data pagewise (= only a particular number of values is retrieved from the Server) the Client can:

- read a **particular number of values** per page:

```
var historyNavigator = client.ReadNodeHistory(
    10, "ns=2;s=Machine/Job/Speed");
```

- read a **particular number of values** per page within an **open time frame** (= undefined StartTime or EndTime)

- read a **particular number of values** per page **from a particular timestamp forward** (= StartTime):

```
var startTime = new DateTime(2017, 2, 16, 15, , );
var historyNavigator = client.ReadNodeHistory(
    startTime, 10, "ns=2;s=Machine/Job/Speed");
```

- read a **particular number of values** per page **up until a particular timestamp** (= EndTime):

```
var endTime = new DateTime(2017, 2, 16, 15, , );
var historyNavigator = client.ReadNodeHistory(
    null, endTime, 10, "ns=2;s=Machine/Job/Speed");
```

- read a **particular number of values** per page **within a closed time frame** (= defined StartTime and EndTime):

```
var startTime = new DateTime(2017, 2, 16, 10, , );
var endTime = new DateTime(2017, 2, 16, 15, , );

var historyNavigator = client.ReadNodeHistory(
    startTime, endTime, 10, "ns=2;s=Machine/Job/Speed");
```

- The values are then processed via an instance that implements the **IOpcNodeHistoryNavigator** interface:

```
do {
    foreach (var value in historyNavigator) {
        Console.WriteLine(
            "{0}: {1}",
            value.Timestamp,
            value);
    }
} while (historyNavigator.MoveNextPage());

historyNavigator.Close();
```

- Always ensure that the *Close* method of the **IOpcNodeHistoryNavigator** instance is called. This is necessary in order for the Server to be able to dispose of the historical data buffered for the request afterwards. As an alternative to the explicit call of the *Close* method the navigator can also be used in a *using* block:

```
using (historyNavigator) {
    do {
        foreach (var value in historyNavigator) {
            Console.WriteLine(
                "{0}: {1}",
                value.Timestamp,
                value);
        }
    } while (historyNavigator.MoveNextPage());
}
```

Different types of aggregation can be chosen for processed reading of the historical data via the **OpcAggregateType**:

- For reading the **lowest value** within a time frame:

```
var minSpeed = client.ReadNodeHistoryProcessed(
    startTime,
    endTime,
    OpcAggregateType.Minimum,
    "ns=2;s=Machine/Job/Speed");
```

- For reading the **average value** within a time frame:

```
var avgSpeed = client.ReadNodeHistoryProcessed(
    startTime,
    endTime,
    OpcAggregateType.Average,
    "ns=2;s=Machine/Job/Speed");
```

- For reading the **highest value** within a time frame:

```
var maxSpeed = client.ReadNodeHistoryProcessed(
    startTime,
    endTime,
    OpcAggregateType.Maximum,
    "ns=2;s=Machine/Job/Speed");
```

Nodes

Method Nodes

The following types are used here: [OpcClient](#), [OpcNodeId](#) and [OpcCallMethod](#).

The **OpcNodeId** of the Node determines which method Node is to be called. The hereby expected parameters of a method can be provided via the *parameters* when calling **CallMethod** or by the according **OpcCallMethod** instance. Note that first the **OpcNodeId** of the *owner* of the method has to be given and then the **OpcNodeId** of the method itself. The **OpcNodeId** of the *owner* determines the identifier of the object Node or the object type Node, that references the method as a *HasComponent reference*.

- Call a single method Node without parameters (the method does not define any IN arguments):

```
// The result array contains the values of the OUT arguments offered by the method.
object[] result = client.CallMethod(
    "ns=2;s=Machine",                /* NodeId of Owner Node */
    "ns=2;s=Machine/StartMachine"    /* NodeId of Method Node */);
```

- Call a single method Node with parameters (the method defines some IN arguments):

```
// The result array contains the values of the OUT arguments offered by the method.
object[] result = client.CallMethod(
    "ns=2;s=Machine",                /* NodeId of Owner Node */
    "ns=2;s=Machine/StopMachine",    /* NodeId of Method Node */
    "Job Change",                    /* Parameter 1: 'reason' */
    10023,                           /* Parameter 2: 'reasonCode' */
    DateTime.Now                      /* Parameter 3: 'scheduleDate' */);
```

- Call several method Nodes without parameters (the methods do not define any IN arguments):

```
OpcCallMethod[] commands = new OpcCallMethod[] {
    new OpcCallMethod("ns=2;s=Machine", "ns=2;s=Machine/StopMachine"),
    new OpcCallMethod("ns=2;s=Machine", "ns=2;s=Machine/ScheduleJob"),
    new OpcCallMethod("ns=2;s=Machine", "ns=2;s=Machine/StartMachine")
};

// The result array contains the values of the OUT arguments offered by the methods.
object[][] results = client.CallMethods(commands);
```

- Call several method Nodes with parameters (the methods do define some IN arguments):

```

OpcCallMethod[] commands = new OpcCallMethod[] {
    new OpcCallMethod(
        "ns=2;s=Machine",           /* NodeId of Owner Node */
        "ns=2;s=Machine/StopMachine", /* NodeId of Method Node */
        "Job Change",               /* Parameter 1: 'reason' */
        10023,                      /* Parameter 2: 'reasonCode' */
        DateTime.Now                /* Parameter 3: 'scheduleDate' */),
    new OpcCallMethod(
        "ns=2;s=Machine",           /* NodeId of Owner Node */
        "ns=2;s=Machine/ScheduleJob", /* NodeId of Method Node */
        "MAN_F01_78910"            /* Parameter 1: 'jobSerial' */),
    new OpcCallMethod(
        "ns=2;s=Machine",           /* NodeId of Owner Node */
        "ns=2;s=Machine/StartMachine", /* NodeId of Method Node */
        10021                      /* Parameter 1: 'reasonCode' */),
};

// The result array contains the values of the OUT arguments offered by the methods.
object[][] results = client.CallMethods(commands);

```

File Nodes

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcFile](#), [OpcFileMode](#), [OpcFileStream](#), [OpcFileInfo](#), [OpcFileMethods](#) and [SafeOpcFileHandle](#).

Nodes of the type **FileType** define per definition of the OPC UA specification certain properties (= Property Nodes) and methods (= Method Nodes) that allow access to a data stream as if your were accessing a file in the file system. Hereby information about the content of the logical and physical file is provide exclusively. According to the specification, a possibly existing path to the file is not provided. The access to the file itself is realized via Open, Close, Read, Write, GetPosition and SetPosition. The data is always processed in a binary way. Like on every other platform in the OPC UA you can choose a mode while opening Open which sets the kind of data access planned. You can also request exclusive access to a file in the OPC UA. After calling the Open method you recieve a numeric key for further file handle. This key always has to be handed over at the methods Read, Write, GetPosition and SetPosition. An open file has to be closed again when no longer needed.

Access to Nodes of the type FileType can be executed manually via the OpcClient by using the ReadNode and CallMethod functions. As an alternative the Framework provides numerous other classes that - modelled after the .NET Framework - allow access to Nodes of the type FileType. The **OpcNodeId** of the Node determines which "File Node" shall be accessed.

Data access with the **OpcFile** class:

- Reading the entire content of a text file:

```
string reportText = OpcFile.ReadAllText(client, "ns=2;s=Machine/Report");
```

- Appending further text data to a text file:

```
OpcFile.AppendAllText(client, "ns=2;s=Machine/Report", "Lorem ipsum");
```

- Opening and reading the file via **OpcFileStream**:

```
using (var stream = OpcFile.OpenRead(client, "ns=2;s=Machine/Report")) {
    var reader = new StreamReader(stream);

    while (!reader.EndOfStream)
        Console.WriteLine(reader.ReadLine());
}
```

- Opening and writing the file via **OpcFileStream**:

```
using (var stream = OpcFile.OpenWrite(client, "ns=2;s=Machine/Report")) {
    var writer = new StreamWriter(stream);

    writer.WriteLine("Lorem ipsum");
    writer.WriteLine("dolor sit");
    // ...
}
```

Data access with the **OpcFileInfo** class:

- Creating an **OpcFileInfo** instance:

```
var file = new OpcFileInfo(client, "ns=2;s=Machine/Report");
```

- Working with the **OpcFileInfo** instance:

```
if (file.Exists) {
    Console.WriteLine($"File Length: {file.Length}");

    if (file.CanUserWrite) {
        using (var stream = file.OpenWrite()) {
            // Your code to write via stream.
        }
    }
    else {
        using (var stream = file.OpenRead()) {
            // Your code to read via stream.
        }
    }
}
```

Data access with the **OpcFileMethods** class:

- via .NET SafeHandle concept (realized via the **SafeOpcFileHandle** class):

```
using (var handle = OpcFileMethods.SecureOpen(client, "ns=2;s=Machine/Report",
OpcFileMode.ReadWrite)) {
    byte[] data = OpcFileMethods.SecureRead(handle, 100);

    long position = OpcFileMethods.SecureGetPosition(handle);
    OpcFileMethods.SecureSetPosition(handle, position + data.Length - 1);

    OpcFileMethods.SecureWrite(handle, new byte[] { 1, 2, 3 });
}
```

- via numeric File Handle:

```
uint handle = OpcFileMethods.Open(client, "ns=2;s=Machine/Report",
OpcFileMode.ReadWrite);

try {
    byte[] data = OpcFileMethods.Read(client, "ns=2;s=Machine/Report", handle, 100);

    ulong position = OpcFileMethods.GetPosition(client, "ns=2;s=Machine/Report",
handle);
    OpcFileMethods.SetPosition(client, "ns=2;s=Machine/Report", handle, position +
data[data.Length - 1]);

    OpcFileMethods.Write(client, "ns=2;s=Machine/Report", handle, new byte[] { 1, 2, 3
});
}
finally {
    OpcFileMethods.Close(client, "ns=2;s=Machine/Report", handle);
}
```

Only file accesses via **OpcFile**, **OpcFileInfo**, **OpcFileStream** and **SafeOpcFileHandle** guarantee an also implicit release of an open file, even if the call of the *Close* method has been “forgotten”. When closing the connection to the Server, at the latest, all open files are closed by the **OpcClient** automatically. However, this is not the case, if the methods of the class **OpcFileMethods** are used without the “Secure” prefix.

The OPC UA specification does not define a way to determine a Node as a Node of the type FileType and therefore as a File-Node. For this the Framework offers the option to identify a File-Node via its Node structure:

```
if (OpcFileMethods.IsFileNode(client, "ns=2;s=Machine/Report")) {
    // Your code to operate on the file node.
}
```

Datatype Nodes

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcNodeInfo](#) and [OpcTypeNodeInfo](#).

In case there a Server provides a Node, through that the Server publishes information about a Server defined Datatype, it can be necessary that a Client wants to query such type information. The simplest way to do so is Browsing. The framework provides especially for Datatype Nodes a specialisation of the **OpcNodeInfo** - the **OpcTypeNodeInfo**. Using the properties of this class additional information can be queried about the user defined Datatype. For example whether the Datatype is an enumeration. In this case it is possible to query the different enum-entries as well. This works as follows:

```
var machineStatusNode = client.BrowseNode("ns=2;s=MachineStatus") as OpcTypeNodeInfo;

if (machineStatusNode != null && machineStatusNode.IsEnum) {
    var members = machineStatusNode.GetEnumMembers();

    foreach (var member in members)
        Console.WriteLine(member.Name);
}
```

Data Nodes

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcNodeInfo](#) and [OpcVariableNodeInfo](#).

Working with Data Nodes is primarily restricted to Reads and Writes regarding the *Value*-Attribut of the Node. This works like on every other Node as already explained in “Reading Values of Node(s)” and “Writing Values of Node(s)”. In case there additional information is required except of the value of the Node (without the need to manually retrieve them using *ReadNode* requests), then they can be directly and simple requested from the Server using Browsing. The following sample represents how this does work:

```
var machineStatusNode = client.BrowseNode("ns=2;s=Machine/Status") as OpcVariableNodeInfo;

if (machineStatusNode != null) {
    Console.WriteLine($"AccessLevel: {machineStatusNode.AccessLevel}");
    Console.WriteLine($"UserAccessLevel: {machineStatusNode.UserAccessLevel}");
    ...
}
```

Furthermore it is possible to continue the Browsing on the Datatype Node used by the Data Node:

```
var dataTypeNode = machineStatusNode.DataType;

if (dataTypeNode.IsSystemType) {
    ...
}
```

Data-Item Nodes

The following types are used here: [OpcClient](#) und [OpcNodeId](#).

Data-Item Nodes, provided by the **OpcDataItemNode** through a Server, are an extension of the simple **OpcDataVariableNode**. They are also primarily used for value provision. Beyond from that they provide additional useful Metadata. One of that additional information is published through the *Definition* property which is defined as Property-Node which is in turn a child of the Data-Item Node. This property is used to ensure the correct processing and interpretation of the data provided by the Node. The value of the property is vendor specific and shall inform the user how the value is achieved. Apart from that it is also possible to work with the Node as represented in “Reading Values of Node(s)” and “Writing Values of Node(s)”.

Data-Item Nodes for analog Values

The following types are used here: [OpcClient](#), [OpcNodeId](#), [OpcNodeInfo](#) and [OpcAnalogItemNodeInfo](#).

This kind of Node represents a specialisation of the Data-Item Node **OpcDataItemNode**. The values provided are especially used in analog environments and can be described through the additional properties *InstrumentRange*, *EngineeringUnit* and *EngineeringUnitRange*. While the *InstrumentRange* describes the value range of the analog data of its source, the *EngineeringUnit* defines the measure of the provided values. Is the value determined under normal conditions, then it is within the customizable value range defined by the *EngineeringUnitRange* property. The *EngineeringUnit* property described measure is defined by the UNECE Recommendation N° 20. These recommendations are based on the International System of Units (short SI Units). The **OpcAnalogItemNodeInfo** provides an appropriate interface for

simplified processing of the provided information of such a Node through Browsing:

```
var temperatureNode = client.BrowseNode("ns=2;s=Machine/Temperature") as
OpcAnalogItemNodeInfo;

if (temperatureNode != null) {
    Console.WriteLine($"InstrumentRange: {temperatureNode.InstrumentRange}");

    Console.WriteLine($"EngineeringUnit: {temperatureNode.EngineeringUnit}");
    Console.WriteLine($"EngineeringUnitRange: {temperatureNode.EngineeringUnitRange}");
}
```

The information contained in an **OpcEngineeringUnitInfo** describes a unit of measurement of the UNECE table for units of measurement. The possible units can be looked up at the OPC Foundation: [UNECE units of measurement in OPC UA \(currently not available\)](#). Alternatively, the MTConnect Institute also offers the UNECE table on GitHub: [UNECE Units of Measure in OPC UA \(MTConnect@GitHub\)](#).

Events

Event Subscription

The following types are used: [OpcClient](#), [OpcObjectTypes](#), [OpcEventReceivedEventArgs](#), [OpcEvent](#), [OpcAlarmCondition](#), [OpcSimpleAttributeOperand](#), [OpcFilter](#), [OpcEventFilter](#) und [OpcEventSeverity](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

Events inform a subscriber (such as Clients) about operations, conditions, and system-specific events. Such information can be delivered to interested parties directly via **global events**. A global event is always published through the Server's "Server" Node. For this reason, global events are also subscribed via the Server Node as follows:

```
client.SubscribeEvent(OpcObjectTypes.Server, HandleGlobalEvents);
```

An event is generally handled with a method of the following signature:

```
private static void HandleGlobalEvents(
    object sender,
    OpcEventReceivedEventArgs e)
{
    Console.WriteLine(e.Event.Message);
}
```

The event data received via the **OpcEventReceivedEventArgs** instance can be retrieved via the *Event* property. The property always returns an instance of the type **OpcEvent**. If the received event data is a specialization of the **OpcEvent** class, it can be simply cast as follows:

```
private static void HandleGlobalEvents( _
    object sender, _
    OpcEventReceivedEventArgs e)
{
    var alarm = e.Event as OpcAlarmCondition;

    if (alarm != null)
        Console.WriteLine("Alarm: " + alarm.Message);
}
```

The call shown above subscribes to all events published globally by the Server. In order to restrict the information received as well as the events which are ever sent from the Server to the Client, an event filter can be defined and transmitted to the Server when a subscription is concluded:

```
// Define an attribute operand using the identifier of the type which defines the
// attribute / property including the name of the attribute / property to evaluate
// by the operand.
var severity = new OpcSimpleAttributeOperand(OpcEventTypes.Event, "Severity");
var conditionName = new OpcSimpleAttributeOperand(OpcEventTypes.Condition, "ConditionName");

var filter = OpcFilter.Using(client)
    .FromEvents(OpcEventTypes.AlarmCondition)
    .Where(severity > OpcEventSeverity.Medium & conditionName.Like("Temperature"))
    .Select();

client.SubscribeEvent(
    OpcObjectTypes.Server,
    filter,
    HandleGlobalEvents);
```

Creating an event filter always requires a **OpcClient** instance, which must already have a connection to the destination Server. This is required because the Client collects the type information from the Server relevant for the assembling of the filter when assembling. In the above example, all properties of the Node type **OpcAlarmConditionNode** are recursively collected up to the **OpcNode** via the Client, and according to the *Where* and *Select* clauses rules for the selection of the Event data are created. The Node types to be analyzed can be passed with comma separated to the *FromEvents(...)*- method:

```
var severity = new OpcSimpleAttributeOperand(OpcEventTypes.Event, "Severity");
var conditionName = new OpcSimpleAttributeOperand(OpcEventTypes.Condition, "ConditionName");

var filter = OpcFilter.Using(client)
    .FromEvents(
        OpcEventTypes.AlarmCondition,
        OpcEventTypes.ExclusiveLimitAlarm,
        OpcEventTypes.DialogCondition)
    .Where(severity > OpcEventSeverity.Medium & conditionName.Like("Temperature"))
    .Select();

client.SubscribeEvent(
    OpcObjectTypes.Server,
    filter,
    HandleGlobalEvents);
```

The *Where(...)* method can then be used to restrict the information collected by *FromEvents(...)* about the properties provided by the Node types. For this, the framework offers various operator overloads ($\leq$, $<$, $>$, $\geq$ and $=$). For logical combinations of the operands it also offers the logical operators OR ($|$) and AND

(&). In addition, various methods are available for further restrictions, such as: *Like*, *Between*, *InList*, *IsNull*, *Not* and *OfType*.

After the restriction has been made, you can use the *Select(...)* method to select the properties that will be additionally selected by the Server when the event occurs (which corresponds to the conditions specified under *Where(...)*) and are transferred to the Client.

Event Nodes

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcEvent](#) und [OpcSubscription](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

It is not always appropriate for a Server to send events globally through the Server Node to all subscribers. Often the context therefore plays a crucial role in whether an event is of interest to a subscriber. Event Nodes are used to define local events. If the Server uses an Event Node to provide event data for a Node, then the Event Node is a so-called “Notifier” of the Node. For this reason, it is also possible to recognize by a *HasNotifier* reference which event Node reports event data to a Node. It should be noted that an Event Node may be a “Notifier” for multiple Nodes. As a result, local events are always subscribed via the notified Nodes:

```
client.SubscribeEvent(machineNodeId, HandleLocalEvents);
```

An event is generally handled with a method of the following signature. The further processing of the event data is identical to the processing of global events (see section 'Working with Events').

```
private static void HandleLocalEvents(
    object sender,
    OpcEventReceivedEventArgs e)
{
    Console.WriteLine(e.Event.Message);
}
```

Of course, local events, as shown in the section 'Working with Events', can also be filtered.

Generally, after a subscriber (a Client) is only informed of events as long as he is in contact with the Server and has initiated a subscription, a subscriber will not know what events have already occurred prior to establishing a connection to the Server. If the Server is to inform subscribers of past events, subscribers can request them from the Server as follows. **In general, however, a Server is not obliged to provide past events.**

```
var subscription = client.SubscribeEvent(
    machineNodeId,
    HandleLocalEvents);

// Query most recent event information.
subscription.RefreshConditions();
```

Event Nodes with Conditions

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcEvent](#), [OpcCondition](#) und [OpcEventSeverity](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

A specialization of the **OpcEvent** (featured in the 'Working with Events' section) is the **OpcCondition** class. It serves to provide event data to which specific conditions are attached. Only in the case that a condition awarded to an Event Node is true will such an event be triggered. The information of the event includes information about the state of the condition as well as information related to the evaluation of the condition. Since this information can vary in complexity depending on the scenario, the **OpcCondition** represents the base class of all event data to which a condition is attached.

In addition to the general properties of an event (provided by the **OpcEvent** base class), one instance of the **OpcCondition** provides information that may be of interest to the Client for further processing of the event. This allows the Client to verify that, in general, the Server evaluates the event as relevant to a Client (see *IsRetained* property). Similarly, the Client can evaluate the state of the condition, that is, whether it is active or inactive (see *IsEnabled* property).

```
private static void HandleLocalEvents(
    object sender,
    OpcEventReceivedEventArgs e)
{
    var condition = e.Event as OpcCondition;

    if (condition.IsRetained) {
        Console.Write((condition.ClientUserId ?? "Comment") + ":");
        Console.WriteLine(condition.Comment);
    }
}
```

If the client does not want to further evaluate the condition, the client can deactivate it:

```
private static void HandleLocalEvents(object sender, OpcEventReceivedEventArgs e)
{
    var condition = e.Event as OpcCondition;

    if (condition.IsEnabled && condition.Severity < OpcEventSeverity.Medium)
        condition.Disable(client);
}
```

In addition, the client can also add a comment to the current state of the condition:

```
private static void HandleLocalEvents(object sender, OpcEventReceivedEventArgs e)
{
    var condition = e.Event as OpcCondition;

    if (condition != null)
        condition.AddComment(client, "Evaluated by me!");
}
```

Event Nodes with Dialog Conditions

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcCondition](#), [OpcDialogCondition](#), [OpcEventSeverity](#) und [OpcDialogRequestedEventArgs](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

A specialization of the **OpcCondition** is the **OpcDialogCondition**. The condition associated with this

event is a dialog with the subscribers. In this case, such a condition consists of a prompt, response options and information as to which option should be selected by default (*DefaultResponse* property), which option to confirm the dialog (*OkResponse* property) and which is used to cancel the dialog (*CancelResponse* property). When such a dialog-driven event is triggered, the Server waits for one of the subscribers to provide it with an answer in the form of the choice made based on the given answer options. The condition for further processing, the operations linked to the dialog, is thus the answer to a task, a question, an information or a warning. The information provided for this by the event can be processed as follows and reported back to the Server:

```
private static void HandleLocalEvents(object sender, OpcEventReceivedEventArgs e)
{
    var condition = e.Event as OpcDialogCondition;

    if (condition != null && condition.IsActive) {
        Console.WriteLine(condition.Prompt);
        Console.WriteLine("    Options:");

        var responseOptions = condition.ResponseOptions;

        for (int index = ; index < responseOptions.Length; index++) {
            Console.Write($"        [{index}] = {responseOptions[index].Value}");

            if (index == condition.DefaultResponse)
                Console.Write(" (default)");

            Console.WriteLine();
        }

        var respond = string.Empty;
        var respondOption = condition.DefaultResponse;

        do {
            Console.Write("Enter the number of the option and press Enter to respond: ");
            respond = Console.ReadLine();

            if (string.IsNullOrEmpty(respond))
                break;
        } while (!int.TryParse(respond, out respondOption));

        condition.Respond(client, respondOption);
    }
}
```

Apart from the standard event handling method, the **OpcClient** class also provides the *DialogRequested* event. If **OpcDialogCondition** event data is received by the Client and not answered by any handler, the Client executes the *DialogRequested* event to perform a dialog processing on it. This also simplifies the handling of the event data somewhat, since these can be passed on typed to the event handler:

```
client.DialogRequested += HandleDialogRequested;
...

private static void HandleDialogRequested(
    object sender,
    OpcDialogRequestedEventArgs e)
{
    // Just use the default response, here.
    e.SelectedResponse = e.Dialog.DefaultResponse;
}
```

You just need to set the *SelectedResponse* property of the event arguments. Calling the *Respond(...)* method of the **OpcDialogCondition** is done by the **OpcClient** after the execution of the event handler.

Event Nodes with Feedback Conditions

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcEvent](#), [OpcCondition](#), [OpcAcknowledgeableCondition](#) und [OpcEventSeverity](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

Based on **OpcCondition** events, the **OpcAcknowledgeableCondition** is a specialization used as the base class for conditions with feedback requirements. Events of this type define that, when their condition is met, a “report with acknowledgment of receipt” is issued. The “return receipt” - that is the feedback - can be used to control further processes as well as to easily acknowledge hints and warnings. The feedback mechanism provided for this purpose is divided into two stages. While the first stage is a kind of “read receipt”, the second level is a kind of “read receipt with a nod”. OPC UA defines the read receipt as a simple confirmation and the read receipt as a nod with acknowledgment. For both types of recognition, the events provide corresponding *Confirm* and *Acknowledge* methods. By definition, the execution of the “acknowledge” process should make an explicit execution of the “confirm” process unnecessary. On the other hand, it is possible to first send a confirmation and then, separately, an acknowledgment. Regardless of the order and the type of feedback, a comment from the operator can optionally be specified for the confirm or acknowledge. An acknowledgment as feedback could be implemented as follows:

```
private static void HandleLocalEvents(object sender, OpcEventReceivedEventArgs e)
{
    var condition = e.Event as OpcAcknowledgeableCondition;

    if (condition != null && !condition.IsAcked) {
        Console.WriteLine($"Acknowledgment is required for condition:
{condition.ConditionName}");
        Console.WriteLine($" -> {condition.Message}");
        Console.Write("Enter your acknowlegment comment and press Enter to acknowledge: ");

        var comment = Console.ReadLine();
        condition.Acknowledge(client, comment);
    }
}
```

In addition, when processing this type of event, you can check whether the event has already been confirmed by *Confirm* (see *IsConfirmed* property) or by *Acknowledge* (see *IsAcked* property). **It should be noted that a Server must always define the interpretation itself as well as the logic following the respective feedback.** So whether a Server makes use of both feedback options or only one is left to the respective developer. In the best case, a Server uses at least the *Acknowledge* method, as it is defined

by the specification as “stronger”.

Event Nodes with Alarm Conditions

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcAlarmCondition](#), [OpcAcknowledgeableCondition](#) und [OpcEventSeverity](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

The most important implementation of the **OpcAcknowledgeableCondition** in OPC UA is the **OpcAlarmCondition**. With the help of **OpcAlarmCondition** events it is possible to define events whose behavior is comparable to a bedside timer. Accordingly, such an event becomes active (see *IsActive* property) if the condition associated with it is met. In the case of an alarm clock, for example, “reaching the alarm time”. For example, an alarm that is set with a wake-up time but should not be activated when it is reached is called a suppressed alarm (see *IsSuppressed* and *IsSuppressedOrShelved* property). But if an alarm becomes active, it can be shelved (see *IsSuppressedOrShelved* property). An alarm can be reset once (“One Shot Shelving”) or in time (“Timed Shelving”) (see *Shelving* Child Node). Alternatively, a reset alarm can also be “unshelved” again (see *Shelving* Child Node).

```
private static void HandleLocalEvents(  
    object sender,  
    OpcEventReceivedEventArgs e)  
{  
    var alarm = e.Event as OpcAlarmCondition;  
  
    if (alarm != null) {  
        Console.WriteLine($"Alarm {alarm.ConditionName} is");  
        Console.WriteLine($"{{(alarm.IsActive ? "active" : "inactive")}}!");  
    }  
}
```

Event Nodes with discrete Alarm Conditions

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcDiscreteAlarm](#), [OpcOffNormalAlarm](#) und [OpcTripAlarm](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

Based on the **OpcAlarmCondition** event data, there are several specializations that have been explicitly defined for specific types of alarms to further specify the form, reason or content of an alarm by the nature of the alarm. A subclass of such self-describing alarms are the discrete alarms. The basis for a discrete alarm is the **OpcDiscreteAlarm** class. It defines an alarm state that is used to classify types into alarm states, where the input for the alarm can only accept a certain number of possible values (e.g. true / false, running / paused / terminated). If an alarm represents a discrete condition that is considered abnormal, the **OpcOffNormalAlarm** or a subclass of it will be used. Starting from this alarm class, the framework offers a further concretization with the **OpcTripAlarm**. The **OpcTripAlarm** becomes active when, for example, an abnormal fault occurs on a monitored device, e.g. when the motor is shut down due to overload.

```
private static void HandleLocalEvents(object sender, OpcEventReceivedEventArgs e)
{
    var alarm = e.Event as OpcDiscreteAlarm;

    if (alarm != null) {
        if (alarm is OpcTripAlarm)
            Console.WriteLine("Trip Alarm!");
        else if (alarm is OpcOffNormalAlarm)
            Console.WriteLine("Off Normal Alarm!");
    }
}
```

Event Nodes with Alarm Conditions for Limits

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#) und [OpcLimitAlarm](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

If the Server checks process-specific limit values and then publishes the output of the check for limit value overruns / underruns, then the **OpcLimitAlarm** class represents the central starting point for entering the classes of limit alarms. Using this class limits are divided into up to four levels. To differentiate them, they are called LowLow, Low, High and HighHigh (called in order of their metric order). By definition, the Server does not need to define all limits.

```
private static void HandleLocalEvents(
    object sender,
    OpcEventReceivedEventArgs e)
{
    var alarm = e.Event as OpcLimitAlarm;

    if (alarm != null) {
        Console.Write(alarm.LowLowLimit);
        Console.Write(" ≤ ");
        Console.Write(alarm.LowLimit);
        Console.Write(" ≤ ");
        Console.Write(alarm.HighLimit);
        Console.Write(" ≤ ");
        Console.Write(alarm.HighHighLimit);
    }
}
```

Event Nodes with Alarm Conditions for exclusive Limits

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcLimitAlarm](#), [OpcExclusiveLimitAlarm](#), [OpcExclusiveDeviationAlarm](#), [OpcExclusiveLevelAlarm](#) und [OpcExclusiveRateOfChangeAlarm](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

A subclass of the **OpcLimitAlarm** events is the class **OpcExclusiveLimitAlarm**. As its name suggests, it serves to define limit alerts for exclusive boundaries. Such a limit alarm uses values for the boundaries that are mutually exclusive. This means that if a limit value has been exceeded / undershot, it is not possible for another limit value to be exceeded or undershot at the same time.

There are three further specializations of the **OpcExclusiveLimitAlarm** within the OPC UA.

OpcExclusiveDeviationAlarm

This type of alarm is used when a slight deviation from defined limits is detected.

OpcExclusiveLevelAlarm

This type of alarm is used when a limit is exceeded. This typically affects an instrument - such as a temperature sensor. This type of alarm becomes active when the observed value is above an upper limit or below a lower limit.

OpcExclusiveRateOfChangeAlarm

This type of alarm is used to report an unusual change or missing change in a measured value with respect to the rate at which the value has changed. The alarm becomes active if the rate at which the value changes exceeds or falls below a defined limit.

Event Nodes with Alarm Conditions for non-exclusive Limits

The following types are used: [OpcClient](#), [OpcEventReceivedEventArgs](#), [OpcLimitAlarm](#), [OpcNonExclusiveLimitAlarm](#), [OpcNonExclusiveDeviationAlarm](#), [OpcNonExclusiveLevelAlarm](#) und [OpcNonExclusiveRateOfChangeAlarm](#).

This section describes a part of the API for topics related to: Alarm & Events, Alarm & Conditions.

A subclass of the **OpcLimitAlarm** events is the class **OpcNonExclusiveLimitAlarm**. As its name suggests, it serves to define limit alerts for non-exclusive boundaries. Such a limit alarm uses values for the limits that are not mutually exclusive. This means that when a limit has been exceeded / undershot, that at the same time another limit may be exceeded / undershot. The limits that are thereby violated can be checked with the properties *IsLowLow*, *IsLow*, *IsHigh* and *IsHighHigh* of the event data.

There are three further specializations of the **OpcNonExclusiveLimitAlarm** within the OPC UA.

OpcNonExclusiveDeviationAlarm

This type of alarm is used when a slight deviation from defined limits is detected.

OpcNonExclusiveLevelAlarm

This type of alarm is used when a limit is exceeded. This typically affects an instrument - such as a temperature sensor. This type of alarm becomes active when the observed value is above an upper limit or below a lower limit.

OpcNonExclusiveRateOfChangeAlarm

This type of alarm is used to report an unusual change or missing change in a measured value with respect to the rate at which the value has changed. The alarm becomes active if the rate at which the value changes exceeds or falls below a defined limit.

Editing the Address Space

Adding Nodes

The following types are used: [OpcClient](#), [OpcAddNode](#), [OpcAddNodeResult](#), [OpcAddNodeResultCollection](#), [OpcAddDataItemNode](#), [OpcAddAnalogItemNode](#), [OpcAddObjectNode](#), [OpcAddFolderNode](#), [OpcAddDataVariableNode](#) and [OpcAddPropertyNode](#).

While a Server provides a predefined set of “default nodes” to its Clients, the Clients can cause the Server to provide additional nodes. This is done by using the “AddNodes” interface of the server. Starting from the **OpcAddNode** class, the framework provides many further subclasses that can be used to create type-specific nodes. A new folder node can thus be created in the following way:

```
OpcAddNodeResult result = client.AddNode(new OpcAddFolderNode(
    name: "Jobs",
    nodeId: OpcNodeId.Null,
    parentNodeId: "ns=2;s=Machine"));
```

The parameters used provide the necessary minimum of information required. The first parameter “name” is used for the Name property, the DisplayName property, the SymbolicName property, and the Description property of the Node. The second parameter “nodeId” tells the Server which identifier is to be used for the node. In case there the identifier is already used in the address space of the Nodes of the Server for another Node, the Node is not created and the Client receives the result code “BadNodeIdRejected”. If OpcNodeId.Null is used instead, as shown in the example, the server will automatically create and assign a new identifier for the Node. The parameter “parentNodeId” finally defines the identifier of the parent Node under which the new Node is to be created in the tree.

Calling the AddNode method returns an instance of the OpcAddNodeResult class. In addition to the information about the result of the operation, it also provides the identifier that was finally used for the new Node:

```
if (result.IsGood)
    Console.WriteLine($"NodeId of 'Jobs': {result.NodeId}");
else
    Console.WriteLine($"Failed to add node: {result.Description}");
```

Besides the possibility to add a single Node, several Nodes can be added simultaneously:

```
OpcNodeId jobsNodeId = result.NodeId;

OpcAddNodeResultCollection results = client.AddNodes(
    new OpcAddDataVariableNode<string>("CurrentJob", jobsNodeId),
    new OpcAddDataVariableNode<string>("NextJob", jobsNodeId),
    new OpcAddDataVariableNode<int>("NumberOfJobs", jobsNodeId));
```

Calling the AddNodes method returns an instance of the OpcAddNodeResultCollection class. Which contains OpcAddNodeResult instances that can be evaluated and processed in the same way as described above.

In addition to the possibility of adding one or more Nodes at the same time, it is possible to pass entire trees of Nodes to the methods AddNode and AddNodes:

```

OpcAddNodeResultCollection results = client.AddNodes(
    new OpcAddObjectNode(
        "JOB001",
        nodeId: OpcNodeId.Null,
        parentNodeId: jobsNodeId,
        new OpcAddDataVariableNode<sbyte>("Status", -1),
        new OpcAddDataVariableNode<string>("Serial", "J01-DX-11.001"),
        new OpcAddAnalogItemNode<float>("Speed", 1200f) {
            EngineeringUnit = new OpcEngineeringUnitInfo(5067859, "m/s", "metre per
second"),
            EngineeringUnitRange = new OpcValueRange(5400, ),
            Definition = "DB100.DBW 0"
        },
        new OpcAddObjectNode(
            "Setup",
            new OpcAddPropertyNode<bool>("UseCutter"),
            new OpcAddPropertyNode<bool>("UseDrill")),
        new OpcAddObjectNode(
            "Schedule",
            new OpcAddPropertyNode<DateTime>("EarliestStartTime"),
            new OpcAddPropertyNode<DateTime>("LatestStartTime"),
            new OpcAddPropertyNode<TimeSpan>("EstimatedRunTime"))),
    new OpcAddObjectNode(
        "JOB002",
        nodeId: OpcNodeId.Null,
        parentNodeId: jobsNodeId,
        new OpcAddDataVariableNode<sbyte>("Status", -1),
        new OpcAddDataVariableNode<string>("Serial", "J01-DX-53.002"),
        new OpcAddAnalogItemNode<float>("Speed", 3210f) {
            EngineeringUnit = new OpcEngineeringUnitInfo(5067859, "m/s", "metre per
second"),
            EngineeringUnitRange = new OpcValueRange(5400, ),
            Definition = "DB200.DBW 0"
        },
        new OpcAddObjectNode(
            "Setup",
            new OpcAddPropertyNode<bool>("UseCutter"),
            new OpcAddPropertyNode<bool>("UseDrill")),
        new OpcAddObjectNode(
            "Schedule",
            new OpcAddPropertyNode<DateTime>("EarliestStartTime"),
            new OpcAddPropertyNode<DateTime>("LatestStartTime"),
            new OpcAddPropertyNode<TimeSpan>("EstimatedRunTime"))));

```

Such a tree can also be constructed using the corresponding properties:

```
var jobsNodeId = result.NodeId;

var job = new OpcAddObjectNode(
    name: "JOB003",
    nodeId: OpcNodeId.Null,
    parentNodeId: jobsNodeId);

job.Children.Add(new OpcAddDataVariableNode<sbyte>("Status", -1));
job.Children.Add(new OpcAddDataVariableNode<string>("Serial", "J01-DX-78.003"));
job.Children.Add(new OpcAddAnalogItemNode<float>("Speed", 1200f) {
    EngineeringUnit = new OpcEngineeringUnitInfo(5067859, "m/s", "metre per second"),
    EngineeringUnitRange = new OpcValueRange(5400, ),
    Definition = "DB100.DBW 0"
});

var setup = new OpcAddObjectNode("Setup");
setup.Children.Add(new OpcAddPropertyNode<bool>("UseCutter"));
setup.Children.Add(new OpcAddPropertyNode<bool>("UseDrill"));

job.Children.Add(setup);

var schedule = new OpcAddObjectNode("Schedule");
schedule.Children.Add(new OpcAddPropertyNode<DateTime>("EarliestStartTime"));
schedule.Children.Add(new OpcAddPropertyNode<DateTime>("LatestStartTime"));
schedule.Children.Add(new OpcAddPropertyNode<TimeSpan>("EstimatedRunTime"));

job.Children.Add(schedule);

OpcAddNodeResult result = client.AddNode(job);
```

In case there other type definitions shall be used for the Nodes than those provided by the corresponding subclasses of OpcAddNode, then it is possible to create object and variable Nodes based on their type definition:

```
client.AddObjectNode(OpcObjectType.DeviceFailureEventType, "FailureInfo");
client.AddVariableNode(OpcVariableType.XYArrayItem, "Coordinates");
```

Unlike adding nodes based on Foundation-defined type definitions, it is also possible to add nodes based on the identifier of their type definition. For this, the type to be used must be declared in advance via an object or variable-specific type definition. This can then be used again and again to add corresponding nodes:

```
// Declare Job Type
var jobType = OpcAddObjectNode.OfType(OpcNodeId.Of("ns=2;s=Types/JobType"));

client.AddNodes(
    jobType.Create("JOB001", nodeId: OpcNodeId.Null, parentNodeId: jobsNodeId),
    jobType.Create("JOB002", nodeId: OpcNodeId.Null, parentNodeId: jobsNodeId),
    jobType.Create("JOB003", nodeId: OpcNodeId.Null, parentNodeId: jobsNodeId),
    jobType.Create("JOB004", nodeId: OpcNodeId.Null, parentNodeId: jobsNodeId));

var scheduleNodeId = OpcNodeId.Parse("ns=2;s=Machine/JOB002/Schedule");

// Declare Shift Time Type
var shiftTimeType = OpcAddVariableNode.OfType(OpcNodeId.Of("ns=2;s=Types/ShiftTimeType"));

OpcAddNodeResult result = client.AddNode(new OpcAddObjectNode(
    "ShiftPlanning",
    nodeId: OpcNodeId.Null,
    parentNodeId: scheduleNodeId,
    shiftTimeType.Create("Early"),
    shiftTimeType.Create("Noon"),
    shiftTimeType.Create("Late")));
```

Deleting Nodes

The following types are used: [OpcClient](#), [OpcDeleteNode](#), [OpcStatus](#) and [OpcStatusCollection](#).

Server-provided and client-added nodes can be deleted using the Server's "DeleteNodes" interface. This requires primarily the identifier of the node to be deleted:

```
OpcStatus result = client.DeleteNode("ns=2;s=Machine/Jobs");
```

The possibility shown in the above example uses an instance of the `OpcDeleteNode` class for deletion, which by default also includes the deletion of all references pointing to the Node. However, if the references to the Node are to be retained, the parameter "includeTargetReferences" have to be set to the value "false":

```
OpcStatus result = client.DeleteNode(
    "ns=2;s=Machine/Jobs",
    includeTargetReferences: false);
```

Besides the possibility to delete a single Node, several Nodes can be deleted at the same time:

```
OpcStatusCollection results = client.DeleteNodes(
    new OpcDeleteNode("ns=2;s=Machine/Jobs/JOB001"),
    new OpcDeleteNode("ns=2;s=Machine/Jobs/JOB002"),
    new OpcDeleteNode("ns=2;s=Machine/Jobs/JOB003"));
```

Adding References

The following types are used: [OpcClient](#), [OpcAddReference](#), [OpcStatus](#) and [OpcStatusCollection](#).

Nodes that already exist in the address space of a Server can have different relationships to one another. These relationships are described by so-called references in the address space of the Server. While nodes are physically placed in the role of the parent or child Node, there may be more logical relationships

between them. These relationships thus serve to define more precisely the function and dependencies of the Nodes with each other. In addition, such trees can be used to define additional trees in the address space of the Server without having to reorganize existing nodes.

To add a reference, the Client uses the Server's "AddReferences" interface as follows:

```
client.AddReference(
    "ns=2;s=Machines/MAC01",
    targetNodeId: "ns=2;s=Plant",
    targetNodeCategory: OpcNodeCategory.Object);
```

The example shown adds a relationship starting from the Node with the identifier "ns=2;s=Plant" to the Node "ns=2;s=Machines/MAC01" of the type "Organizes".

The statement shown corresponds 1:1 to the result of the following example:

```
client.AddReference(
    "ns=2;s=Machines/MAC01",
    targetNodeId: "ns=2;s=Plant",
    targetNodeCategory: OpcNodeCategory.Object,
    direction: OpcReferenceDirection.ParentToChild,
    referenceType: OpcReferenceType.Organizes);
```

While the first identifier always indicates the source Node, the parameter "targetNodeId" specifies the identifier of the destination Node. The additional parameter "targetNodeCategory" must correspond to the Category property (= NodeClass according to Foundation) of the destination Node, because this ensures that the Server knows that the desired destination Node is sufficiently known.

In addition to the possibility to add a single reference, several references can be added simultaneously:

```
client.AddReferences(
    new OpcAddReference("ns=2;s=Machines/MAC01", "ns=2;s=Plant01",
    OpcNodeCategory.Object),
    new OpcAddReference("ns=2;s=Machines/MAC02", "ns=2;s=Plant01",
    OpcNodeCategory.Object),
    new OpcAddReference("ns=2;s=Machines/MAC03", "ns=2;s=Plant02",
    OpcNodeCategory.Object));
```

The example above organizes three nodes each representing one machine for itself below the "Machines" Node below the "Plant01" and the "Plant02" Nodes. The "Organizes" relationship used here by default means that the nodes are still available below the "Machines" Node, but also below the "Plant01" Node, the Nodes "MAC01" and "MAC02", as well as the Node "MAC03" below the Node "Plant02".

Deleting References

The following types are used: [OpcClient](#), [OpcDeleteReference](#), [OpcStatus](#) and [OpcStatusCollection](#).

Already existing references between the Nodes in the address space of the Server can be deleted via the "DeleteReferences" interface of the Server:

```
client.DeleteReference(
    nodeId: "ns=2;s=Machines/MAC03",
    targetNodeId: "ns=2;s=Plant");
```

In this case, the example shown deletes all "Organizes" references in the direction of the Node with the

identifier "ns=2;s=Machines/MAC03" as well as in the direction of the Node with the identifier "ns=2;s=Plant" between the both Nodes exist.

If, on the other hand, you want to delete only "Organizes" references from the source to the destination Node in one specific direction, you can do this as follows:

```
client.DeleteReference(
    nodeId: "ns=2;s=Machines/MAC03",
    targetNodeId: "ns=2;s=Plant",
    direction: OpcReferenceDirection.ChildToParent);
```

If, on the other hand, you want to delete references that are not of the "Organizes" type, they can be specified using the additional "referenceType" or "referenceTypeId" parameter as follows:

```
client.DeleteReference(
    nodeId: "ns=2;s=Machines/MAC03",
    targetNodeId: "ns=2;s=Plant",
    direction: OpcReferenceDirection.ChildToParent,
    referenceType: OpcReferenceType.HierarchicalReferences);
```

In addition to the possibility to delete individual references, several references can be deleted at the same time:

```
client.DeleteReferences(
    new OpcDeleteReference("ns=2;s=Machines/MAC01", "ns=2;s=Plant01"),
    new OpcDeleteReference("ns=2;s=Machines/MAC02", "ns=2;s=Plant01"),
    new OpcDeleteReference("ns=2;s=Machines/MAC03", "ns=2;s=Plant02"));
```

Client Configuration

General Configuration

The following types are used here: [OpcClient](#), [OpcCertificateStores](#) und [OpcCertificateStoreInfo](#).

In all code snippets depicted here the Client is always configured via the Code (if the default configuration of the Client is not applied). The **OpcClient** instance is the central port for configuring the Client application, the session parameter and the connection parameter. All settings concerning security can be found as an instance of the **OpcClientSecurity** class via the *Security* property of the Client. All settings concerning the Certificate Store can be found as an instance of the **OpcCertificateStores** class via the *CertificateStores* property of the Client.

If the Client shall also be configurable via XML, the configuration of the Client can be loaded either from a specific or a random XML file. The necessary steps are described in the section "Preparing the Client Configuration via XML".

As soon as the according preparations for configuring the Client configuration via XML have been made, the settings can be loaded as follows:

- Loading the configuration file via the App.config

```
client.Configuration =
    OpcApplicationConfiguration.LoadClientConfig("Opc.UaFx.Client");
```

- Loading the configuration file via the path to the XML file

```
client.Configuration =
OpcApplicationConfiguration.LoadClientConfigFile("MyClientAppNameConfig.xml");
```

Amongst others, the following options are provided for configuring the Client application:

- Configuring the application

- via Code:

```
// Default: Value of AssemblyTitleAttribute of entry assembly.
client.ApplicationName = "MyClientAppName";

// Default: A null reference to auto complete on connect to "urn:/" +
// ApplicationName
client.ApplicationUri = "http://my.clientapp.uri/";
```

- via XML (underneath the *OpcApplicationConfiguration* element):

```
<ApplicationName>MyClient Configured via XML</ApplicationName>
<ApplicationUri>http://myclient/application</ApplicationUri>
```

- Configuring the session parameters

- via Code:

```
client.SessionTimeout = 30000; // Default: 60000
client.SessionName = "My Session Name"; // Default: null
```

- via XML (underneath the *OpcApplicationConfiguration* element):

```
<DefaultSessionTimeout>600000</DefaultSessionTimeout>
```

- Configuring the connection parameters

- via Code:

```
client.OperationTimeout = 10000; // Default: 60000
client.DisconnectTimeout = 5000; // Default: 10000
client.ReconnectTimeout = 5000; // Default: 10000
```

- via XML (underneath the *OpcApplicationConfiguration* element):

```
<OperationTimeout>120000</OperationTimeout>
```

- Configuring the Certificate Store

- via Code:

```
// Default: ".\CertificateStores\Trusted"
client.CertificateStores.ApplicationStore.Path
    = @"%LocalApplicationData%\MyClientAppName\App Certificates";

// Default: ".\CertificateStores\Rejected"
client.CertificateStores.RejectedStore.Path
    = @"%LocalApplicationData%\MyClientAppName\Rejected Certificates";

// Default: ".\CertificateStores\Trusted"
client.CertificateStores.TrustedIssuerStore.Path
    = @"%LocalApplicationData%\MyClientAppName\Trusted Issuer Certificates";

// Default: ".\CertificateStores\Trusted"
client.CertificateStores.TrustedPeerStore.Path
    = @"%LocalApplicationData%\MyClientAppName\Trusted Peer Certificates";
```

- via XML (underneath the *OpcApplicationConfiguration* element):

```
<SecurityConfiguration>
  <ApplicationCertificate>
    <StoreType>Directory</StoreType>
    <StorePath>.\CertificateStores\App</StorePath>
    <SubjectName>CN=MyClient, C=US, S=Arizona, O=YourCompany,
DC=localhost</SubjectName>
    <!-- <Thumbprint>3a35fb798fc6dee8a7e7e4652b0e28fc14c6ee0f</Thumbprint> -->
  </ApplicationCertificate>

  <TrustedIssuerCertificates>
    <StoreType>Directory</StoreType>
    <StorePath>.\CertificateStores\Trusted</StorePath>
  </TrustedIssuerCertificates>

  <TrustedPeerCertificates>
    <StoreType>Directory</StoreType>
    <StorePath>.\CertificateStores\Trusted</StorePath>
  </TrustedPeerCertificates>

  <NonceLength>32</NonceLength>

  <RejectedCertificateStore>
    <StoreType>Directory</StoreType>
    <StorePath>.\CertificateStores\Rejected</StorePath>
  </RejectedCertificateStore>
</SecurityConfiguration>
```

Certificate Configuration

The following types are used here: [OpcClient](#), [OpcCertificateManager](#), [OpcClientSecurity](#), [OpcCertificateStores](#) and [OpcCertificateStoreInfo](#).

Certificates of the type .der, .pem, .pfx and .p12 are recommended. If the Client shall use a secure Server endpoint (where the **OpcSecurityMode** equals *Sign* or *SignAndEncrypt*), the certificate has to have a private key.

1. An **existing certificate** is loaded from any path:

```
var certificate = OpcCertificateManager.LoadCertificate("MyClientCertificate.pfx");
```

2. A **new certificate** is generated (in memory):

```
var certificate = OpcCertificateManager.CreateCertificate(client);
```

3. Save a certificate in any path:

```
OpcCertificateManager.SaveCertificate("MyClientCertificate.pfx", certificate);
```

4. Set the Client certificate:

```
client.Certificate = certificate;
```

5. The certificate has to be stored in the **Application Store**:

```
if (!client.CertificateStores.ApplicationStore.Contains(certificate))
    client.CertificateStores.ApplicationStore.Add(certificate);
```

- If **no or an invalid certificate** is used, a new certificate is generated / used by default. If the Client shall only use the mentioned certificate this function has to be deactivated. For **deactivating the function** set the property **AutoCreateCertificate** to the value *false*:

```
client.CertificateStores.AutoCreateCertificate = false;
```

User Identity Configuration

The following types are used here: [OpcClient](#), [OpcUserIdentity](#), [OpcClientIdentity](#), [OpcCertificateIdentity](#), [OpcClientSecurity](#), [OpcCertificateStores](#) and [OpcCertificateStoreInfo](#).

If a Server is expecting additional information about the user identity other than the Client certificate, the user has to be set via the *UserIdentity* property. You can choose between identities based on username-password and a certificate. If the Server supports an anonymized user identity, no special identity has to be set.

- Setting a user identity consisting of **username-password**:

```
client.Security.UserIdentity = new OpcClientIdentity("userName", "password");
```

- Setting a user identity via **certificate (with private key)**:

```
client.Security.UserIdentity = new OpcCertificateIdentity(new  
X509Certificate2("Doe.pfx"));
```

- Setting an **anonymous** user identity (pre-configured by default):

```
client.Security.UserIdentity = null;
```

Server Endpoint Configuration

The following types are used here: [OpcClient](#), [OpcClientSecurity](#), [OpcSecurityPolicy](#), [OpcSecurityMode](#), [OpcSecurityAlgorithm](#) and [OpcDiscoveryClient](#).

By default the Client chooses the Server endpoint with the **simplest security configuraton**. Hereby it chooses an endpoint with the **OpcSecurityMode** of *None*, *Sign* or *SignAndEncrypt*. According to the OPC Foundation the level of a policy of an endpoint serves as a relative measure for security mechanisms used by the endpoint. Per definition an endpoint with a higher level is more secure than an endpoint with a lower level. By default the Client ignores the Policy-Level of the endpoints.

- If the Client shall exclusively consider secure endpoints, the **UseOnlySecureEndpoints** property has to be set to the value *true*:

```
client.Security.UseOnlySecureEndpoints = true;
```

- If the Client shall choose an endpoint defining the highest Policy-Level, the **UseHighLevelEndpoint** property has to be set to the value *true*:

```
client.Security.UseHighLevelEndpoint = true;
```

- If the Client shall choose an endpoint with the best security configuration, the **EndpointPolicy** property has to be set as follows:

```
client.Security.EndpointPolicy = new OpcSecurityPolicy(  
OpcSecurityMode.None, OpcSecurityAlgorithm.Basic256);
```

4. To examine the endpoints provided by the Server use the **OpcDiscoveryClient**:

```
using (var client = new OpcDiscoveryClient("opc.tcp://localhost:4840/")) {
    var endpoints = client.DiscoverEndpoints();

    foreach (var endpoint in endpoints) {
        // Your code to operate on each endpoint.
    }
}
```

Further Security Settings

The following types are used here: [OpcClient](#), [OpcClientSecurity](#), [OpcCertificateValidationFailedEventArgs](#), [OpcCertificateStores](#) and [OpcCertificateStoreInfo](#).

A Server sends its certificate to the Client for authentication whilst connecting. Using the Server certificate, the Client can decide if to establish a connection to this Server and therefore trust it.

- For additional checking of the domains deposited in the Server certificate the property **VerifyServersCertificateDomains** can be used (deactivated by default):

```
client.Security.VerifyServersCertificateDomains = true;
```

- If the Client shall accept **only trustworthy** certificates, the default acceptance of all certificates has to be deactivated as follows:

```
client.Security.AutoAcceptUntrustedCertificates = false;
```

- As soon as the default acceptance of all certificates has been deactivated, a custom checking of certificates should be considered:

```
client.CertificateValidationFailed += HandleCertificateValidationFailed;
...
private void HandleCertificateValidationFailed(
    object sender,
    OpcCertificateValidationFailedEventArgs e)
{
    if (e.Certificate.SerialNumber == "...")
        e.Accept = true;
}
```

- If the Server certificate is categorized as **untrusted** it can be manually declared **trusted**. Therefore it has to be saved in the TrustedPeerStore:

```
// In context of the event handler the sender is an OpcClient.
var client = (OpcClient)sender;

if (!client.CertificateStores.TrustedPeerStore.Contains(e.Certificate))
    client.CertificateStores.TrustedPeerStore.Add(e.Certificate);
```

Configuration via XML

If the Client shall also be configurable via XML, the configuration of the Client can be directly loaded either from a specific or from a random XML file.

Using a certain XML file it has to show the following default XML tree:

```
<?xml version="1.0" encoding="utf-8" ?>
<OpcApplicationConfiguration
  xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ua="http://opcfoundation.org/UA/2008/02/Types.xsd"
  schemaLocation="AppConfig.xsd">
```

In case that a random XML file shall be used for the configuration, a .config file has to be created that refers to the XML file the configuration of the Client shall be loaded from. The following section shows which entries the .config file therefore has to contain and which structure the XML file has to show.

Creating and preparing the App.config of the application:

1. Add an App.config (if not already existing) to the project
2. Insert the following *configSections* element underneath the *configuration* element:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="Opc.UaFx.Client"
      type="Opc.Ua.ApplicationConfigurationSection,
        Opc.UaFx.Advanced,
        Version=2.0.0.0,
        Culture=neutral,
        PublicKeyToken=0220af0d33d50236" />
  </configSections>
```

3. Also insert the following *Opc.UaFx.Client* element underneath the *configuration* element:

```
<Opc.UaFx.Client>
  <ConfigurationLocation xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd">
    <FilePath>ClientConfig.xml</FilePath>
  </ConfigurationLocation>
</Opc.UaFx.Client>
```

4. The value of the *FilePath* element can show towards a random file path where the XML configuration file to be used can be found. The value depicted here would refer to a configuration file lying next to the application.
5. Save the changes to the App.config

Creating and preparing the XML configuration file:

1. Create an XML file with the file name used in the App.config and save the used path.
2. Insert the following default XML tree for XML configuration files:

```
<?xml version="1.0" encoding="utf-8" ?>
<OpcApplicationConfiguration
  xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ua="http://opcfoundation.org/UA/2008/02/Types.xsd"
  schemaLocation="AppConfig.xsd">
```

3. Save the changes to the XML file.

Client Application Delivery

This is how you prepare your OPC UA Client application for the use in productive environments.

Application certificates - Using a concrete certificate

For productive use don't use a certificate automatically generated by the Framework.

If you already have an appropriate certificate for your application you can load your PFX-based certificate from any random Store and assign it to the Client instance via the **OpcCertificateManager**:

```
var certificate = OpcCertificateManager.LoadCertificate("MyClientCertificate.pfx");
client.Certificate = certificate;
```

Note that the application name has to be included in the certificate as "Common Name" (CN) and has to match with the value of the *AssemblyTitle* attribute:

```
[assembly: AssemblyTitle("<Common Name (CN) in Certificate>")]
```

If that isn't the case you have to set the name used in the certificate via the **ApplicationName** property of the Client instance. If the "Domain Component" (DC) part is used in the certificate the value of the **ApplicationUri** property of the application has to show the same value:

```
client.ApplicationName = "<Common Name (CN) in Certificate>";
client.ApplicationUri = new Uri("<Domain Component (DC) in Certificate>");
```

If you don't already have an appropriate certificate you can use as an application certificate for your Client you should at least create and use a self-signed certificate via the Certificate Generator of the OPC Foundation. The Certificate Generator (Opc.Ua.CertificateGenerator.exe) included in the SDK of the Framework is opened as follows:

```
Opc.Ua.CertificateGenerator.exe -sp . -an MyClientAppName
```

The first parameter (-sp) sets saving the certificate in the current directory. The second parameter (-an) sets the name of the Client application using the application certificate. Replace "MyClientAppName" by the name of your Client application. Note that **the Framework for choosing the application certificate uses the value of the AssemblyTitle attribute and therefore the same value as stated in this attribute is used for "MyClientAppName"**. In alternative to the value in the *AssemblyTitle* attribute the value used in the application certificate can be set via the **ApplicationName** property of the Client instance:

```
client.ApplicationName = "MyDifferentClientAppName";
```

It is important that either the value of the *AssemblyTitle* attribute or the value of the **ApplicationName** property equals the value of the second parameter (-an). If you want to set further properties of the certificate as, for example, the validity in months (default 60 months) or the name of the company or the names of the domains the Client will be working on, call the generator with the parameter "/" in order to receive a list of all further / possible parameter values:

```
Opc.Ua.CertificateGenerator.exe /?
```

After the Certificate Generator was opened with the corresponding parameters, the folders "certs" and "private" are in the current directory. Without changing the names of the folders and the files, copy both folders in the directory that you set as Store for the application certificates. By default that is the folder

"Trusted" in the folder "CertificateStores" next to the application.

If you have set the parameter "ApplicationUri" (-au) you have to set the same value on the **ApplicationUri** property of the Client instance:

```
client.ApplicationUri = new Uri("<ApplicationUri>");
```

Configuration Surroundings - All files necessary for an XML-based configuration

If the application shall be configurable through a random XML file referenced in the App.config, App.config has to be in the same directory as the application and hold the name of the application as a prefix:

```
<MyClientAppName>.exe.config
```

If the application is configured through a (certain) XML file, ensure that the file is accessible for the application.

Licensing

The OPC UA SDK comes with an **evaluation license which can be used unlimited for each application run for 30 minutes**. If this restriction limits your evaluation options, you can request another evaluation license from us.

Just ask our support (via support@traeger.de) or let us consult you directly and clarify open questions with our developers!

After receiving your personalized **license key for OPC UA Client development** it has to be committed to the framework. Hereto insert the following code line into your application **before** accessing the **OpcClient class** for the first time. Replace *<insert your license code here>* with the license key you received from us.

```
Opc.UaFx.Client.Licenser.LicenseKey = "<insert your license code here>";
```

If you purchased a **bundle license key for OPC UA Client and Server development** from us, it has to be committed to the framework as follows:

```
Opc.UaFx.Licenser.LicenseKey = "<insert your license code here>";
```

Additionally you receive information about the license currently used by the framework via the *LicenseInfo* property of the **Opc.UaFx.Client.Licenser class** for Client licenses and via the **Opc.UaFx.Licenser class** for bundle licenses. This works as follows:

```
ILicenseInfo license = Opc.UaFx.Client.Licenser.LicenseInfo;

if (license.IsExpired)
    Console.WriteLine("The OPA UA SDK license is expired!");
```

Note that a once set **bundle license becomes ineffective by additionally committing a Client license key!**

In the course of development/evaluation, it is mostly irrelevant whether the test license or the license already purchased is being used. However, as soon as the application goes into productive use, it is

annoying if the application stops working during execution due to an invalid license. For this reason, we recommend implementing the following code snippet in the Client application and at least executing it when the application is started:

```
#if DEBUG
    Opc.UaFx.Client.Licenser.FailIfUnlicensed();
#else
    Opc.UaFx.Client.Licenser.ThrowIfUnlicensed();
#endif
```

You can receive further information about licensing, purchase or other questions directly on our product page at: opcua.traeger.de/en.

¹⁾ The *OPC UA Wrapper Server* is automatically started by the OPC UA Client or reused if it already exists. The *OPC UA Wrapper Server* 'wraps' the accesses of the OPC UA Client and sends them to the *OPC Classic Server* the answers received from the *OPC Classic Server* are sent to the *OPC UA Wrapper Server* as OPC UA responses to *OPC UA Client* back again.

Table of Contents

Tested? You want it?	1
The Client Frame	2
A Client for OPC UA	2
A Client for OPC Classic	2
Values of Node(s)	4
Reading Values	4
Writing Values	5
Processing Values	5
Browsing Nodes	7
Which Nodes does the Server have?	7
Inspecting Node by Node	8
High-Speed Browsing	8
High-Speed Browsing - Details	9
Subscriptions	11
Subscription Creation	11
Subscription Filtering	13
Structured Data	15
Simplest Access	15
Name-based Access	16
Dynamic Access	17
Typed Access	17
Generate Data Types	18
Define Data Types	19
Data Types with optional Fields	20
Historical Data	21
Nodes	24
Method Nodes	24
File Nodes	25
Datatype Nodes	27
Data Nodes	28
Data-Item Nodes	28
Data-Item Nodes for analog Values	28
Events	29
Event Subscription	29
Event Nodes	31
Event Nodes with Conditions	31
Event Nodes with Dialog Conditions	32
Event Nodes with Feedback Conditions	34
Event Nodes with Alarm Conditions	35
Event Nodes with discrete Alarm Conditions	35
Event Nodes with Alarm Conditions for Limits	36
Event Nodes with Alarm Conditions for exclusive Limits	36
Event Nodes with Alarm Conditions for non-exclusive Limits	37
Editing the Address Space	37
Adding Nodes	38
Deleting Nodes	41
Adding References	41
Deleting References	42
Client Configuration	43
General Configuration	43
Certificate Configuration	45