OPC UA SDK
Client + Server Development

Server Development Guide & Tutorial

Tested? You want it?

License Model Prices Quotation Order Now

C# ▼ |VB|

The Server Frame

1. Add reference to the **Opc.UaFx.Advanced** Server Namespace:
```
using Opc.UaFx.Server;
```
2. Instance of the OpcServer Class with desired standard base address:
```
var server = new OpcServer("opc.tcp://localhost:4840/");
```
3. Start Server and create Clients:
```
server.Start();
```
4. Your code to process Client requests:
```
// Your code to process client requests.
```
5. Close all sessions before closing the application and shut down the Server:
```
server.Stop();
```
6. Using the using block this looks as follows:
```
using (var server = new OpcServer("opc.tcp://localhost:4840/")) {
    server.Start();
    // Your code to process client requests.
}
```

Node Management

Node Creation

The following types are used: OpcServer, OpcNode, OpcNodeId, OpcNodeManager, OpcFolderNode, OpcVariableNode, OpcDataVariableNode, OpcDataVariableNode and OpcFileNode.

An **OpcNode** defines a data point of the Server. This can be a logical folder (**OpcFolderNode**), a variable (**OpcVariableNode**), a method (**OpcMethodNode**), a file (**OpcFileNode**) and much more. An **OpcNode** is unambiguously identified by an **OpcNodeId**. It exists of a value (a text, a number, ...) - the original ID - and of an index of the Namespace to which a Node is assigned. The Namespace is determinated by a Uri (= Uniform Resource Identifier). The Namespaces available are decided by the Node-Manager used by the Server.

Every Node-Manager defines at least one Namespace. Those Namespaces are used for the categorization of the Nodes of a Node-Manager, by which in return a Node can be assigned to a particular Node-Manager. The Default-Namespace of a Node-Manager is used in case no other Namespace is assigned to a Node. The Nodes defined by a Node-Manager are also called *Nodes in the Address Space of the Node-Manager*.

During the starting procedure of the Server the Server asks its Node-Managers to produce their Address Space, meaning their (static) Nodes. More (dynamic) Nodes can also be added to or removed from the Address Space of a Node-Manager during the execution of the Server. An always static Address Space can also be generated without an explicit custom Node-Manager by telling the Server the static Nodes for the Namespace *http://{host}/{path}/nodes/* directly. Instead of the Nodes of the static Address Space custom

Node-Managers can be defined.

- Create a custom Address Space with a Root Node for the Default Namespace *http://{host}/{path}/nodes/*:

```
var machineNode = new OpcFolderNode("Machine");
var machineIsRunningNode = new OpcDataVariableNode<bool>(machineNode, "IsRunning");

// Note: An enumerable of nodes can be also passed.
var server = new OpcServer("opc.tcp://localhost:4840/", machineNode);
```

- Define a custom Node-Manager:

```
public class MyNodeManager : OpcNodeManager
{
    public MyNodeManager()
        : base("http://mynamespace/")
    {
    }
}
```

- Create a custom Address Space with a Root Node by custom Node-Manager:

```
protected override IEnumerable<IOpcNode> CreateNodes(OpcNodeReferenceCollection
references)
{
    // Define custom root node.
    var machineNode = new OpcFolderNode(new OpcName("Machine",
this.DefaultNamespaceIndex));

    // Add custom root node to the Objects-Folder (the root of all server nodes):
    references.Add(machineNode, OpcObjectTypes.ObjectsFolder);

    // Add custom sub node beneath of the custom root node:
    var isMachineRunningNode = new OpcDataVariableNode<bool>(machineNode,
"IsRunning");

    // Return each custom root node using yield return.
    yield return machineNode;
}
```

- Introduce a custom Node-Manager to the Server:

```
// Note: An enumerable of node managers can be also passed.
var server = new OpcServer("opc.tcp://localhost:4840/", new MyNodeManager());
```

Node Accessibility

The following types are used: OpcServer, IOpcNode, OpcNodeId, OpcNodeManager, OpcFolderNode, OpcVariableNode, OpcDataVariableNode and OpcDataVariableNode.

Not all nodes should always be visible to all users of the server. To restrict the visibility of the nodes, any criteria can be applied via the own node manager. Restricting the accessibility by identity and a certain node might already match your needs. The following simple tree should illustrate the use.

```
protected override IEnumerable<IOpcNode> CreateNodes(OpcNodeReferenceCollection references)
{
    var machine = new OpcObjectNode(
            "Machine",
            new OpcDataVariableNode<int>("Speed", value: 123),
            new OpcDataVariableNode<string>("Job", value: "JOB0815"));

    references.Add(machine, OpcObjectTypes.ObjectsFolder);
    yield return machine;
}
```

If a user shall now have access to all nodes except the “Speed” node, the possible implementation of the `IsNodeAccessible` method can look like this:

```
protected override bool IsNodeAccessible(OpcContext context, OpcNodeId viewId, IOpcNodeInfo
node)
{
    if (context.Identity.DisplayName == "a")
        return true;

    if (context.Identity.DisplayName == "b" && node.Name.Value == "Speed")
        return false;

    return base.IsNodeAccessible(context, viewId, node);
}
```

Node Updates

The following types are used: OpcStatusCode and OpcVariableNode.

During the execution of the Server, various nodes often have to be updated according to the underlying system. Simple Object-, Folder- or Method-Nodes rarely need to be updated - however Variable-Nodes more regular. Depending on the available information it is possible to store the Timestamp and quality of the value together with the actual value of a Variable-Node. The following example outlines the handling.

```
var variableNode = new OpcVariableNode(...);

variableNode.Status.Update(OpcStatusCode.Good);
variableNode.Timestamp = DateTime.UtcNow;
variableNode.Value = ...;

variableNode.ApplyChanges(...);
```

It should be noted that Client applications are only informed about the changes to the node when `ApplyChanges` is called (assuming that a Client have completed an active subscription for the node and the Value attribute).

Values of Node(s)

Reading Values

The following types are used: OpcServer, OpcVariableNode, OpcDataVariableNode, OpcDataVariableNode, OpcReadAttributeValueCallback, OpcAttributeValue, OpcReadAttributeValueContext, OpcReadVariableValueCallback, OpcReadVariableValueContext and OpcVariableValue.

An **OpcNode** defines its metadata by *attributes*. Contrasting the generally always provided *attributes* like *Name*, *DisplayName* or *Description*, the *Value Attribute* is only available on *Variable-Nodes*. The values of *attributes* are saved by the concerning Node-Instances internally by default. If the value of another source of data is to be established, appropriate Callback-Methods for provision of the values can be defined. Here the signature of the ReadVariableValue-Callback-Method differenciates from the other ReadAttributeValue-Callback-Methods. In case of the *Value Attribute* instead of an **OpcAttributeValue** instance a **OpcVariableValue** instance is expected. This instance 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 status information about the quality of the value. Note that the Read-Callbacks are retrieved at every read operation of the metadata by a Client. This is the case when using the services Read and Browse.

- Set the default value of the Value Attribute of a Variable-Node:

```
var machineIsRunningNode = new OpcDataVariableNode<bool>("IsRunning", false);
```

- Set the value of the Value Attribute of a Variable-Node:

```
machineIsRunningNode.Value = true;
```

- Set the value of a Description Attribute:

```
machineIsRunningNode.Description = "My description";
```

- Inform all Clients (in case of an active subscription) about the attribute changes and accept changes:

```
machineIsRunningNode.ApplyChanges(server.SystemContext);
```

- Determine the value of the Description Attribute from another data source than the internal:

```
machineIsRunningNode.ReadDescriptionCallback = HandleReadDescription;
...
private OpcAttributeValue<string> HandleReadDescription(
        OpcReadAttributeValueContext context,
        OpcAttributeValue<string> value)
{
    return ReadDescriptionFromDataSource(context.Node) ?? value;
}
```

- Determine the value of the Value Attribute of a Variable-Node from another data source than the internal:

```
machineIsRunningNode.ReadVariableValueCallback = HandleReadVariableValue;
...
private OpcVariableValue<object> HandleReadVariableValue(
        OpcReadVariableValueContext context,
        OpcVariableValue<object> value)
{
    return ReadValueFromDataSource(context.Node) ?? value;
}
```

Writing Values

The following types are used: OpcServer, OpcVariableNode, OpcDataVariableNode, OpcDataVariableNode, OpcWriteAttributeValueCallback, OpcAttributeValue, OpcWriteAttributeValueContext, OpcWriteVariableValueCallback, OpcWriteVariableValueContext and OpcVariableValue.

An **OpcNode** defines its metadata by *attributes*. Contrasting the generally always provided *attributes* like *Name*, *DisplayName* or *Description*, the *Value Attribute* is only available on *Variable-Nodes*. The values of *attributes* are saved by the concerning Node-Instances internally by default. If the value is to be saved into another data source, appropriate Callback-Methods for saving the values can be defined. Here the signature of the WriteVariableValue-Callback-Method differenciates from the other WriteAttributeValue-Callback-Methods. In case of the *Value Attribute* instead of an **OpcAttributeValue** instance a **OpcVariableValue** instance is expected. This instance 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 status information about the quality of the value. Note that the Write-Callbacks are retrieved at every write operation of the metadata by a Client. This is the case when using the Write service.

- Set the default value of the Value Attribute of a Variable-Node:

```
var machineIsRunningNode = new OpcDataVariableNode<bool>("IsRunning", false);
```

- Set the value of the Value Attribute of a Variable-Node:

```
machineIsRunningNode.Value = true;
```

- Set the value of a Description Attribute:

```
machineIsRunningNode.Description = "My description";
```

- Inform all Clients (in case of an active subscription) about the attribute changes and accept changes:

```
machineIsRunningNode.ApplyChanges(server.SystemContext);
```

- Save the value of the Description Attribute from another data source than the internal:

```
machineIsRunningNode.WriteDescriptionCallback = HandleWriteDescription;
...
private OpcAttributeValue<string> HandleWriteDescription(
        OpcWriteAttributeValueContext context,
        OpcAttributeValue<string> value)
{
    return WriteDescriptionToDataSource(context.Node, value) ?? value;
}
```

- Save the value of the Value Attribute of a Variable-Node from another data source than the internal:

```
machineIsRunningNode.WriteVariableValueCallback = HandleWriteVariableValue;
...
private OpcVariableValue<object> HandleWriteVariableValue(
        OpcWriteVariableValueContext context,
        OpcVariableValue<object> value)
{
    return WriteValueToDataSource(context.Node, value) ?? value;
}
```

Historical Data

The following types are used: OpcServer, OpcNodeManager, IOpcNode, OpcHistoryValue, OpcHistoryModificationInfo, OpcValueCollection, OpcStatusCollection, OpcDeleteHistoryOptions, OpcReadHistoryOptions, IOpcNodeHistoryProvider and OpcNodeHistory<T>.

According to the OPC UA specification every Node of the category **Variable** supports the historical logging of the values of its *Value Attribute*. With every change in value of the *Value Attribute* the new value is saved together with the time stamp of the *Value Attribute*. These **pairs consisting of value and time stamp** are called **historical data**. The Server decides on where to save the data. However, the Client can determine via the *IsHistorizing Attribute* of the Node, if the Server supplies historical data for a Node and / or historically saves value changes. A Client can read, update, replace, delete or create historical data. Mostly, historical data is read by the Client.

The historical data provided by the Server can be administrated either directly in the Node-Manager of the current Node via the in-memory based Node-Historian or via a custom Historian. Note that according to OPC UA historical values always use their time stamp as a key. Correspondingly, it applies that a time stamp under every historical value of a Node is always unambiguous and therefore identifies only one certain value and its status information. Also, the historical data saved this way is distinguished between pure and modified historical data. The latter represents a kind of Changelog regarding to databanks. This Changelog can be used to process historical data that had been valid before a manipulation of the initial historical data. At the same time it is possible to retrace any changes made to the historical data. For example, if a historical value is replaced, the prior value is saved in the modified history. An historical value removed from the history is also saved in the modified history. Additionally, the kind of change, the time stamp of the change and the username of the instructor of the change is saved.

If a Client wants to read the (modified) historical data of a Node:

- the according Node has to be a Variable-Node, the record of historical data has to be activated and access to it must be cleared.
  - If an **OpcNodeHistorian** is used it takes over the activation and release of the historical data record:

```
// "this" points to the Node-Manager of the node.
var machineIsRunningHistorian = new OpcNodeHistorian(this, machineIsRunningNode);
```

  - Manual activation and release of the historical data history:

```
machineIsRunningNode.AccessLevel |= OpcAccessLevel.HistoryReadOrWrite;
machineIsRunningNode.UserAccessLevel |= OpcAccessLevel.HistoryReadOrWrite;

machineIsRunningNode.IsHistorizing = true;
```

- changes to the *Value Attributs* of the Variable-Node have to be monitored and transferred in a storage for the historical data:
  - If an **OpcNodeHistorian** is used it can be hired for automatical updates of the history:

```
machineIsRunningHistorian.AutoUpdateHistory = true;
```

  - For manual overwatch of the changes to the *Value Attribute* the *BeforeApplyChanges* event of the Variable-Node should be subscribed to:

```
machineIsRunningNode.BeforeApplyChanges += HandleBeforeApplyChanges;
...
private void HandleBeforeApplyChanges(object sender, EventArgs e)
{
    // Update (modified) Node History here.
}
```

- historical data has to be provided to the Client.
  - If an **IOpcNodeHistoryProvider** like the **OpcNodeHistorian** is used it has to be mentioned to the Server via the Node-Manager:

```
protected override IOpcNodeHistoryProvider RetrieveNodeHistoryProvider(IOpcNode
node)
{
    if (node == machineIsRunnigNode)
        return machineIsRunningHistorian;

    return base.RetrieveNodeHistoryProvider(node);
}
```

- If a custom **IOpcNodeHistoryProvider** is used its *ReadHistory* Method will be used:

```
public IEnumerable<OpcHistoryValue> ReadHistory(
        OpcContext context,
        DateTime? startTime,
        DateTime? endTime,
        OpcReadHistoryOptions options)
{
    // Read (modified) Node History here.
}
```

- If the Node-Manager shall itself take care of the history of its Nodes, the *ReadHistory* Method has to be implemented:

```
protected override IEnumerable<OpcHistoryValue> ReadHistory(
        IOpcNode node,
        DateTime? startTime,
        DateTime? endTime,
        OpcReadHistoryOptions options)
{
    // Read (modified) Node History here.
}
```

If a Client wants to generate the historical data of a Node the new values have to be filed in the history as well as in the modified history:

- If an **IOpcNodeHistoryProvider** like the **OpcNodeHistorian** is used it has to be mentioned to the Server via the Node-Manager:

```
protected override IOpcNodeHistoryProvider RetrieveNodeHistoryProvider(IOpcNode node)
{
    if (node == machineIsRunnigNode)
        return machineIsRunningHistorian;

    return base.RetrieveNodeHistoryProvider(node);
}
```

- If a custom **IOpcNodeHistoryProvider** is used its *CreateHistory* Method will be used:

```
public OpcStatusCollection CreateHistory(
        OpcContext context,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Create (modified) Node History here.
}
```

- If the Node-Manager shall itself take care of the history of its Nodes, the *CreateHistory* Method has to be implemented:

```
protected override OpcStatusCollection CreateHistory(
        IOpcNode node,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Create (modified) Node History here.
}
```

If a Client wants to delete the historical data of a Node, the values to be deleted have to be transferred into the modified history and deleted from the actual history. If modified history values are to be deleted this can happen directly in the modified history:

- If an **IOpcNodeHistoryProvider** like the **OpcNodeHistorian** is used it must be mentioned to the Server via the Node-Manager:

```
protected override IOpcNodeHistoryProvider RetrieveNodeHistoryProvider(IOpcNode node)
{
    if (node == machineIsRunnigNode)
        return machineIsRunningHistorian;

    return base.RetrieveNodeHistoryProvider(node);
}
```

- If a custom **IOpcNodeHistoryProvider** is used one of its *DeleteHistory* Methods will be used:

```
public OpcStatusCollection DeleteHistory(
        OpcContext context,
        OpcHistoryModificationInfo modificationInfo,
        IEnumerable<DateTime> times)
{
    // Delete Node History entries and add them to the modified Node History here.
}

public OpcStatusCollection DeleteHistory(
        OpcContext context,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Delete Node History entries and add them to the modified Node History here.
}

public OpcStatusCollection DeleteHistory(
        OpcContext context,
        OpcHistoryModificationInfo modificationInfo,
        DateTime? startTime,
        DateTime? endTime,
        OpcDeleteHistoryOptions options)
{
    // Delete Node History entries and add them to the modified Node History here.
}
```

- If the Node Manager itself shall take care of the history of its Nodes the *DeleteHistory* Methods have to be implemented:

```
protected override OpcStatusCollection DeleteHistory(
        IOpcNode node,
        OpcHistoryModificationInfo modificationInfo,
        IEnumerable<DateTime> times)
{
    // Delete Node History entries and add them to the modified Node History here.
}

protected override OpcStatusCollection DeleteHistory(
        IOpcNode node,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Delete Node History entries and add them to the modified Node History here.
}

protected override OpcStatusCollection DeleteHistory(
        IOpcNode node,
        OpcHistoryModificationInfo modificationInfo,
        DateTime? startTime,
        DateTime? endTime,
        OpcDeleteHistoryOptions options)
{
    // Delete Node History entries and add them to the modified Node History here.
}
```

If a Client wants to replace the historical data of a Node the values to be replaced have to be transferred into the modified history and replaced in the actual history:

- If an **IOpcNodeHistoryProvider** like the **OpcNodeHistorian** is used it has to be mentioned to the Server via the Node-Manager:

```
protected override IOpcNodeHistoryProvider RetrieveNodeHistoryProvider(IOpcNode node)
{
    if (node == machineIsRunnigNode)
        return machineIsRunningHistorian;

    return base.RetrieveNodeHistoryProvider(node);
}
```

- If a custom **IOpcNodeHistoryProvider** is used the *ReplaceHistory* Method is used:

```
public OpcStatusCollection ReplaceHistory(
        OpcContext context,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Replace Node History entries and add them to the modified Node History here.
}
```

- If the Node-Manager itself shall take care of the history of its Nodes the *ReplaceHistory* Methods has to be implemented:

```
protected override OpcStatusCollection ReplaceHistory(
        IOpcNode node,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Replace Node History entries and add them to the modified Node History here.
}
```

If a Client wants to generate historical data of a Node (if it does not exist yet) or replace it (if it already exists) - so according to OPC UA update it - non-existant entries have to be written into the history and modified history. Existant entries have to be replaced in the history and written into the modified history:

- If an **IOpcNodeHistoryProvider** like the **OpcNodeHistorian** is used it has to be mentioned to the Server via the Node-Manager:

```
protected override IOpcNodeHistoryProvider RetrieveNodeHistoryProvider(IOpcNode node)
{
    if (node == machineIsRunnigNode)
        return machineIsRunningHistorian;

    return base.RetrieveNodeHistoryProvider(node);
}
```

- If a custom **IOpcNodeHistoryProvider** is used the *UpdateHistory* Method is used:

```
public OpcStatusCollection UpdateHistory(
        OpcContext context,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Update (modified) Node History entries here.
}
```

- If the Node-Manager itself shall take care of the history of its Nodes the *UpdateHistory* Methods has to be implemented:

```
protected override OpcStatusCollection UpdateHistory(
        IOpcNode node,
        OpcHistoryModificationInfo modificationInfo,
        OpcValueCollection values)
{
    // Update (modified) Node History entries here.
}
```

In use of the Class **OpcNodeHistory<T>** the data of the history and the modified history can be administrated in the Store. Apart from several Methods operating the usual access scenarios to historical data, the separate constructors of the Class allow to set the capacity of the history. Also, the history can be “preloaded” and monitored via several events.

Definition of a history depending on the kind of historical data:

- The Class **OpcHistoryValue**is used as a type parameter for simple history:

```
var history = new OpcNodeHistory<OpcHistoryValue>();
```

- The Class **OpcModifiedHistoryValue** is used as a type parameter for modified history:

```
var modifiedHistory = new OpcNodeHistory<OpcModifiedHistoryValue>();
```

Using the Class **OpcNodeHistory<T>** the usual history scenarios like Read, Create, Delete, Replace and Update can be implemented as follows:

- Szenario: **Create History**:

```
var results = OpcStatusCollection.Create(OpcStatusCode.Good, values.Count);

for (int index = ; index < values.Count; index++) {
    var result = results[index];
    var value = OpcHistoryValue.Create(values[index]);

    if (MatchesValueType(value)) {
        if (history.Contains(value.Timestamp)) {
            result.Update(OpcStatusCode.BadEntryExists);
        }
        else {
            history.Add(value);

            var modifiedValue = value.CreateModified(modificationInfo);
            modifiedHistory.Add(modifiedValue);

            result.Update(OpcStatusCode.GoodEntryInserted);
        }
    }
    else {
        result.Update(OpcStatusCode.BadTypeMismatch);
    }
}

return results;
```

- Scenario: **Delete History**
  - Via time stamp:

```
var results = OpcStatusCollection.Create(OpcStatusCode.Good, times.Count());

int index = ;

foreach (var time in times) {
    var result = results[index++];

    if (this.history.Contains(time)) {
        var value = this.history[time];
        this.history.RemoveAt(time);

        var modifiedValue = value.CreateModified(modificationInfo);
        this.modifiedHistory.Add(modifiedValue);
    }
    else {
        result.Update(OpcStatusCode.BadNoEntryExists);
    }
}

return results;
```

  - Via values:

```
var results = OpcStatusCollection.Create(OpcStatusCode.Good, values.Count);

for (int index = ; index < values.Count; index++) {
    var timestamp = OpcHistoryValue.Create(values[index]).Timestamp;
    var result = results[index];

    if (history.Contains(timestamp)) {
        var value = history[timestamp];
        history.RemoveAt(timestamp);

        var modifiedValue = value.CreateModified(modificationInfo);
        modifiedHistory.Add(modifiedValue);
    }
    else {
        result.Update(OpcStatusCode.BadNoEntryExists);
    }
}

return results;
```

- Via time span:

```
var results = new OpcStatusCollection();

bool isModified = (options & OpcDeleteHistoryOptions.Modified)
        == OpcDeleteHistoryOptions.Modified;

if (isModified) {
    modifiedHistory.RemoveRange(startTime, endTime);
}
else {
    var values = history.Enumerate(startTime, endTime).ToArray();
    history.RemoveRange(startTime, endTime);

    for (int index = ; index < values.Length; index++) {
        var value = values[index];
        modifiedHistory.Add(value.CreateModified(modificationInfo));

        results.Add(OpcStatusCode.Good);
    }
}

return results;
```

- Scenario: **Replace History**:

```
var results = OpcStatusCollection.Create(OpcStatusCode.Good, values.Count);

for (int index = ; index < values.Count; index++) {
    var result = results[index];
    var value = OpcHistoryValue.Create(values[index]);

    if (this.MatchesNodeValueType(value)) {
        if (this.history.Contains(value.Timestamp)) {
            var oldValue = this.history[value.Timestamp];
            history.Replace(value);

            var modifiedValue = oldValue.CreateModified(modificationInfo);
            modifiedHistory.Add(modifiedValue);

            result.Update(OpcStatusCode.GoodEntryReplaced);
        }
        else {
            result.Update(OpcStatusCode.BadNoEntryExists);
        }
    }
    else {
        result.Update(OpcStatusCode.BadTypeMismatch);
    }
}

return results;
```

- Scenario: **Update History**:

```
var results = OpcStatusCollection.Create(OpcStatusCode.Good, values.Count);

for (int index = ; index < values.Count; index++) {
    var result = results[index];
    var value = OpcHistoryValue.Create(values[index]);

    if (MatchesValueType(value)) {
        if (history.Contains(value.Timestamp)) {
            var oldValue = this.history[value.Timestamp];
            history.Replace(value);

            var modifiedValue = oldValue.CreateModified(modificationInfo);
            modifiedHistory.Add(modifiedValue);

            result.Update(OpcStatusCode.GoodEntryReplaced);
        }
        else {
            history.Add(value);

            var modifiedValue = value.CreateModified(modificationInfo);
            modifiedHistory.Add(modifiedValue);

            result.Update(OpcStatusCode.GoodEntryInserted);
        }
    }
    else {
        result.Update(OpcStatusCode.BadTypeMismatch);
    }
}

return results;
```

- Scenario: **Read History**:

```
bool isModified = (options & OpcReadHistoryOptions.Modified)
        == OpcReadHistoryOptions.Modified;

if (isModified) {
    return modifiedHistory
            .Enumerate(startTime, endTime)
            .Cast<OpcHistoryValue>()
            .ToArray();
}

return history
        .Enumerate(startTime, endTime)
        .ToArray();
```

Nodes

Method Nodes

The following types are used here: OpcNodeManager, OpcMethodNode and OpcMethodContext.

Code sections that fulfill an isolated task are called subprograms in programming. Those subprograms are often described simply as functions or methods. Those kind of methods can be called via method Nodes in

the OPC UA. A method Node is defined by the **OpcMethodNode** class. Method Nodes are called by an OPC UA Client via the server-sided **Call** service.

The framework defines a method Node through the one-on-one application of a function pointer (delegate in C#) to a Node of the category *OpcNodeCategory.Method*. Herefore the structure of the delegate is examined via .NET reflections and based on that the method Node with its *IN* and *OUT* arguments is defined.

Define a method Node in the Node manager:

1. through a method without a parameter:

```
var startMethodNode = new OpcMethodNode(
        machineNode,
        "StartMachine",
        new Action(this.StartMachine));
...
private void StartMachine()
{
    // Your code to execute.
}
```

2. through a method with a parameter:

```
var startMethodNode = new OpcMethodNode(
        machineNode,
        "StartMachine",
        new Action<int>(this.StartMachine));
...
private void StartMachine(int reasonNumber)
{
    // Your code to execute.
}
```

3. through a method with a callback value:

```
var startMethodNode = new OpcMethodNode(
        machineNode,
        "StartMachine",
        new Func<int>(this.StartMachine));
...
private int StartMachine()
{
    // Your code to execute.
    return statusCode;
}
```

4. through a method with a parameter and a callback value:

```
var startMethodNode = new OpcMethodNode(
        machineNode,
        "StartMachine",
        new Func<int, string, int>(this.StartMachine));
...
private int StartMachine(int reasonNumber, string operatorName)
{
    // Your code to execute.
    return statusCode;
}
```

5. through a method that needs access to contextual information about the actual “Call” (for this the first parameter has to be of the type **OpcMethodNodeContext**):

```
var startMethodNode = new OpcMethodNode(
        machineNode,
        "StartMachine",
        new Func<OpcMethodNodeContext, int, int>(this.StartMachine));
...
private int StartMachine(OpcMethodNodeContext context, int reasonNumber)
{
    // Your code to execute.

    this.machineStateVariable.Value = "Started";
    this.machineStateVariable.ApplyChanges(context);

    return statusCode;
}
```

Also, there is the option to supply additional information about the arguments (callback values and parameters) of a method via the **OpcArgument attribute**. This information is considered for the definition of the method Node arguments and supplied to every Client when browsing the Node. Such a definition of additional information will look as follows:

```
[return: OpcArgument("Result", Description = "The result code of the machine driver.")]
private int StartMachine(
        [OpcArgument("ReasonNumber", Description = "0: Maintenance, 1: Manufacturing, 2:
Service")]
        int reasonNumber,
        [OpcArgument("OperatorName", Description = "Optional. Name of the operator of the
current shift.")]
        string operatorName)
{
    // Your code to execute.
    return 10;
}
```

File Nodes

The following types are used: OpcNodeManager and OpcFileNode.

Nodes of the type **FileType** define per OPC UA Specification definition certain Property Nodes and Method Nodes allowing to access a data stream as if accessing. Exclusive information about the content of the logical or physical file is provided. According to the specification, a possibly existing path to the data is not provided. The access to the file itself is realized by Open, Close, Read, Write, GetPosition and SetPosition. The data is always processed binarily. As in every other platform in OPC UA you can set a mode that provides the kind of planned data access when opening Open. You can also request exclusive access to a file. After opening the Open Method you receive a numeric key for further file handle. This key always has to be passed in the Methods Read, Write, GetPosition and SetPosition. Once a file is opened it has to be closed again when no longer needed.

Define a File Node in the Node-Manager:

```
var protocollFileNode = new OpcFileNode(
        machineNode,
        "Protocoll.txt",
        new FileInfo(@"..\Protocoll.log"));
```

All other operations to work with the represented file are already provided by the **OpcFileNode** class.

Datatype Nodes

The following types are used: OpcNodeManager, OpcNodeId, OpcDataTypeAttribute, OpcDataTypeNode and OpcEnumMemberAttribute.

In some scenarios it is necessary to describe the Server provided data using user-defined data types. Such a data type can for example be an enumeration. Depending from the entry (differentiable by their name) there can be used a different value or a combination of values which are / can again represented by other entries. In the last case such an enumeration is called a flag-enumeration. Are the bits of an enum-entry bitwise set in its flag enumeration value, then the whole value applies although it does not match the exact value of the entry (because of other enum-entries are applicable). The thereby valid (combinations of) values have to be provided as Name-Value-Pairs by the Server using a specific ID, to enable Read- and Write-Access on Nodes - which are using the type of enumeration - using valid values. To publish a user-defined enumeration as an enumeration in the Address Space of the Server, the enumeration have to use the **OpcDataTypeAttribute**. Using this attribute the data of the **OpcNodeId** associated with the type of enumeration is defined. Finally, the user-defined data type must be published via one of the Server's Node-Managers. How this works in detail can be seen in the following code example:

```
// Define the node identifier associated with the custom data type.
[OpcDataType(id: "MachineStatus", namespaceIndex: 2)]
internal enum MachineStatus : int
{
    Unknown = ,
    Stopped = 1,
    Started = 2,
    Waiting = 3,
    Suspended = 4
}

...

// MyNodeManager.cs
protected override IEnumerable<IOpcNode> CreateNodes(OpcNodeReferenceCollection references)
{
    ...

    // Publish a new data type node using the custom type.
    return new IOpcNode[] { ..., new OpcDataTypeNode<MachineStatus>() };
}
```

Additional information about the individual enum-entries can be defined using the **OpcEnumMemberAttribute**. The thereby optional supported *Description* property is only used in case there the entry is part of a flag enumeration. The previously represented enumeration could then look like as follows:

```
[OpcDataType(id: "MachineStatus", namespaceIndex: 2)]
internal enum MachineStatus : int
{
    Unknown = ,
    Stopped = 1,
    Started = 2,

    [OpcEnumMember("Paused by Job")]
    WaitingForOrders = 3,

    [OpcEnumMember("Paused by Operator")]
    Suspended = 4,
}
```

Data Nodes

The following types are used: OpcNodeManager, OpcDataVariableNode and OpcDataVariableNode.

With the help of the **OpcDataVariableNode** it is possible to provide simple scalar as well as complex data structures. In addition to the value itself, these self-describing nodes also provide information about the data types valid for the value. This includes, for example, the length of an array. Such a data node is created as follows:

```
// Node of the type Int32
var variable1Node = new OpcDataVariableNode<int>(machineNode, "Var1");

// Node of the type Int16
var variable2Node = new OpcDataVariableNode<short>(machineNode, "Var2");

// Node of the type String
var variable3Node = new OpcDataVariableNode<string>(machineNode, "Var3");

// Node of the type float-array
var variable4Node = new OpcDataVariableNode<float[]>(machineNode, "Var4", new float[] {
0.1f, 0.5f });

// Node of the type MachineStatus enum
var variable5Node = new OpcDataVariableNode<MachineStatus>(machineNode, "Var5");
```

Data-Item Nodes

The following types are used: OpcNodeManager, OpcDataItemNode and OpcDataItemNode.

The data provided by an OPC UA Server often does not come directly 1: 1 from the underlying system of the server. Even though these data variables can be provided by means of instances of **OpcDataVariableNodes**, the origin or the *Definition* - how a value of a data point is established - is of interest for the correct further processing and interpretation. Especially when used by third parties, this information is not only a part of the documentation, but also a helpful tool for internal data processing. This is exactly where the **OpcDataItemNode** comes in and provides the *Definition* property with the necessary information about the realization of the values of the Data-Item Node. In addition, the *ValuePrecision* property provides a value that tells you how accurate the values can be. This Node is defined as follows:

```
var statusNode = new OpcDataItemNode<int>(machineNode, "Status");
statusNode.Definition = "Status Code in low word, Progress Code in high word encoded in
BCD";
```

In general, the value of the *Definition* property depends on the manufacturer.

Data-Item Nodes for analog Values

The following types are used: OpcNodeManager, OpcAnalogItemNode, OpcAnalogItemNode, OpcValueRange and OpcEngineeringUnitInfo.

Nodes of the type **AnalogItemType** essentially represent a specialization of the **OpcDataItemNode**. The additionally offered properties allow the provided analog values to be specified more precisely. The *InstrumentRange* serves as the definition of the range of values used by the source of the analog data. The *EngineeringUnit* is used to classify the unit of measure associated with the value in accordance with the UNECE Recommendations N° 20. These recommendations are based on the International System of Units, short SI Units. These two properties are supplemented by the *EngineeringUnitRange* which can be provided according to the *EngineeringUnit* value range valid during normal operation. Such a node can then be defined in the Node-Manager as follows:

```
var temperatureNode = new OpcAnalogItemNode<float>(machineNode, "Temperature");

temperatureNode.InstrumentRange = new OpcValueRange(80.0, -40.0);
temperatureNode.EngineeringUnit = new OpcEngineeringUnitInfo(4408652, "°C", "degree
Celsius");
temperatureNode.EngineeringUnitRange = new OpcValueRange(70.8, 5.0);
```

The UnitID expected in the constructor of the **OpcEngineeringUnitInfo** can be taken from the UNECE table for measurement units at the OPC Foundation:UNECE units of measure in OPC UA

NodeSets

NodeSets describe the contents of the address space of a server in the form of the XML (eXtensible Markup Language). Part of the description is the definition of data types and their transported logical (data-dependent information) as well as physical (in memory) structure. In addition, the definitions of node types as well as concrete node instances can be found in a NodeSet. The root element “UANodeSet” also describes the relationships between the individual nodes defined in the NodeSet, but also defined in other NodeSets, as well as in the specification.

Companion specifications not only extend the general definition of the address space, but also provide their own data types and node types. As a result, one or more NodeSets exist for each companion specification. For a server to meet a companion specification, the server must implement the types and behaviors defined in this specification.

Another case in which NodeSets are used as a description of the address space is the configuration of controllers. This means that the entire configured configuration of a controller can be exported from configuration software such as TIA Portal and used to initialize the OPC UA server of a controller.

Regardless of the source of a NodeSet, if a NodeSet is to be used by a server, it must provide the necessary logic to import and implement the address space described in the NodeSet. How it works shows the next sections.

Import NodeSets

The following types are used here: OpcNodeSet, OpcNodeSetManager and OpcNodeManager.

Nodes which are described in a NodeSet can be imported via the **OpcNodeSetManager** 1: 1:

```
var umatiManager = OpcNodeSetManager.Create(
        OpcNodeSet.Load(@".\umati.xml"),
        OpcNodeSet.Load(@".\umati-instances.xml"));

using (var server = new OpcServer("opc.tcp://localhost:4840/", umatiManager)) {
    server.Start();
    ...
}
```

When calling **OpcNodeSetManager.Create(...)**, 1-n NodeSets can be specified. The OpcNodeManager created when calling OpcNodeSetManager.Create takes care of importing the NodeSets and thus generates the nodes defined in the NodeSets within the address space of the server when starting the server. On the other hand, if you want a custom NodeManager to handle the import of a NodeSet, you can do so simply by overriding the ImportNodes method of the OpcNodeManager class:

```
protected override IEnumerable<OpcNodeSet> ImportNodes()
{
    yield return OpcNodeSet.Load(@".\umati.xml");
    yield return OpcNodeSet.Load(@".\umati-instances.xml");
}
```

Implement NodeSets

The following types are used here: OpcNodeManager and IOpcNode.

Often the simple import of a NodeSet is not enough, as the associated logic for the connection of the underlying system is still missing. This logic is necessary, for example, to map the reading of a node to the reading of, for example, a word in a data block. The same applies again to the writing of a node.

To implement this logic, the **OpcNodeManager.ImplementNode** method within a custom NodeManager will be overridden as follows:

```
protected override void ImplementNode(IOpcNode node)
{
    // Implement your Node(s) here.
}
```

For example, in the case of a UMATI node set, the logic for simulating the status of a lamp could be implemented as follows:

```
private static readonly OpcNodeId LampTypeId = "ns=2;i=1041";
private readonly Random random = new Random();

protected override void ImplementNode(IOpcNode node)
{
    if (node is OpcVariableNode variableNode && variableNode.Name == "2:Status") {
        if (variableNode?.Parent is OpcObjectNode objectNode && objectNode.TypeDefinitionId
== LampTypeId) {
            variableNode.ReadVariableValueCallback = (context, value) => new
OpcVariableValue<object>(this.random.Next(, 2));
        }
    }
}
```

Events

Event Reporting

The following types are used: OpcServer, OpcNodeManager, OpcEventSeverity and OpcText.

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

Events inform a subscriber about operations, conditions, and system specific circumstances. Such information can be delivered directly through **global events** to interested parties. A global event can be triggered and dispatched either by a **OpcNodeManager** or by a **OpcServer** instance. For this, the framework offers various method overloads of the *ReportEvent(...)*- method. To trigger a global event using a **OpcServer** instance, you have the following options:

```
var server = new OpcServer(...);
// ...

server.ReportEvent(
        OpcEventSeverity.Medium,
        "Recognized a medium urgent situation.");

// Same usage as before + arguments support.
server.ReportEvent(
        OpcEventSeverity.Medium,
        "Recognized a medium urgent situation at machine {0}.",
        machineId);

// Sames usage as before + source node.
server.ReportEvent(
        sourceNode,
        OpcEventSeverity.Medium,
        "Recognized a medium urgent situation.");

// Same usage as before + arguments support.
server.ReportEvent(
        sourceNode,
        OpcEventSeverity.Medium,
        "Recognized a medium urgent situation at machine {0}.",
        machineId);

// Same usage as before + explicit source information.
server.ReportEvent(
        sourceNodeId,
        sourceNodeName,
        OpcEventSeverity.Medium,
        "Recognized a medium urgent situation.");

// Same usage as before + arguments support.
server.ReportEvent(
        sourceNodeId,
        sourceNodeName,
        OpcEventSeverity.Medium,
        "Recognized a medium urgent situation at machine {0}.",
        machineId);
```

The same method overloads can also be found as instance methods of a **OpcNodeManager** instance.

Event Nodes

The following types are used: OpcServer, OpcNodeManager and OpcEventNode.

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

It is not always appropriate to send events globally through the Server 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. The base class of all Event Nodes represents the class **OpcEventNode**. Using these it is possible to provide simple events in the form of local events, as shown in the 'Providing Events' section. Since this is a Node, the Event Node (**OpcEventNode**) must first be created in the **OpcNodeManager** like any other Node:

```
var activatedEvent = new OpcEventNode(machineOne, "Activated");
```

So that an event can now be sent by this Event Node, it have to be defined as a 'Notifier'. For this purpose, the Event Node is registered as a 'Notifier' for each Node via which a subscription is to be able to receive the local event. This works as follows:

```
machineOne.AddNotifier(this.SystemContext, activatedEvent);
```

Before an event is triggered, all information relevant for the event must be entered on the Event Node. Which information is changed and how it is determined depends on the individual case of application. In general, this works as follows:

```
activatedEvent.SourceNodeId = sourceNodeId;
activatedEvent.SourceName = sourceNodeName;
activatedEvent.Severity = OpcEventSeverity.Medium;
activatedEvent.Message = "Recognized a medium urgent situation.";
```

Additionally the Event Node **OpcEventNode** offers further properties:

```
// Server generated value to identify a specific Event
activatedEvent.EventId = ...;

// The time the event occured
activatedEvent.Time = ...;

// The time the event has been received by the underlaying system / device
activatedEvent.ReceiveTime = ...;
```

After configuring the event to be created, only the *ReportEvent(...)*- method of the **OpcEventNode** instance needs to be called:

```
activatedEvent.ReportEvent(this.SystemContext);
```

This will automatically invoke the *ApplyChanges(...)* method on the Node, creates a snapshot of the Node and send it to all subscribers. After calling the *ReportEvent(...)* method, the properties of the **OpcEventNode** can be changed as desired.

Generally, after a subscriber is only informed of events, as long as he is in contact with the Server and has subscribed events, a subscriber will not know what events have already occurred prior to establishing a connection to the Server. If a Server is to inform subscribers retrospectively about past events, these can be provided by the Server on request from the subscriber as follows:

```
machineOne.QueryEventsCallback = (context, events) => {
    // Ensure that an re-entrance upon notifier cross-references will not add
    // events to the collection which are already stored in.
    if (events.Count != )
        return;

    events.Add(activatedEvent.CreateEvent(context));
};
```

It should be noted at this point that each Node under which an Event Node has been registered as 'Notifier' must separately specify such a callback. **In general, however, the Server is not obliged to provide past events.** In addition, it is always possible to create a snapshot of the Node using the *CreateEvent(...)* method of the **OpcEventNode**, to cache it and to provide the cached snapshots when the *QueryEventsCallback* is called.

Event Nodes with Conditions

The following types are used: OpcServer, OpcNodeManager and OpcConditionNode.

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

A specialization of the **OpcEventNode** (presented in section 'Providing Event Nodes') is the class **OpcConditionNode**. It serves the definition of events to which certain conditions are attached. Only in case the condition given to the Event Node is true an event should be triggered. Information associated with the Node also includes information about the state of this condition, as well as information associated with the evaluation of the condition. Since this information can vary in complexity depending on the scenario, the **OpcConditionNode** represents the base class of all Event Nodes to which a condition is attached. Such a Node is created like an **OpcEventNode**. In the following, therefore, only the specific further properties are shown:

```
var maintenanceEvent = new OpcConditionNode(machineOne, "Maintenance");

// Interesting for a client yes or no
maintenanceEvent.IsRetained = true; // = default

// Condition is enabled or disabled
maintenanceEvent.IsEnabled; // use ChangeIsEnabled(...)

// Status of the source the condition is based upon
maintenanceEvent.Quality = ...;
```

With the *AddComment(...)* method and the child Node of the same name, the *Comment* property of the Node can be changed. The result of the change can be evaluated using the following properties:

```
// Identifier of the user who supplied the Comment
maintenanceEvent.ClientUserId = ...;

// Last comment provided by a user
maintenanceEvent.Comment = ...;
```

If the same Event Node is to be processed in multiple tracks, then a new event branch can be opened. For this the *CreateBranch(...)* method of the Node can be used. The unique key for the branch is stored in the *BranchId* property. The following snippet shows the most important parts to work with branches of a **OpcConditionNode**:

```
// Uses a new GUID as BranchId
var maintenanceBranchA = maintenanceEvent.CreateBranch(this.SystemContext);

// Uses a custom NodeId as BranchId
var maintenanceBranchB = maintenanceEvent.CreateBranch(this.SystemContext, new
OpcNodeId(10001));

...

// Identifies the branch of the event
maintenanceEvent.BranchId = ...;

// Previous severity of the branch
maintenanceEvent.LastSeverity = ...;
```

Event Nodes with Dialog Conditions

The following types are used: OpcServer, OpcNodeManager and OpcDialogConditionNode.

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

A specialization of the **OpcConditionNode** is the **OpcDialogConditionNode**. The condition associated with this Node is a dialog with the subscribers. In this case, such a condition consists of a prompt, response options as well as information which option is 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 response 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. If a Node has been created as usual, the corresponding properties can be defined according to the scenario:

```
var outOfMaterial = new OpcDialogConditionNode(machineOne, "MaterialAlert");

outOfMaterial.Message = "Out of Material"; // Generic event message
outOfMaterial.Prompt = "The machine is out of material. Refill material supply to
continue.";
outOfMaterial.ResponseOptions = new OpcText[] { "Continue", "Cancel" };
outOfMaterial.DefaultResponse = ; // Index of ResponseOption to use
outOfMaterial.CancelResponse = 1;  // Index of ResponseOption to use
outOfMaterial.OkResponse = ;      // Index of ResponseOption to use
```

A dialog condition answered by a subscriber is then handled by the Node's *RespondCallback* as follows.

```
outOfMaterial.RespondCallback = this.HandleOutOfMaterialResponse;

...

private OpcStatusCode HandleOutOfMaterialResponse(
        OpcNodeContext<OpcDialogConditionNode> context,
        int selectedResponse)
{
    // Handle the response
    if (context.Node.OkResponse == selectedResponse)
        ContinueJob();
    else
        CancelJob();

    // Apply the response
    context.Node.RespondDialog(context, response);

    return OpcStatusCode.Good;
}
```

Event Nodes with Feedback Conditions

The following types are used: OpcServer, OpcNodeManager and OpcAcknowledgeableConditionNode.

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

Based on the **OpcConditionNode**, the **OpcAcknowledgeableConditionNode** 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 serve to control further processes as well as to easily acknowledge hints and warnings. The specified 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 Node provides two child Nodes *Confirm* and *Acknowledge*. 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. Such a Node is created as already known:

```
var outOfProcessableBounds = new OpcAcknowledgeableConditionNode(machineOne,
"OutOfBoundsAlert");

// Define the condition as: Needs to be acknowledged
outOfProcessableBounds.ChangeIsAcked(this.SystemContext, false);

// Define the condition as: Needs to be confirmed
outOfProcessableBounds.ChangeIsConfirmed(this.SystemContext, false);
```

During the further process flows, an incoming feedback can be checked using the Node's *IsAcked* and *IsConfirmed* property:

```
if (outOfProcessableBounds.IsAcked) {
    ...
}

if (outOfProcessableBounds.IsConfirmed) {
    ...
}
```

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 should at least use the *Acknowledge* method, as it is defined by the specification as “stronger”.

Event Nodes with Alarm Conditions

The following types are used: OpcServer, OpcNodeManager and OpcAlarmConditionNode.

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

The most important implementation of the **OpcAcknowledgeableConditionNode** in OPC UA may be the **OpcAlarmConditionNode**. With the help of the **OpcAlarmConditionNode** it is possible to define Event Nodes whose behavior is comparable to a bedside timer. Accordingly, this Node 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 has been set with a wake-up time, but should not be active when it is reached, is called a suppressed alarm (see *IsSuppressed* and *IsSuppressedOrShelved*). But if an alarm becomes active, it can be shelved (see *IsSuppressedOrShelved* property). In this case, an alarm can be reset once (“One Shot Shelving”) or timed (“Timed Shelving”) (see *Shelving* property). Alternatively, a reset alarm can also be “unshelved” again (see *Shelving* property). An example of the **OpcAlarmConditionNode** API is shown in the following code:

```
var overheating = new OpcAlarmConditionNode(machineOne, "OverheatingAlert");
var idle = new OpcAlarmConditionNode(machineOne, "IdleAlert");

...

overheating.ChangeIsActive(this.SystemContext, true);
idle.ChangeIsActive(this.SystemContext, true);

...

if (overheating.IsActive)
    CancelJob();

if (!idle.IsActive)
    ProcessJob();
else if (idle.IsSuppressed)
    SimulateJob();
```

Event Nodes with discrete Alarm Conditions

The following types are used: OpcServer, OpcNodeManager, OpcDiscreteAlarmNode, OpcOffNormalAlarmNode and OpcTripAlarmNode.

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

Starting from the **OpcAlarmConditionNode**, there are several specializations that have been explicitly defined for specific types of alarms to specify the form, reason or content of an alarm more precisely 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 **OpcDiscreteAlarmNode** class. It defines an alarm Node 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 is to represent a discrete condition that is considered abnormal, consider using the **OpcOffNormalAlarmNode** or one of its subclasses. Based on this alarm class, the framework offers a further concretization with the **OpcTripAlarmNode**. The **OpcTripAlarmNode** becomes active when, for example, an abnormal fault occurs on a monitored device, e.g. when the motor is shut down due to overload. The aforementioned Nodes are created as follows:

```
var x = new OpcDiscreteAlarmNode(machineOne, "discreteAlert");
var y = new OpcOffNormalAlarmNode(machineOne, "offNormalAlert");
var z = new OpcTripAlarmNode(machineOne, "tripAlert");
```

Event Nodes with Alarm Conditions for Limits

The following types are used: OpcServer, OpcNodeManager and OpcLimitAlarmNode.

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

If process-specific limit values are to be checked and the output of the test is to be published in the case of limit value overruns / underruns, the **OpcLimitAlarmNode** class provides the central starting point for entering the classes of limit alarms. With this class limits can be 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, it is not necessary to define all limits. For this reason, the class offers the possibility to set the desired limits from the beginning:

```
var positionLimit = new OpcLimitAlarmNode(
        machineOne, "PositionLimit", OpcLimitAlarmStates.HighHigh |
OpcLimitAlarmStates.LowLow);

positionLimit.HighHighLimit = 120; // e.g. mm
positionLimit.LowLowLimit = ;     // e.g. mm
```

Event Nodes with Alarm Conditions for exclusive Limits

The following types are used: OpcServer, OpcNodeManager and OpcExclusiveLimitAlarmNode.

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

A subclass of the **OpcLimitAlarmNode** is the class **OpcExclusiveLimitAlarmNode**. 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. The thereby violated boundary is described with the *Limit* property of the Node.

In the OPC UA, there are three further specializations of the **OpcExclusiveLimitAlarmNode**.

OpcExclusiveDeviationAlarmNode

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

OpcExclusiveLevelAlarmNode

This type of alarm is typically used to report 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.

OpcExclusiveRateOfChangeAlarmNode

This type of alarm is commonly used to report an unusual change or absence of a measured value in relation 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: OpcServer, OpcNodeManager and OpcNonExclusiveLimitAlarmNode.

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

A subclass of the **OpcLimitAlarmNode** is the class **OpcNonExclusiveLimitAlarmNode**. As its name suggests, it serves to define limit alerts for non-exclusive boundaries. Such a limit alarm uses values for the limits that do **not exclude each other**. 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 Node.

As part of the OPC UA, there are three further specializations of the **OpcNonExclusiveLimitAlarmNode**.

OpcNonExclusiveDeviationAlarmNode

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

OpcNonExclusiveLevelAlarmNode

This type of alarm is typically used to report 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.

OpcNonExclusiveRateOfChangeAlarmNode

This type of alarm is commonly used to report an unusual change or absence of a measured value in relation 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.

Monitoring Request and Response Messages

The following types are used: OpcServer, OpcRequestValidatingEventArgs, OpcRequestValidatingEventHandler, OpcRequestProcessingEventArgs, OpcRequestProcessingEventHandler, OpcRequestProcessedEventArgs, OpcRequestProcessedEventHandler, OpcRequestValidatedEventArgs, OpcRequestValidatedEventHandler, IOpcServiceRequest and IOpcServiceResponse.

The requests sent by Clients to a Server are processed by the Server as instances of the IOpcServiceRequest interface, validated and answered by instances of the IOpcServiceResponse interface. The requests received by the Server can be additionally monitored, logged, routed or denied via the events RequestProcessing, RequestValidating, RequestValidated and RequestProcessed via user-defined methods. This is particularly useful in situations when the mechanisms provided by the framework are not sufficient for the project-specific requirements, in particular for certain restrictions. The procedure of processing up to the answer of inquiries runs through the following steps:

1. Receiving the raw data of a Request (Protocol Level of the framework)
2. Deserialization of the raw data for a Request (Message Level of the framework)
3. Preprocessing request: **RequestProcessing event**
4. Delegation of the request to the corresponding service (Service Level of the framework)
5. Validation of the Request
    1. Default validations (Session, Identity, ...)
    2. userdefined validation: **RequestValidating event**
    3. final validation (check of custom validation)
    4. custom completion of validation: **RequestValidated event**
6. Processing the request (Application Level of the framework)
7. Generating the answer via the corresponding service (Service Level of the framework)
8. Post processing of the Request and its Response: **RequestProcessed event**
9. Serialization of the Response to raw data (Message Level of the framework)
10. Sending the raw data of the Response (Protocol Level of the framework)

The events mentioned under points 3, 5.2, 5.4 and 8. offer the developer of the Server the opportunity to monitor or influence the processing of Requests via user-defined code. The user-defined code of the RequestProcessing event is executed immediately after receipt and processing of the user data in the form of an IOpcServiceRequest instance. The information provided here is used for the primary diagnosis of message traffic between Client and Server. An event handler registered here should not throw an exception:

```
private static void HandleRequestProcessing(object sender, OpcRequestProcessingEventArgs e)
{
    Console.Write("Processing: " + e.Request.ToString());
}

// ...

server.RequestProcessing += HandleRequestProcessing;
```

The context provided in the OpcRequestProcessingEventArgs always corresponds to an instance of the OpcContext class, which describes only the general environment of message processing. The information provided in this event handler is also supplemented in the subsequent RequestValidating event with information about the Session and Identity. In the case of Requests that require a Session, the OpcContext object provided is the specialization OpcOperationContext. The OpcOperationContext can be used to perform additional session-related validations:

```
private static nodesPerSession = new Dictionary<OpcNodeId, int>();

private static void HandleRequestValidating(object sender, OpcRequestValidatingEventArgs e)
{
    Console.Write(" -> Validating: " + e.Request.ToString());

    if (e.RequestType == OpcRequestType.AddNodes) {
        var sessionId = e.Context.SessionId;
        var request = (OpcAddNodesRequest)e.Request;

        lock (sender) {
            if (!nodesPerSession.TryGetValue(sessionId, out var count))
                nodesPerSession.Add(sessionId, count);

            count += request.Commands.Count;
            nodesPerSession[sessionId] = count;

            e.Cancel = (count >= 100);
        }
    }
}

// ...

server.RequestValidating += HandleRequestValidating;
```

The example shows how to limit the number of “AddNodes” requests per session to 100 “AddNode” commands. Any further Rrequest will be refused once the restriction has been reached. This is done by setting the Cancel property of the event's arguments to “true”, which automatically sets the Result property code of the event's arguments to the “BadNotSupported” value. It is also possible to cancel the Request by (additional) setting a “Bad” code. An event handler registered at the RequestValidating event may throw an exception. However, if an exception is raised in the event handler or the Cancel property is set to “true” or the Result property of the event's arguments is set to a “Bad” code, then the event handlers of the RequestValidated event are not executed (which is the .NET Framework known Validating-Validated-Pattern). If, on the other hand, the Request is not aborted, the event handlers of the RequestValidated event are executed:

```
void HandleRequestValidated(object sender, OpcRequestValidatedEventArgs e)
{
    Console.Write(" -> Validated");
}

// ...

server.RequestValidated += HandleRequestValidated;
```

Again, as with the RequestProcessing event, the information provided serves as the primary diagnostic of message traffic between Client and Server. The benefit of the event is that only after calling the event the Server also tries to process and answer the Request. An event handler registered here should not throw an exception. After completing the query processing performed by the Server, the Request is finally answered with the resulting results. The resulting Response can be evaluated together with the Request in the RequestProcessed event:

```
private static void HandleRequestProcessed(object sender, OpcRequestProcessedEventArgs e)
{
    if (e.Response.Success)
        Console.WriteLine(" -> Processed!");
    else
        Console.WriteLine(" -> FAILED: {0}!", e.Exception?.Message ??
e.Response.ToString());
}

// ...

server.RequestProcessed += HandleRequestProcessed;
```

As shown in the above example, the arguments of the event additionally provides information about an exception that may have occurred during the processing. An event handler registered here should not throw an exception.

Server Configuration

General Configuration

The following types are used here: OpcServer, OpcCertificateStores and OpcCertificateStoreInfo.

In all code snippets depicted the Server is always configured via the code (if the default configuration of the Server is not used). The **OpcServer** instance is the central port for the configuration of the Server application. All settings concerning security can be found as an instance of the **OpcServerSecurity** class via the *Security* property of the Server. All settings concerning the Certificate Store can be found as an instance of the **OpcCertificateStores** class via the *CertificateStores* property of the Server.

If the Server shall be configurable via XML you can load the configuration of the Server either from a selected or a random XML file. Instructions are provided in the section “Preparations of Server Configuration via XML”.

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

- Loading the configuration file via App.config

```
server.Configuration =
OpcApplicationConfiguration.LoadServerConfig("Opc.UaFx.Server");
```

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

```
server.Configuration =
OpcApplicationConfiguration.LoadServerConfigFile("MyServerAppNameConfig.xml");
```

For configuring the Server application amongst others are the following options:

- Configurating the application
  - via Code:

```
// Default: Value of AssemblyTitleAttribute of entry assembly.
server.ApplicationName = "MyServerAppName";

// Default: A null reference to auto complete on start to "urn::" +
ApplicationName
server.ApplicationUri = "http://my.serverapp.uri/";
```

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

```
<ApplicationName>MyServerAppName</ApplicationName>
<ApplicationUri>http://my.serverapp.uri/</ApplicationUri>
```

- Configuring the Certificate Store
  - via Code:

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

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

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

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

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

```
  <SecurityConfiguration>
    <ApplicationCertificate>
      <StoreType>Directory</StoreType>
      <StorePath>%LocalApplicationData%\MyServerAppName\CertificateStores\App
Certificates</StorePath>
      <SubjectName>MyServerAppName</SubjectName>
    </ApplicationCertificate>

    <RejectedCertificateStore>
      <StoreType>Directory</StoreType>
<StorePath>%LocalApplicationData%\MyServerAppName\CertificateStores\Rejected
Certificates</StorePath>
    </RejectedCertificateStore>

    <TrustedIssuerCertificates>
      <StoreType>Directory</StoreType>
      <StorePath>%LocalApplicationData%\MyServerAppName\CertificateStores\Trusted
Issuer Certificates</StorePath>
    </TrustedIssuerCertificates>

    <TrustedPeerCertificates>
      <StoreType>Directory</StoreType>
      <StorePath>%LocalApplicationData%\MyServerAppName\CertificateStores\Trusted
Peer Certificates</StorePath>
    </TrustedPeerCertificates>
  </SecurityConfiguration>
```

Certificate Configuration

The following types are used here: OpcServer, OpcCertificateManager, OpcServerSecurity, OpcCertificateStores and OpcCertificateStoreInfo.

Recommended are certificates of types .der, .pem, **.pfx** and **.p12**. If the Server shall provide a secure endpoint (in which 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("MyServerCertificate.pfx");
   ```
2. A **new certificate** is generated (in storage):
   ```
   var certificate = OpcCertificateManager.CreateCertificate(server);
   ```
3. Save a certificate in any path:
   ```
   OpcCertificateManager.SaveCertificate("MyServerCertificate.pfx", certificate);
   ```
4. Set the Server certificate:
   ```
   server.Certificate = certificate;
   ```
5. The certificate has to be stored in the **Application Store**:
   ```
   if (!server.CertificateStores.ApplicationStore.Contains(certificate))
       server.CertificateStores.ApplicationStore.Add(certificate);
   ```
6. If **no or an invalid certificate** is used, a new certificate is generated / used by default. If the Server shall only use the mentioned certificate this function has to be deactivated. For

deactivating the function set the property **AutoCreateCertificate** to the value *false*:

```
server.CertificateStores.AutoCreateCertificate = false;
```

User Identity Configuration

The following types are used here: OpcServer, OpcUserIdentity, OpcServerIdentity, OpcCertificateIdentity, OpcServerSecurity, OpcAccessControlList, OpcAnonymousAcl, OpcUserNameAcl, OpcCertificateAcl, OpcAccessControlEntry, OpcOperationType, OpcRequestType and OpcAccessControlMode.

By default a Server allows access without a concrete user identity. This kind of authentication is called anonymous authentication. When a user identity is mentioned it has to be known to the Server in order to access the Server with this identity. For example, if a username-password pair or a certificate shall be used for user identification, the according ACLs (Access Control Lists) have to be configured and activated. Part of the configuration of control lists is the configuration of ACEs (Access Control Entries). Those are defined by a principal with a certain identity (username-password pair or certificate) and registered in a list.

- Deactivating the anonymous ACL:

```
server.Security.AnonymousAcl.IsEnabled = false;
```

- Configuring the **username-password pair**-based ACL:

```
var acl = server.Security.UserNameAcl;

acl.AddEntry("username1", "password1");
acl.AddEntry("username2", "password2");
acl.AddEntry("username3", "password3");
...
acl.IsEnabled = true;
```

- Configuring the **certificate**-based ACL:

```
var acl = server.Security.CertificateAcl;

acl.AddEntry(new X509Certificate2(@".\user1.pfx"));
acl.AddEntry(new X509Certificate2(@".\user2.pfx"));
acl.AddEntry(new X509Certificate2(@".\user3.pfx"));
...
acl.IsEnabled = true;
```

All Access Control Lists defined by the Framework up until now use the mode “Whitelist” as Access Control Mode. In this mode every entry has - only by defining an Access Control Entry - access to all Types of Requests, even if the access was not explicitly allowed to the entry. Therefore all non-allowed actions have to be denied to the entries. Allowed and denied operations can be set directly on the entry which is available after the note in the ACL.

1. Remember an Access Control Entry:

```
var user1 = acl.AddEntry("username1", "password1");
```

2. Deny the Access Control Entry two rights:

```
user1.Deny(OpcRequestType.Write);
user1.Deny(OpcRequestType.HistoryUpdate);
```

3. Allow a previously denied right:

```
user1.Allow(OpcRequestType.HistoryUpdate);
```

Server Endpoint Configuration

The following types are used here: OpcServer, OpcServerSecurity, OpcSecurityPolicy, OpcSecurityMode and OpcSecurityAlgorithm.

Endpoints of a Server are defined through the cross product of used Base-Addresses and configured security strategies for endpoints. The Base-Addresses consist of supported scheme-port pairs and the host (IP address or DNS name), where several schemes (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”) can be set for data exchange on different ports. By default the Server does not use a special policy to supply a secure endpoint. Therefore there are as many endpoints as there are Base-Addresses. If a Server defines exactly one Base-Address there is only one endpoint with this Base-Address and the security policy with the mode *None*. If there are n different Base-Addresses there are n different endpoints with exactly the same security policy, even if only one special security policy is set. But if there are m different security policies (s_1, s_2, s_3, …, s_m), n different Base-Addresses (b_1, b_2, …, b_n) create the endpoints that are created by a pairing of policy and Base-Address (s_1+b_1, s_1+b_2, …, s_1+b_n, s_2+b_1, s_2+b_2, …, s_2+b_n, s_3+b_1, s_3+b_2, …, s_3+b_n, s_m+b_n, …).

Additional to the Security-Mode of the protection of communication to be used, an Endpoint-Policy defines a Security-Algorithm and a level. According to the OPC Foundation the level of policy of an endpoint exists as a relative measure for security policies used for the endpoint. An endpoint with a higher level is defined more secure as an endpoint with a lower level (note that this is merely a neither watched nor imposed guideline).

If two Security-Policies are followed, they could be defined like this:

- Security-Policy A: Level=0, Security-Mode=None, Security-Algorithm=None
- Security-Policy B: Level=1, Security-Mode=Sign, Security-Algorithm=Basic256

If furthermore three Base-Addresses are set for different schemes:

- Base-Address A: "https://mydomain.com/"
- Base-Address B: "opc.tcp://192.168.0.123:4840/"
- Base-Address C: "opc.tcp://192.168.0.123:12345/"

The result of the cross product will be these endpoint descriptions:

- Endpoint 1: Address="https://mydomain.com/", Level=0, Security-Mode=None, Security-Algorithm=None
- Endpoint 2: Address="https://mydomain.com/", Level=1, Security-Mode=Sign, Security-Algorithm=Basic256
- Endpoint 3: Address="opc.tcp://192.168.0.123:4840/", Level=0, Security-Mode=None, Security-Algorithm=None
- Endpoint 4: Address="opc.tcp://192.168.0.123:4840/", Level=1, Security-Mode=Sign, Security-Algorithm=Basic256
- Endpoint 5: Address="opc.tcp://192.168.0.123:12345/", Level=0, Security-Mode=None, Security-Algorithm=None
- Endpoint 6: Address="opc.tcp://192.168.0.123:12345/", Level=1, Security-Mode=Sign, Security-Algorithm=Basic256

For configuring the (primary) Base-Address either the constructor of the **OpcServer** Class or the **Address** property of an **OpcServer** instance can be used:

```
var server = new OpcServer("opc.tcp://localhost:4840/");
server.Address = new Uri("opc.tcp://localhost:4840/");
```

If the Server shall support further Base-Addresses these can be administrated through the methods **RegisterAddress** and **UnregisterAddress**. All of those Base-Addresses used (therefore registered) by the Server can be called via the **Addresses** property. If the value of the **Address** property was not set primarily the first address defined through **RegisterAddress** will be used for the **Address** property.

Define two more Base-Addresses:

```
server.RegisterAddress("https://mydomain.com/");
server.RegisterAddress("net.tcp://192.168.0.123:12345/");
```

Unregister two Base-Addresses from the Server in order for the “main” Base-Address to change:

```
server.UnregisterAddress("https://mydomain.com/");

// server.Address becomes: "net.tcp://192.168.0.123:12345/"
server.UnregisterAddress("opc.tcp://localhost:4840/");
```

If all addresses of the **Addresses** property are unregistered the value of the **Address** property is not set.

Definition of a secure security policy for endpoints of the Server:

```
server.Security.EndpointPolicies.Add(new OpcSecurityPolicy(
        OpcSecurityMode.Sign, OpcSecurityAlgorithm.Basic256, 3));
```

By defining a concrete security policy for endpoints the default policy with the mode *None* is lost. In order for this policy (not recommended for the productive use) to be supported by the Server it has to be registered explicitly in the Endpoint-Policy list:

```
server.Security.EndpointPolicies.Add(new OpcSecurityPolicy(
        OpcSecurityMode.None, OpcSecurityAlgorithm.None, ));
```

Further Security Settings

The following types are used here: OpcServer, OpcServerSecurity, OpcCertificateValidationFailedEventArgs, OpcCertificateStores und OpcCertificateStoreInfo.

A Client sends its certificate to the Server for authentification during the connecting. The Server can decide if to approve a connection and trust or untrust a Client using the certificate.

- If the Server shall accept **only trusted** certificates the default acceptance of all certificates must be deactivated as follows:

  ```
  server.Security.AutoAcceptUntrustedCertificates = false;
  ```

- As soon as the default acceptance of all certificates has been deactivated a custom check of certificates is necessary:

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

- If the Client certificate is judged as **untrusted** it can be declared **trusted** manually by saving it in the TrustedPeerStore:

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

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

Configuration via XML

If the Server shall also be configurable via XML the Server configuration can be loaded either from a specific or 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">
</OpcApplicationConfiguration>
```

If a random XML file shall be used for configuration a .config file (referring to an XML file from which the configuration for the Server shall be loaded) has to be created. This section shows which entries the .config file has to have and how the XML file must be structured.

Compiling and preparing the App.config of the application:

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

```
<configSections>
  <section name="Opc.UaFx.Server"
           type="Opc.Ua.ApplicationConfigurationSection,
                 Opc.UaFx.Advanced,
                 Version=2.0.0.0,
                 Culture=neutral,
                 PublicKeyToken=0220af0d33d50236" />
</configSections>
```

3. Also insert this *Opc.UaFx.Server* element underneath the *configuration* elements:

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

4. The value of the *FilePath* element can show to a random data path where you will find the XML

configuration file needed. The value shown here would show to a configuration file lying next to the application.
5. Save the changes to App.config

Creating and preparing the XML configuration file:

1. Create an XML file with the name used in the App.config and save under the path used in App.config.
2. Insert this 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">
</OpcApplicationConfiguration>
```

3. Save changes to XML file

Server Application Delivery

This is how you prepare your OPC UA Server application for the use in productive environment.

Application Certificate - *Using a concrete certificate*
Don't use an automatically Framework-generated certificate in productive use.

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 Server instance via the **OpcCertificateManager**:

```
var certificate = OpcCertificateManager.LoadCertificate("MyServerCertificate.pfx");
server.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 Server 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:

```
server.ApplicationName = "<Common Name (CN) in Certificate>";
server.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 Server 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 MyServerAppName
```

The first parameter (-sp) sets saving the certificate in the current list. The second parameter (-an) sets the name of the Server application using the application certificate. Replace “MyServerAppName” by the name of your Server application. Note that **the Framework for chosing the application certificate uses the value of the *AssemblyTitle* attribute and therefore the same value as stated in this attribute is used for “MyServerAppName”**. 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 Server instance:

```
server.ApplicationName = "MyDifferentServerAppName";
```

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 Server 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 list. Without changing the names of the folders and the files, copy both folders in the list 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 Server instance:

```
server.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 list as the application and hold the name of the application as a prefix:

```
<MyServerAppName>.exe.config
```

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

System Configuration - *Administrative Setup*

Execute the application in the target system once only with administrative rights to ensure that the Server has permission to access the network resources. This is necessary, if e.g. the Server shall use a Base-Address with the scheme “http” or “https”.

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 Server development** it has to be committed to the framework. Hereto insert the following code line into your application **before** accessing the **OpcServer class** for the first time. Replace *<insert your license code here>* with the license key you

received from us.

```
Opc.UaFx.Server.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.Server.Licenser class** for Server licenses and via the **Opc.UaFx.Licenser class** for bundle licenses. This works as follows:

```
ILicenseInfo license = Opc.UaFx.Server.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 Server 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 Server application and at least executing it when the application is started:

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

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

Table of Contents

Tested? You want it? 1
The Server Frame 2
Node Management 2
Node Creation 2
Node Accessibility 3
Node Updates 4
Values of Node(s) 4
Reading Values 4
Writing Values 5
Historical Data 6
Nodes 15
Method Nodes 15
File Nodes 17
Datatype Nodes 18
Data Nodes 19
Data-Item Nodes 19
Data-Item Nodes for analog Values 20
NodeSets 20
Import NodeSets 21
Implement NodeSets 21
Events 22
Event Reporting 22
Event Nodes 23
Event Nodes with Conditions 25
Event Nodes with Dialog Conditions 26
Event Nodes with Feedback Conditions 26
Event Nodes with Alarm Conditions 27
Event Nodes with discrete Alarm Conditions 28
Event Nodes with Alarm Conditions for Limits 28
Event Nodes with Alarm Conditions for exclusive Limits 29
Event Nodes with Alarm Conditions for non-exclusive Limits 29
Monitoring Request and Response Messages 30
Server Configuration 32
General Configuration 32
Certificate Configuration 34
User Identity Configuration 35
Server Endpoint Configuration 36
Further Security Settings 37
Configuration via XML 38
Server Application Delivery 39
Licensing 40