OPC UA SDK
Client + Server Development

Client Development Guide & Tutorial

Getestet? Du willst es?

Lizenzmodell Preise Angebot Jetzt bestellen

C# ▼ |VB|

Der Client Frame

Ein Client für OPC UA

1. Verweis zum **Opc.UaFx.Advanced** Client Namespace hinzufügen:
```
using Opc.UaFx.Client;
```
2. Eine Instanz der OpcClient Klasse mit der Adresse des Servers erzeugen:
```
var client = new OpcClient("opc.tcp://localhost:4840/");
```
3. Verbindung zum Server aufbauen und Sitzung starten:
```
client.Connect();
```
4. Ihr Code zur Interaktion mit dem Server:
```
// Your code to interact with the server.
```
5. Vor dem Beenden der Anwendung die Verbindung wieder trennen:
```
client.Disconnect();
```
6. Unter Verwendung des using Blocks sieht das dann so aus:
```
using (var client = new OpcClient("opc.tcp://localhost:4840/")) {
    client.Connect();
    // Your code to interact with the server.
}
```
7. Sind die Nodes/NodeIds des Servers...
    1. **unbekannt:** Hier finden Sie heraus, welche Nodes der Server bereitstellt
    2. **bekannt:** Node Werte lesen oder Node Werte schreiben

Ein Client für OPC Classic

1. Verweis zum **Opc.UaFx.Advanced** Classic und Client Namespace hinzufügen:
```
using Opc.UaFx.Client;
using Opc.UaFx.Client.Classic;
```
2. Eine Instanz der OpcClient Klasse mit der Adresse des Servers erzeugen:
```
var client = new OpcClient("opc.com://localhost:48410/<progId>/<classId>");
```
3. Verbindung zum Server aufbauen und Sitzung starten:
```
client.Connect();
```
4. Ihr Code zur Interaktion mit dem Server:
```
// Your code to interact with the server.
```
5. Vor dem Beenden der Anwendung die Verbindung wieder trennen:

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

6. Unter Verwendung des using Blocks sieht das dann so aus:

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

7. Sind die Nodes/NodeIds des Servers...
    1. **unbekannt:** Hier finden Sie heraus, welche Nodes der Server bereitstellt
    2. **bekannt:** Node Werte lesen oder Node Werte schreiben

Was beschreibt die Adresse des Servers opc.com://localhost:48410/<progId>/<classId>**?**

- **opc.com** gibt an, dass eine Verbindung zu einem *OPC Classic Server* hergestellt werden soll.
- **localhost** steht für den Namen oder die IP Adresse des Rechners, auf dem der *OPC Classic Server* ausgeführt wird.
- **48410** die optionale Portnummer die der *OPC UA Wrapper Server*[1)] verwenden soll. Fehlt diese wird eine Portnummer anhand der <classId> generiert.
- **<progId>** ist ein Platzhalter, ersetzen Sie diesen durch die ProgID der **DCOM-Anwendungs-Informationen** des *OPC Classic Servers*,
  z.B. 'OPCManager.DA.XML-DA.Server.DA'.
- **<classId>** ist ein Platzhalter, ersetzen Sie diesen durch die ClassId (auch CLSID oder AppID) der **DCOM-Anwendungs-Informationen** des *OPC Classic Servers*,
  z.B. '{E4EBF7FA-CCAC-4125-A611-EAC4981C00EA}'.

Wie wird die <classId> beziehungsweise <progId> des Servers ermittelt?

1. **Methode: 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. **Methode: Systeminformationen**
   Führen Sie folgende Schritte auf dem Rechner des *OPC Classic Servers* durch:
    1. *AppID* über die Komponentendienste ermitteln
        - Systemsteuerung
        - Verwaltung
        - Komponentendienste
        - 'Komponentendienste' erweitern
        - 'Computer' erweitern
        - 'Arbeitsplatz' erweitern
        - 'DCOM-Konfiguration' erweitern
        - '<OPC Classic Server>' auswählen
        - 'Eigenschaften' öffnen

- erster Reiter 'Allgemein'
- 'Anwendungs-ID:' (= *AppID*) kopieren

2. *ProgId* über die Systemregistrierungsdatenbank ermitteln
   - 'Windows-Taste' + 'R'
   - 'regedit' eingeben
   - 'Ausführen' klicken
   - den Schlüssel 'Computer' erweitern
   - dann den Schlüssel 'HKEY_LOCAL_MACHINE'
   - 'SOFTWARE' erweitern
   - 'SYSWOW6432Node' erweitern (nur auf 64 Bit Systemen ansonsten weiter)
   - 'Classes' erweitern
   - 'CLSID' erweitern
   - '<classId>' erweitern
   - 'ProgID' auswählen
   - '(Standard)' Wert (= *ProgID*) in der 'Daten'-Spalte kopieren

Werte von Node(s)

Werte lesen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcValue, OpcAttribute und OpcReadNode.

Welcher Node gelesen werden soll, wird durch den **OpcNodeId** des Nodes festgelegt. Wird ein Node Wert gelesen, dann wird standardmäßig immer der aktuelle Wert des *Value Attributs* gelesen. Der dabei ermittelte **OpcValue** besteht neben dem eigentlichen Wert aus einem Zeitstempel, zu dem der Wert an der Quelle des Wertes festgestellt worden ist (**SourceTimestamp**), sowie aus einem weiteren Zeitstempel, zu dem der Wert vom Server erfasst wurde (**ServerTimestamp**). Soll ein anderes Attribut des Nodes gelesen werden, muss das entsprechende **OpcAttribute** beim Aufruf von **ReadNode** beziehungsweise bei der jeweiligen **OpcReadNode** Instanz angegeben werden.

- Den Wert des Value Attributs eines einzelnen Nodes lesen:

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

- Die Werte des Value Attributs mehrerer Nodes lesen:

```
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);
```

- Den Wert des DisplayName Attributs eines einzelnen Nodes lesen:

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

- Die Werte des DisplayName Attributs mehrerer Nodes lesen:

```
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);
```

Werte schreiben

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcValue, OpcAttribute, OpcStatus, OpcStatusCollection und OpcWriteNode.

Welcher Node geschrieben werden soll, wird durch den **OpcNodeId** des Nodes festgelegt. Wird ein Node Wert geschrieben, dann wird standardmäßig immer der aktuelle Wert des *Value Attributs* geschrieben. Der dabei festgelegte **OpcValue** erhält automatisch als Zeitstempel der Quelle (**SourceTimestamp**) den aktuellen Zeitstempel. Soll ein anderes Attribut des Nodes geschrieben werden, muss das entsprechende **OpcAttribute** beim Aufruf von **WriteNode** beziehungsweise bei der jeweiligen **OpcWriteNode** Instanz angegeben werden.

- Den Wert eines einzelnen Nodes schreiben:

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

- Die Werte mehrerer Nodes schreiben:

```
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);
```

- Den Wert des DisplayName Attributs eines einzelnen Nodes schreiben:

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

- Die Werte des DisplayName Attributs mehrerer Nodes schreiben:

```
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);
```

Werte verarbeiten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcValue, OpcStatus und OpcStatusCollection.

Die **ReadNode** Methoden liefern immer eine **OpcValue** Instanz, während die **ReadNodes** Methoden eine Auflistung von **OpcValue** Instanzen liefern (je gelesenem Node einen **OpcValue**). Der eigentliche Wert, der gelesen wurde, steht dabei in der *Value* Eigenschaft der **OpcValue** Instanzen. Das Ergebnis der Leseanforderung kann über die *Status* Eigenschaft überprüft werden. Der Zeitstempel, zu dem der gelesene Wert an der Quelle festgestellt wurde, kann über die *SourceTimestamp* Eigenschaft abgerufen werden. Dementsprechend kann der Zeitstempel, zu dem der gelesene Wert vom Server erfasst wurde, über die *ServerTimestamp* Eigenschaft abgerufen werden.

- Den Wert eines einzelnen Nodes lesen:
```
OpcValue value = client.ReadNode("ns=2;s=Machine/Job/Speed");
```
- Das Ergebnis der Leseanforderung prüfen:
```
if (value.Status.IsGood) {
    // Your code to operate on the value.
}
```
- Den skalaren Wert der **OpcValue** Instanz abrufen:
```
int intValue = (int)value.Value;
```
- Den Arraywert der **OpcValue** Instanz abrufen:
```
int[] intValues = (int[])value.Value;
```

Die **WriteNode** Methoden liefern immer eine **OpcStatus** Instanz, während die **WriteNodes** Methoden eine **OpcStatusCollection** Instanz liefern (die für jeden geschriebenen Node einen **OpcStatus** enthält). Das Ergebnis der Schreibeanforderung kann somit über die Eigenschaften der **OpcStatus** Instanz(en) überprüft werden.

- Den skalaren Wert eines einzelnen Nodes schreiben:
```
OpcStatus status = client.WriteNode("ns=2;s=Machine/Job/Speed", 1200);
```
- Den Arraywert eines einzelnen Nodes schreiben:
```
int[] values = new int[3] { 1200, 1350, 1780 };
OpcStatus status = client.WriteNode("ns=2;s=Machine/Job/Speeds", values);
```
- Das Ergebnis der Schreibeanforderung prüfen:
```
if (!status.IsGood) {
    // Your code to handle a failed write operation.
}
```

Unter Anwendung der einzelnen Schritte zur Verarbeitung von skalaren Werten und Arraywerten kann der Arraywert eines Variablen-Nodes wie folgt modifiziert werden:

```
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!");
    }
}
```

Browsen von Nodes

Welche Nodes hat der Server?

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcNodeInfo, OpcAttribute, OpcAttributeInfo und OpcObjectTypes.

Ausgehend von einem Server dessen Address-Space (alle verfügbaren Nodes) noch (nahezu) unbekannt sind, empfiehlt es sich die bereitgestellten Nodes des Servers zu inspizieren. Hierbei kann entweder ein grafischer OPC UA Client wie die

OPC Watch

verwendet werden oder auch manuell der Address-Space des Servers untersucht werden.

Unter Einsatz der Klasse **OpcObjectTypes** können bereits vordefinierte Server-Nodes mittels Browsing untersucht werden. Der quasi „Root“ (also die Wurzel) aller Knoten des Servers repräsentiert den Standard-Node mit der Bezeichnung „ObjectsFolder“. Wird somit beim „ObjectsFolder“-Node mit dem Browsing begonnen, dann kann auf diese Weise der gesamte Address-Space des Servers ermittelt werden:

```
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);
}
```

Die über den gezeigten Codeausschnitt ausgegegeben Informationen enthalten unter anderen (in runden Klammern) die Node-Ids der Nodes des Servers. Basierend auf diesen NodeId's können die Nodes des Servers direkt angesprochen werden. Hierzu wird einfach die NodeId als String (in doppelten Anführungsstrichen) an die entsprechende Methode der **OpcClient**-Klasse übergeben. Wie das zum Beispiel beim Lesen eines Node-Wertes aussieht, wird im Abschnitt Werte lesen gezeigt.

Node für Node 'besuchen'

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcNodeInfo, OpcAttribute, OpcAttributeInfo, OpcMethodNodeInfo, OpcArgumentInfo und OpcObjectTypes.

Das Browsen in der OPC UA kann mit .NET Reflections verglichen werden. Mittels Browsing ist es somit möglich, den gesamten „Address Space“ eines Servers dynamisch zu ermitteln und zu untersuchen. Dazu gehören unter anderem alle Nodes, deren Referenzen zueinander sowie deren Node Typen. Eingeleitet wird das Browsing über den **OpcNodeId** des Nodes, bei dem der Browse-Vorgang gestartet werden soll. Ausgehend von der dabei erhaltenen **OpcNodeInfo** kann das Browsen auf Child-, Parent- oder Attribut-Ebene fortgesetzt werden.

Handelt es sich bei einem Node um einen Node, der eine Methode darstellt, dann liefert das Browsing eine **OpcMethodNodeInfo** Instanz. Über diese Instanz können dann zusätzlich die Input- / Output-Argumente der Methode untersucht werden.

1. Die OpcNodeInfo des gewünschten Nodes ermitteln:
```
OpcNodeInfo machineNode = client.BrowseNode("ns=2;s=Machine");
```
2. Zum Browsen der Kind-Nodes die **Child** oder **Children** Methode verwenden:
```
OpcNodeInfo jobNode = machineNode.Child("Job");

foreach (var childNode in machineNode.Children()) {
    // Your code to operate on each child node.
}
```
3. Zum Browsen der Attribute die **Attribute** oder **Attributes** Methode verwenden:
```
OpcAttributeInfo displayName = machineNode.Attribute(OpcAttribute.DisplayName);

foreach (var attribute in machineNode.Attributes()) {
    // Your code to operate on each attribute.
}
```
4. Im Falle eines Nodes, der eine Methode darstellt, kann diese wie folgt untersucht werden:
```
if (childNode.Category == OpcNodeCategory.Method) {
    var methodNode = (OpcMethodNodeInfo)childNode;

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

High-Speed Browsen

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeInfo OpcNodeId, OpcBrowseNode,

OpcReferenceType, OpcBrowseNodeDegree und OpcBrowseOptions.

Stellt ein Server einen sehr umfangreichen „Address Space“ bereit ist das Browsen aller Nodes häufig etwas langsam insbesondere dann, wenn die vielen Node-Informationen auch noch über eine rechte schwache Hardware- oder Netzwerkkonfiguration transportiert werden müssen. Die zusätzliche Last auf dem Server führt auch hier zu Leistungseinbussen, wenn Clients den gesamten Node-Baum Node für Node durchlaufen.

Zur Optimierung dieser Situation bietet das SDK die Möglichkeit festzulegen, wie viele Nodes und Node-Ebenen zugleich untersucht werden sollen. Hierzu dient die `OpcBrowseNodeDegree`-Enumeration. Abhängig von der Menge der zugleich gewünschten Nodes wird der entsprechende Wert der Enumeration über die Degree-Eigenschaft oder im Konstruktor der OpcBrowseNode-Klasse bestimmt. Zusätzlich besteht die Möglichkeit die Nutzdaten beim Browsen auf ein Minimum oder eben nur auf die wirklich relevanten Daten zu reduzieren. Hierzu bietet die `OpcBrowseOptions`-Enumeration diverse Möglichkeiten. Zu beachten ist, dass mindestens die `ReferenceTypeId` Teil der Node-Informationen sein muss. Möchte man weitere Node-Informationen, wie den DisplayName, erhalten, dann muss (bei Verwendung der Options-Eigenschaft) auch der DisplayName mit eingeschlossen werden.

Das Browsen kann über die folgenden Eigenschaften parametrisiert und somit weiter optimiert werden:

```
// 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...
}
```

Anzumerken ist auch, dass wenn Nodes, die nur in einer bestimmten Beziehung zueinander stehen (welche über ReferenceTypes ausgedrückt wird) untersucht werden sollen, kann auch hier das Browsen weiter optimiert werden. Die für das Browsen relevanten Node-References können ebenso als Parameter für die Browse-Operation verwendet werden (siehe hierzu den Codeausschnitt oben), damit auch nur die Nodes besucht werden, die einen der angegebenen ReferenceTypes verwenden.

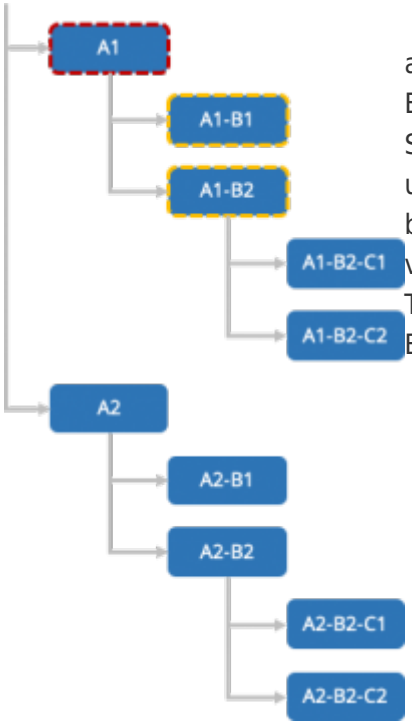
High-Speed Browsen - Details

Stellt ein Server einen sehr umfangreichen „Address Space“ bereit ist das **Browsen aller Nodes häufig etwas langsam** insbesondere dann, wenn die vielen Node-Informationen auch noch über eine rechte

schwache Hardware- oder Netzwerkkonfiguration transportiert werden müssen. Die zusätzliche Last auf dem Server führt auch hier zu Leistungseinbussen, wenn Clients den gesamten Node-Baum Node für Node durchlaufen.

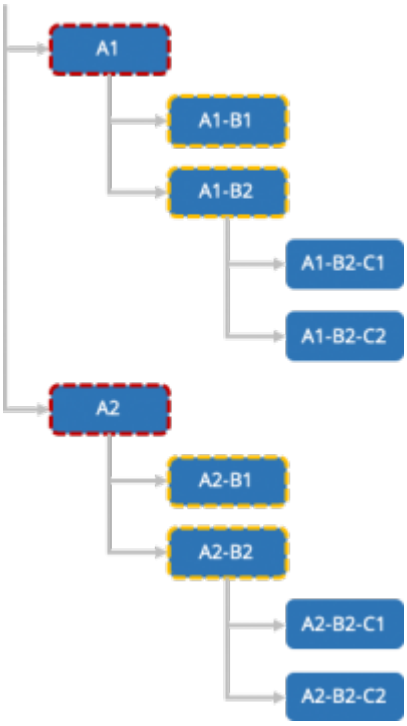
In solchen Fällen ist es wichtig den **Workload** für den Server und die gesamte **Kommunikation** zwischen Client und Server so **schlank** und **kurz wie möglich zu halten**. Hierzu bietet OPC UA diverse Mechanismen, mit denen ein Client den Server signalisieren kann, dass viel mehr Nodes als nur die einer Ebene untersucht werden. Der Server hat dann die Möglichkeit die Informationen bereits vorzubereiten und dem Client Paket-weise mitzuteilen. Das verringert somit Server-seitig die weitere interne Wiederverarbeitung des Node-Baums immer auf Anfrage des Clients. Zusätzlich kann ein Client, mit der entsprechenden Logik, weitere Nodes schon im Voraus vom Server abfragen, von denen bekannt ist, dass man deren Node-Informationen und die ihrer Kinder und deren Kindes-Kinder (und so weiter) auch noch verarbeiten möchte.

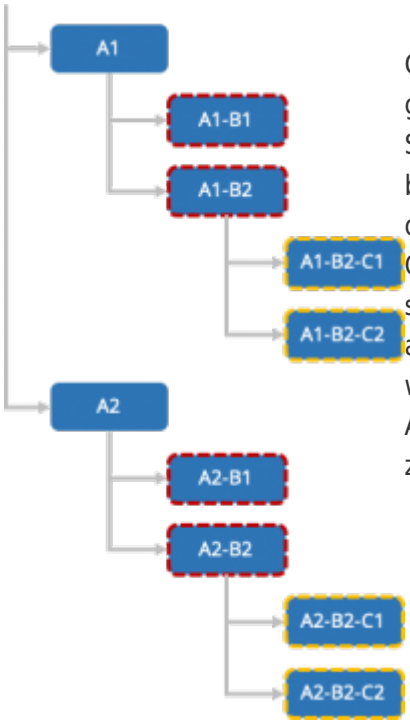
Das **SDK bietet zur Optimierung der Browse-Operationen diverse Parameter**, über die das Verhalten beim Browsen bestimmt werden kann. Abhängig vom Browse-Verlauf entscheidet das SDK, anhand der Parameter, welche Nodes mit welchen Informationen für den aktuellen Browse-Vorgang Server-seitig schon vorab vorbereitet werden sollen. Diese Nodes ruft dann das SDK automatisch beim weiteren Browsen ab, speichert diese schon vorab für den weiteren Browse-Vorgang im Speicher zwischen und liefert diese dann aus dem internen Cache anstatt den Server um diese Informationen zu bemühen.



Zur Steuerung, welche Nodes automatisch vom Server zugleich mit angefordert werden sollen, wird ein Wert der `OpcBrowseNodeDegree`-Enumeration verwendet. Der Wert `Self` (engl. selbst) entspricht dem Standardverhalten, bei dem immer nur die Kinder des aktuellen Nodes untersucht werden. Im Schema links wird das Browsen beim Node `A1` begonnen. Als Ergebnis liefert der Server nur die Nodes direkt unterhalb von Node `A1`, also die Nodes `A1-B1` und `A1-B2`. Die Nodes aller weiteren Teilbäume, auch die der Nodes neben dem Node `A1`, bei dem mit der Browse-Operation begonnen wurde, sind nicht Teil der Browse-Operation.

Mit dem Wert `Sibling` (engl. Geschwister) werden vom Server aller Kinder der Geschwister-Nodes zugleich mit abgerufen, wenn die Kind-Nodes des aktuellen Nodes untersucht werden. In diesem Fall ermittelt das SDK bei Start der Browse-Operation bei Node `A1` nicht nur die direkten Kind-Nodes von Node `A1`, sondern auch die aller Nodes die quasi Geschwister von `A1` sind. Somit werden bei der Browse-Operation auch die Teilbäume von `A2` bis `An` auch ermittelt. Die dabei erhaltenen Node-Informationen speichert das SDK zwischen und ruft diese aus dem Speicher anstatt vom Server (erneut) ab, sobald vom Entwickler der Teilbaum von z.B. Node `A2` benötigt wird. Im Bild rechts ist zu erkennen, dass eine Browse-Operation - ausgehend von Node `A1` – sich so verhält, als würde man die Nodes `A1` bis `An` gleichzeitig browsen.





Wird der Wert `Generation` verwendet, werden alle Nodes der gleichen Generation zugleich untersucht, was bedeutet, dass alle Nodes der gleichen Ebenentiefe untersucht werden. Konkret bedeutet das, wie in Schema links gezeigt, dass wenn eine Browse-Operation beim Node `A1-B1` begonnen wird, dass nicht nur dessen Teilbaum (wie beim Browsen mit dem `OpcBrowseNodeDegree.Self`), sondern auch die Teilbäume von seinen Geschwistern (wie beim Browsen mit dem `OpcBrowseNodeDegree.Sibling`) sowie die Teilbäume seiner Cousinen/Cousins n-ten Grades ebenfalls abgerufen werden. Auf diese Weise verhält sich das Browsen dann so, als würde man die Nodes `A1-B1`, `A1-B2`, `A2-B1` und `A2-B2` zugleich browsen. Auch hier werden die Node-Informationen der Teilbäume zwischengespeichert und dem Entwickler bei Bedarf bereitgestellt.

Zu beachten ist, dass die Verwendung der einzelnen `OpcBrowseNodeDegree`-Werte immer mit etwas Vorsicht verbunden ist. Stellt zum Beispiel ein Server einen relativ „flachen Address Space“ zur Verfügung, der jedoch sehr viele Nodes je Teilbaum enthält, dann kann die Option `Generation` zu verhältnismäßig langen Browse-Operationen führen, die dann wiederum auch sehr Speicher-intensiv ausfallen können. Gleiches gilt auch für die Option `Self`, insbesondere dann, wenn ein sehr viele kleine Teilbäume auf der gleichen Ebene besitzt, die wiederum auch viele weitere Kind-Nodes enthalten.

Die generelle **Empfehlung** ist deshalb, dass der Entwickler entweder die Komplexität des Servers schon vorab kennt und darauf basierend den Grad der Browse-Operationen wählt oder das bei unbekannten/wechselnden Servern bei der Wahl des Grades `Sibling` oder `Generation` die Node-Informationen über die Options-Eigenschaft reduziert werden. Etwaige weitere notwendige Node-Informationen kann dann der Entwickler bei Bedarf über eine gesonderte Browse-Operation abrufen.

Subscriptions

Subscriptions erstellen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcSubscribeDataChange, OpcSubscribeEvent, OpcSubscription, OpcMonitoredItem, OpcMonitoredItemCollection, OpcDataChangeReceivedEventHandler, OpcDataChangeReceivedEventArgs, OpcEventReceivedEventHandler, OpcEventReceivedEventArgs und OpcNotification.

Subscriptions in der OPC UA können mit Abonnements einer oder mehrerer Zeitschriften im Bundle verglichen werden. Anstelle von Zeitschriften werden jedoch Node Events (= in der OPC UA: Monitored Item) oder auch Änderungen der Node Values (= in der OPC UA: Monitored Item) abonniert. Je Subscription kann festgelegt werden, in welchem Intervall die Benachrichtigungen (**OpcNotification**-Instanzen) der Monitored Items veröffentlicht werden sollen (**OpcSubscription.PublishingInterval**). Welcher Node abonniert werden soll, wird durch den **OpcNodeId** des Nodes festgelegt. Standardmäßig wird das *Value-Attribut* überwacht (= engl. monitored). Soll ein anderes Attribut des Nodes überwacht werden, muss das entsprechende **OpcAttribute** beim Aufruf von **SubscribeDataChange** beziehungsweise bei der jeweiligen **OpcSubscribeDataChange**-Instanz angegeben werden.

- Benachrichtigungen über Änderungen des Node Values abonnieren:

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

- Benachrichtigungen über Änderungen mehrerer Node Values abonnieren:

```
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);
```

- Benachrichtigungen über Änderungen von Node Values behandeln:

```
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);
}
```

- Benachrichtigungen über Node-Ereignisse abonnieren:

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

- Benachrichtigungen über Node-Ereignisse mehrerer Nodes abonnieren:

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

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

- Benachrichtigungen über Node Events behandeln:

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

- Konfiguration der **OpcSubscription**:

```
subscription.PublishingInterval = 2000;

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

- Benachrichtigungen über Änderungen mehrerer Node Values über eine einzige Subscription abonnieren und benutzerdefinierte Werte der *Tag*-Eigenschaft des **OpcMonitoredItem**'s festlegen:

```
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 addd OpcMonitoredItems.
OpcSubscription subscription = client.SubscribeNodes();

for (int index = ; 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);
}
```

Subscriptions filtern

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcSubscribeDataChange, OpcDataChangeFilter, OpcDataChangeTrigger, OpcSubscription, OpcMonitoredItem, OpcMonitoredItemCollection, OpcDataChangeReceivedEventHandler, OpcDataChangeReceivedEventArgs und OpcNotification.

Grundsätzlich wird eine Client Anwendung über eine einmal erstellte Subscription nur über Änderungen des Status und des Values einer Node benachrichtigt. Damit aber ein Server entsprechend bestimmter Richtlinien den Client über Änderungen an der Node benachrichtigt, besteht die Möglichkeit den Auslöser (= engl. trigger) einer Benachrichtigung festzulegen. Hierzu wird der gewünschte Trigger über einen Wert der **OpcDataChangeTrigger** Enumeration bestimmt. Die folgenden Beispiele abonnieren auf diese Weise Benachrichtigungen über Änderungen des Status oder des Values oder des Timestamps einer Node.

- Benachrichtigungen über Änderungen des Node Values und Node Timestamps abonnieren:

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

- Benachrichtigungen über Änderungen mehrerer Node Values und Node Timestamps abonnieren:

```
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);
```

- Benachrichtigungen über Änderungen von Node Values und Node Timestamps behandeln:

```
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);
}
```

Der in den vorangegangenen Beispielen gezeigte **OpcDataChangeTrigger**-Wert kann ebenso über eine Instanz der **OpcDataChangeFilter**-Klasse bestimmt werden. Auf diese Weise kann ein einmalig definierter Filter für mehrere Subscriptions und Monitored Items verwendet werden:

```
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);
```

Strukturierte Daten

Im den folgenden Abschnitten **wird davon ausgegangen**, dass der Server einen Node bereitstellt (über die NodeId „ns=2;s=Machine/Operator“), welcher vom **(fiktiven) Datentypen „StaffType“** Gebrauch macht.
Die Struktur des Datentypen wird dabei wie folgt definiert:

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

Einfachster Zugriff

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcValue und OpcDataObject.

Zum einfachen Zugriff auf Werte von Variablen-Nodes mit strukturierten Daten unterstützt das Framework die Verwendung des Schlüsselwortes `dynamic`. Zugriffe auf Variablen, welche mittels `dynamic` deklariert werden, werden von .NET zur Laufzeit ausgewertet. Das bedeutet, dass ohne vorherige explizite Implementierung eines .NET Typens auf die Daten eines strukturierten Datentypens zugegriffen werden kann. Ein derartiger Zugriff könnte wie folgt aussehen:

```
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);
```

Namen-basierter Zugriff

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcValue, OpcDataObject und OpcDataField.

Zum Namen-basierten Zugriff auf Werte von Variablen-Nodes mit strukturierten Daten kann direkt die **OpcDataObject** Klasse verwendet werden. Sind die Namen der Felder des Datentypen bekannt, kann auf diese auf die folgende Weise zugegriffen werden:

```
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);
```

Dynamischer Zugriff

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeSet, OpcAutomatism und OpcDataTypeSystem.

Für den „einfachsten Zugriff“ (siehe Einfachster Zugriff) mittels `dynamic` (in Visual Basic nicht typisiert mit `Dim`) wie auch dem Namen-basierten Zugriff (siehe Namen-basierter Zugriff) benötigt der verwendete **OpcClient** Informationen über die vom Server bereitgestellten Datentypen. Diese Informationen ermittelt die **OpcClient**-Klasse automatisch beim Aufruf von *Connect()* vom Server, wenn zuvor die Eigenschaft *UseDynamic* entweder auf dem **OpcClient** oder der statischen Klasse **OpcAutomatism** auf den Wert `true` gesetzt wird.

Durch die Aktivierung des Features ladet der **OpcClient** die notwendigen Typinformationen vom Server, damit im Anschluss des Aufrufs von *Connect()* auf diese ohne explizite Codierung oder ähnlichen wie in den vorherigen Abschnitten gezeigt zugegriffen werden kann.

Liegt eine **UANodeSet.xml** beziehungsweise eine XML-Datei mit der Beschreibung des Servers (eine XML-Datei deren Inhalt mit *UANodeSet* beginnt) vor, kann diese Datei auch in die Client-Anwendung geladen werden. Der **OpcClient** kann dann die Informationen aus dem NodeSet beziehen und muss nicht beim Herstellen einer Verbindung die Typinformation abrufen. Dies könnte dann wie folgt aussehen:

```
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);
}
```

Zu beachten ist, dass die **OpcClient**-Klasse beim Aufruf von *GetDataTypeSystem()* nach dem Setzen eines NodeSets eine Instanz der **OpcDataTypeSystem**-Klasse bereitstellt, welche das im NodeSet beschriebene Typ-System beschreibt. Wird hingegen kein NodeSet über die *NodeSet*-Eigenschaft der **OpcClient**-Klasse festgelegt, liefert der Aufruf von *GetDataTypeSystem()* eine **OpcDataTypeSystem**-Instanz, welche das Typ-System des Servers beschreibt, welches beim Aufruf von *Connect()* vom Server abgerufen wurde.

Typisierter Zugriff

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcDataTypeAttribute und OpcDataTypeEncodingAttribute.

Für den typisierten Zugriff werden .NET Typen in der Client-Anwendung so definiert, wie sie im Server definiert sind. Alle nötigen Metadaten werden dabei über Attribute bereitgestellt. Definition der .NET Typen für strukturierte Datentypen:

```
[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; }
}
```

Nach der Client-seitigen Definition eines .NET Typen für einen Server-seitig definierten strukturierten Datentypen kann dieser wie folgt verwendet werden:

```
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);
```

Datentypen generieren

Ist der typisierte Zugriff (siehe Typisierter Zugriff) zu verwenden besteht die Möglichkeit entweder nur bestimmte oder alle Datentypen eines OPC UA Servers über die OPC Watch zu generieren.

Generierung über den Server

Zur Generierung eines einzelnen in .NET implementierten Datentypen sind folgende Schritte durchzuführen:

1. **OPC Watch** öffnen
2. Neue Verbindung anlegen (+) und ggf. konfigurieren
3. Verbindung zum Server herstellen (Stecker-Symbol)
4. Einen Knoten auswählen...
    1. entweder einen Variablen-Knoten mit strukturierten DataType als Wert
    2. oder einen DataType-Knoten unter /Types/DataTypes/BaseDataType/Structure oder /Enumeration
5. Rechtsklick auf diesen Knoten
6. „Generate DataType“ klicken
7. Den Code in die Anwendung einfügen, fertig!

Zur Generierung aller Datentypen besteht die Möglichkeit diese entweder in einer Code-Datei (*.cs) oder Assembly-Datei (*.dll) zu generieren. Hierzu befolgen sind die folgenden Schritte durchzuführen:

1. **OPC Watch** öffnen
2. Neue Verbindung anlegen (+) und ggf. konfigurieren
3. Verbindung zum Server herstellen (Stecker-Symbol)
4. Server-Knoten „opc.tcp://...“ auswählen
5. Rechtsklick auf diesen Knoten
6. „Generate Models“ klicken
7. Im Dialog den gewünschten Dateitypen auswählen
8. „Speichern“ klicken
9. Die Datei dem Projekt hinzufügen, fertig!

Generierung über ein NodeSet

1. **OPC Watch** öffnen
2. In der oberen rechten Ecke der Anwendung das erste Symbol anklicken
3. NodeSet-Datei (gewöhnlich eine XML-Datei einer Companion Spezifikation) öffnen
4. NodeSet wird geladen und im Anschluss ein „Speichern unter...“-Dialog angezeigt
5. Im Dialog den gewünschten Dateitypen auswählen
6. „Speichern“ klicken
7. Die Datei dem Projekt hinzufügen, fertig!

Datentypen definieren

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcData, OpcDataTypeAttribute, OpcDataTypeEncodingAttribute, OpcDataTypeSystem und OpcDataTypeInfo.

Alternativ zur Generierung des Datentypen als .NET-Code oder .NET-Assembly (siehe Datentypen generieren) besteht können diese auch manuell definiert werden. Hierzu implementiert man den Typen, wie er vom Server definiert wurde. Das bedeutet, dass der .NET Type (unabhängig ob Struktur oder Klasse)

in genau der gleichen Reihenfolge die Felder des strukturierten Datentypen - bezüglich ihres Typens - bereitstellen muss. Alle weiteren Metadaten werden über entsprechende Attribute bereitgestellt:

```
[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;
    ...
}
```

Die zur Definition benötigten Informationen können entweder über das Handbuch zum OPC UA Server, dem verantwortlichen SPS-Entwickler oder über die **OpcClient**-Klasse bezogen werden. Zur Ermittlung der notwendigen Informationen über die **OpcClient**-Klasse können sie den Variablen-Node - der den strukturierten Datentypen verwendet - wie folgt untersuchen:

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

if (node is OpcVariableNodeInfo variableNode) {
    OpcNodeId dataTypeId = variableNode.DataTypeId;
    OpcDataTypeInfo dataType = client.GetDataTypeSystem().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);
}
```

Datentypen mit optionalen Feldern

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcData, OpcDataTypeAttribute, OpcDataTypeEncodingAttribute, OpcDataTypeEncodingMaskAttribute und OpcDataTypeMemberSwitchAttribute.

Zur Reduzierung der transportierten Datenmenge, besteht die Möglichkeit bestimmte Felder eines strukturierten Datentypens in Abhängigkeit von bestimmten Bedingungen beziehungsweise von dem Wert eines anderen Feldes innerhalb der Datenstruktur als existent oder fehlend im Datenstrom zu „markieren“. Zur „Markierung“, dass ein Feld verfügbar ist, dient somit entweder der Wert eines anderen Feldes oder

ein einzelnes Bit innerhalb der EncodingMask (ein Feld welches als Präambel vor den Daten im Datenstrom kodiert ist). Die Größe der EncodingMask wird in Anzahl Bytes im 'OpcDataTypeEncodingMaskAttribute' angegeben; wird die Eigenschaft 'Size' nicht explizit festgelegt, wird ihr Wert (bei Verwendung von 'OpcEncodingMaskKind.Auto') mit der kleinsten Anzahl an benötigten Bytes (anhand der Anzahl der optionalen Felder) belegt.

Zur Definition der optionalen Felder stehen unter anderen die folgenden Möglichkeiten zur Verfügung:

```
[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;
}
```

Historische Daten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, IOpcNodeHistoryNavigator, OpcHistoryValue und OpcAggregateType.

Laut OPC UA Spezifikation unterstützt jeder Node der Kategorie **Variable** die Aufzeichnung der Werte seines *Value Attributs* im zeitlichen Verlauf. Hierbei wird bei jeder Wertänderung des *Value Attributs* der neue Wert zusammen mit dem Zeitstempel (engl. Timestamp) des *Value Attributs* gespeichert. Diese **Paare bestehend aus Wert und Zeitstempel** werden als **historische Daten** bezeichnet. Wo der Server die Daten speichert, muss der Server selbst entscheiden. Der Client hingegen kann über das

IsHistorizing Attribut des Nodes feststellen, ob der Server für einen Node historische Daten bereitstellt beziehungsweise Wertänderungen historisch speichert. Von welchem Node auf die historischen Daten zugegriffen werden soll, wird durch den **OpcNodeId** des Nodes festgelegt. Hierbei kann der Client historische Daten lesen (engl. read), ändern (engl. update), ersetzen (engl. replace), löschen (engl. delete) oder auch erzeugen (engl. create). Am häufigsten werden die historischen Daten durch den Client gelesen. Eine Verarbeitung aller historischen Werte, unabhängig vom Lesen mit oder ohne Navigator, ist nicht notwendig.

Zum Lesen der historischen Daten kann der Client:

- Alle Werte innerhalb eines **offenen Zeitfensters** (= StartTime oder EndTime ist undefiniert) lesen
  - Alle Werte **ab einem bestimmten Zeitstempel** (= StartTime) lesen:

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

  - Alle Werte **bis zu einem bestimmten Zeitstempel** (= EndTime) lesen:

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

- Alle Werte innerhalb eines **geschlossenen Zeitfensters** (= StartTime und EndTime sind definiert) lesen:

```
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");
```

- Verarbeitet werden die Werte dann über eine Instanz, welche die **IEnumerable** Schnittstelle implementiert:

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

Zum seitenweisen Lesen (= es wird immer nur eine bestimmte Anzahl von Werten vom Server abgerufen) der historischen Daten kann der Client:

- Eine **bestimmte Anzahl von Werten** je Seite lesen:

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

- Eine **bestimmte Anzahl von Werten** je Seite innerhalb eines **offenen Zeitfensters** (= StartTime oder EndTime ist undefiniert) lesen
  - Eine **bestimmte Anzahl von Werten** je Seite **ab einem bestimmten Zeitstempel** (= StartTime) lesen:

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

  - Eine **bestimmte Anzahl von Werten** je Seite **bis zu einem bestimmten Zeitstempel** (= EndTime) lesen:

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

- Eine **bestimmte Anzahl von Werten** je Seite **innerhalb eines geschlossenen Zeitfensters** lesen:

```
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");
```

- Verarbeitet werden die Werte dann über eine Instanz, welche die **IOpcNodeHistoryNavigator** Schnittstelle implementiert:

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

historyNavigator.Close();
```

- Es muss immer sichergestellt werden, dass die *Close* Methode der **IOpcNodeHistoryNavigator** Instanz aufgerufen wird. Das ist notwendig, damit der Server die historischen Daten, die er für diese Anfrage gepuffert hat, wieder verwerfen kann. Alternativ zum expliziten Aufruf der *Close* Methode kann der Navigator auch in einem *using* Block verwendet werden:

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

Zum „verarbeiteten“ (engl. processed) Lesen der historischen Daten können verschiedene Arten der Aggregation über den **OpcAggregateType** ausgewählt werden:

- Zum Lesen des **kleinsten Wertes** innerhalb eines Zeitfensters:

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

- Zum Lesen des **durchschnittlichen Wertes** innerhalb eines Zeitfensters:

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

- Zum Lesen des **größten Wertes** innerhalb eines Zeitfensters:

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

Nodes

Methodenknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId und OpcCallMethod.

Welcher Methodenknoten aufgerufen werden soll, wird durch den **OpcNodeId** des Nodes festgelegt. Die dabei von einer Methode erwarteten Parameter können über die *Parameter* beim Aufruf von **CallMethod** beziehungsweise bei der jeweiligen **OpcCallMethod** Instanz angegeben werden. Zu beachten ist, dass immer zuerst der **OpcNodeId** des *Owners* der Methode und anschließend der **OpcNodeId** der Methode selbst angegeben werden muss. Der **OpcNodeId** des *Owners* legt dabei den Identifier des Objektknotens beziehungsweise des Objekttypknotens fest, der die Methode als *HasComponent-Referenz* referenziert.

- Einen einzelnen Methodenknoten ohne Parameter aufrufen (die Methode besitzt keine IN Argumente):

```
// 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*/);
```

- Einen einzelnen Methodenknoten mit Parameter aufrufen (die Methode besitzt IN Argumente):

```
// 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' */);
```

- Mehrere Methodenknoten ohne Parameter aufrufen (die Methoden besitzen keine IN Argumente):

```
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);
```

- Mehrere Methodenknoten mit Parameter aufrufen (die Methoden besitzen IN Argumente):

```
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);
```

Dateiknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcFile, OpcFileMode, OpcFileStream, OpcFileInfo, OpcFileMethods und SafeOpcFileHandle.

Nodes vom Typen **FileType** definieren per Definition durch die OPC UA Spezifikation bestimmte Eigenschaften (= Property Nodes) und Methoden (= Method Nodes), über die auf einen Datenstrom (engl. data stream) so zugegriffen werden können, als würde man auf eine Datei im Dateisystem zugreifen. Dabei werden ausschließlich Informationen über den Inhalt der logischen oder physikalischen Datei bereitgestellt. Ein eventuell vorhandener Pfad zur Datei wird, gemäß der Spezifikation, nicht zur Verfügung gestellt. Der Zugriff auf die Datei an sich wird mittels Open, Close, Read, Write, GetPosition und SetPosition realisiert. Dabei werden die Daten stets binär verarbeitet. Wie bei jeder anderen Plattform lässt sich auch bei der OPC UA beim Aufruf von Open ein Modus angeben, der die Art des geplanten Dateizugriffs vorgibt. Auch in der OPC UA kann man den exklusiven Zugriff auf eine Datei anfordern. Nach Aufruf der Open Methode erhält man einen numerischen Schlüssel für den weiteren Dateizugriff (engl. file handle). Dieser Schlüssel muss bei den Methoden Read, Write, GetPosition und SetPosition stets mit übergeben werden. Eine einmal geöffnete Datei muss wieder geschlossen (engl. close) werden, sobald diese nicht länger benötigt wird.

Der Zugriff auf Nodes vom Typen FileType kann über den OpcClient manuell durch die Verwendung der ReadNode und CallMethod Funktionen durchgeführt werden. Alternativ dazu bietet das Framework zahlreiche weitere Klassen, die unter anderem nach dem Vorbild des .NET Frameworks den Zugriff auf Nodes vom Typen FileType durchführen lassen. Auf welchen „File Node“ zugegriffen werden soll, wird durch den **OpcNodeId** des Nodes festgelegt.

Dateizugriff mit der **OpcFile** Klasse:

- Lesen des gesamten Inhalts einer Textdatei:

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

- Anhängen von weiteren Textdaten an eine Textdatei:

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

- Öffnen und Lesen der Datei mittels **OpcFileStream**:

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

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

- Öffnen und Schreiben der Datei mittels **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");
    // ...
}
```

Dateizugriff mit der **OpcFileInfo** Klasse:

- Erstellen einer **OpcFileInfo** Instanz:

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

- Arbeiten mit der **OpcFileInfo** Instanz:

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

    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.
        }
    }
}
```

Dateizugriff mit der **OpcFileMethods** Klasse:

- via .NET SafeHandle Konzept (umgesetzt durch die **SafeOpcFileHandle** Klasse):

```
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[data.Length - 1]);

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

- via numerischen 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);
}
```

Nur Dateizugriffe mittels **OpcFile**, **OpcFileInfo**, **OpcFileStream** und **SafeOpcFileHandle** garantieren auch eine implizite Freigabe einer geöffneten Datei, auch wenn der Aufruf der *Close* Methode „vergessen“ wurde. Spätestens beim Schließen der Verbindung zum Server werden dann vom **OpcClient** alle geöffneten Dateien automatisch geschlossen. Das ist jedoch nicht der Fall, wenn die Methoden der Klasse **OpcFileMethods** ohne „Secure“-Präfix verwendet werden.

Die OPC UA Spezifikation definiert keinen Weg, über den man einen Node als Node vom Typen FileType und somit als Datei-Node ausmachen kann. Hierfür bietet das Framework die Möglichkeit, einen Datei-Node über dessen Node Struktur als Datei-Node zu identifizieren:

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

Datentypknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcNodeInfo und OpcTypeNodeInfo.

Stellt ein Server einen Knoten bereit, über den er Informationen über einen durch den Server definierten Datentypen genauer beschreibt, dann kann es notwendig sein das der Client diese Informationen abrufen und weiterverarbeiten muss. Der einfachste Weg das zu tun ist mittels Browsing. Speziell für Datentypknoten liefert deshalb das Framework eine Spezialisierung der **OpcNodeInfo** - die **OpcTypeNodeInfo**. Über die Eigenchaften dieser Klasse können zusätzliche Informationen über den benutzerdefinierten Datentypen abgerufen werden. Wie zum Beispiel ob der Datentyp eine Enumeration ist. Ist das der Fall, dann können ebenso die von diesem Typen bereitgestellten Enum-Einträge abgerufen werden. Das funktioniert wie folgt:

```
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);
}
```

Datenknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcNodeInfo und OpcVariableNodeInfo.

Das Arbeiten mit Datenknoten beschränkt sich häufig auf das reine Lesen und Schreiben des *Value*-Attributs des Knotens. Dies funktioniert wie bei allen Knoten wie bereits unter „Werte von Node(s) lesen“ und „Werte von Node(s) schreiben“ gezeigt. Werden über den Wert des Knotens hinausgehende Informationen benötigt (ohne diese händisch per *ReadNode* Anfragen abzurufen), dann können diese mittels Browsing in einem, direkt und einfach vom Server angefordert werden. Das folgende Beispiel zeigt wie:

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

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

Darüber hinaus ist es auch möglich das Browsing direkt beim vom Datenknoten verwendeten Datentypknoten fortzusetzen:

```
var dataTypeNode = machineStatusNode.DataType;

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

Datenpunktknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient und OpcNodeId.

Datenpunktknoten, welche durch eine **OpcDataItemNode** durch einen Server bereitgestellt werden, stellen eine Erweiterung einer einfachen **OpcDataVariableNode** dar. Auch sie dienen primär der Bereitstellung von Werten. Stellen aber darüber hinaus auch noch weitere nützliche Metadaten zur Verfügung. Dazu gehört vor allem die *Definition* Eigenschaft welche als Property-Node diesem Node als Kindknoten zugeordnet ist. Sie dient der korrekten Weiterverarbeitung und Interpretation der vom Knoten bereitgestellten Daten. Der dabei vom Hersteller abhängige Wert soll dem Anwender beschreiben, wie der repräsentierte Wert des Knotens zustande kommt. Davon abgesehen funktionieren auch auf dieser Spezialisierung die bereits aus „Werte von Node(s) lesen“ und „Werte von Node(s) schreiben“ bekannten Operationen.

Datenpunktknoten für analoge Werte

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcNodeId, OpcNodeInfo und OpcAnalogItemNodeInfo.

Diese Art von Knoten stellt eine Spezialisierung des Datenpunktknotens **OpcDataItemNode** dar. Die im Speziellen bereitgestellten Werte sind primär analoger Natur und können über die zusätzlichen Eigenschaften *InstrumentRange*, *EngineeringUnit* und *EngineeringUnitRange* genauer beschrieben werden. Während die *InstrumentRange* den Wertebereich der analogen Daten der Quelle beschreibt, definiert die *EngineeringUnit* die Maßeinheit des vom Knoten bereitgestellten Wertes. Wird der Wert im normalen Betrieb ermittelt, so liegt dieser auch im Wertebereich der über die *EngineeringUnitRange* Eigenschaft festgelegt werden kann. Die durch die *EngineeringUnit* Eigenschaft beschriebene Maßeinheit definiert sich durch die UNECE Empfehlungen N° 20. Diese Empfehlungen basieren auf dem internationalen System für Maßeinheiten (engl. International System of Units, kurz SI Units). Zur vereinfachten Verarbeitung der zusätzlichen Informationen dieser Node bietet die **OpcAnalogItemNodeInfo** die entsprechende Schnittstelle beim Browsen:

```
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}");
}
```

Die in einer **OpcEngineeringUnitInfo** enthaltenen Informationen beschreiben eine Maßeinheit der UNECE Tabelle für Maßeinheiten. Die möglichen Einheiten können bei der OPC Foundation nachgeschlagen werden: UNECE Maßeinheiten in OPC UA (aktuell nicht verfügbar). Alternativ bietet auch das MTConnect Institut die UNECE Tabelle auf GitHub an: UNECE Maßeinheiten in OPC UA (MTConnect@GitHub).

Ereignisse

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcObjectTypes, OpcEventReceivedEventArgs, OpcEvent, OpcAlarmCondition, OpcSimpleAttributeOperand, OpcFilter, OpcEventFilter und OpcEventSeverity.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Ereignisse informieren einen Abonnenten (wie Clients) über Abläufe, Zustände und Systemspezifische Begebenheiten. Derartige Informationen können Interessenten direkt über **globale Ereignisse** zugestellt werden. Ein globales Ereignis wird stets über den „Server“ Node des Servers veröffentlicht. Aus diesem Grund werden globale Ereignisse auch über den Server Node wie folgt abonniert:

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

Behandelt wird ein Ereignis im Allgemeinen mit einer Methode der folgenden Signatur:

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

Die über die **OpcEventReceivedEventArgs**-Instanz empfangenen Ereignisdaten können über die *Event*-Eigenschaft abgerufen werden. Die Eigenschaft liefert dabei stets eine Instanz vom Typen **OpcEvent**. Wenn es sich bei den empfangenen Ereignisdaten aber um Daten eine Spezialisierung der **OpcEvent**-Klasse handelt, dann können diese wie folgt einfach gecastet werden:

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

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

Der zuvor gezeigte Aufruf abonniert alle vom Server global veröffentlichten Ereignisse. Zur Einschränkung der dabei erhaltenen Informationen sowie der Ereignisse die überhaupt vom Server an den Client gesendet werden kann ein Ereignisfilter (engl. event filter) definiert und beim Abschluss eines Abonnements (engl. Subscription) an den Server übermittelt werden:

```
// 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);
```

Bei der Erstellung eines Ereignisfilters wird stets eine **OpcClient**-Instanz benötigt, die bereits eine Verbindung zum Zielserver aufgebaut haben muss. Diese wird benötigt, da beim Zusammenstellen des Filters vom Server die für den Filter relevanten Typ-Informationen vom Server abgerufen werden. Im obigen Beispiel werden so über den Client alle Eigenschaften des Node-Typens **OpcAlarmConditionNode** rekursiv bis zum **OpcNode** gesammelt und entsprechend der *Where*- und *Select*-Klauseln Regeln für die Auswahl der Ereignisdaten erstellt. Die dabei zu analysierenden Node-Typen können mit Komma getrennt an die *FromEvents(...)*-Methode übergeben werden:

```
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);
```

Mit Hilfe der *Where(...)*-Methode können dann die durch *FromEvents(...)* gesammelten Informationen über die von den Node-Typen bereitgestellten Eigenschaften eingeschränkt werden. Hierzu bietet das Framework diverse Operatorenüberladungen (<=, <, >, >= und ==). Zur logischen Kombination der Operanden stehen zudem die logischen Operatoren ODER (|) und UND bereit (&). Zudem stehen diverse Methoden für weitere Einschränkungen bereit, wie: *Like*, *Between*, *InList*, *IsNull*, *Not* und *OfType*.

Nach erfolgter Einschränkung können mittels *Select(...)*-Methode die Eigenschaften selektiert werden, die beim Eintreten eines Ereignisses (das den unter *Where(...)* vorgegebenen Bedingungen entspricht) zusätzlich vom Server an den Client übertragen werden.

Ereignisknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcEvent und OpcSubscription.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Es ist nicht immer zweckmäßig, dass ein Server Ereignisse durchwegs global über den Server Node an alle Abonnenten sendet. Häufig spielt deshalb der Kontext eine entscheidende Rolle, ob ein Ereignis für einen Abonnementen von Interesse ist. Zur Definition von dafür vorgesehenen lokalen Ereignissen dienen Ereignisknoten. Verwendet der Server einen Ereignisknoten zur Bereitstellung von Ereignisdaten für einen Knoten, dann ist der Ereignisknoten ein sogenannter „Benachrichtiger“ (engl. notifier) des Knotens. Aus diesem Grund erkennt man auch an einer *HasNotifier*-Referenz welcher Ereignisknoten Ereignisdaten an einen Knoten meldet. Zu beachten ist, dass ein Ereignisknoten ein „Benachrichtiger“ für mehrere Knoten sein kann. Daraus ergibt sich auch, dass lokale Ereignisse stets über die benachrichtigten Knoten abonniert werden:

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

Behandelt wird ein Ereignis im Allgemeinen mit einer Methode der folgenden Signatur. Die weitere Verarbeitung der Ereignisdaten verhält sich identisch zur Verarbeitung globaler Ereignisse (siehe Abschnitt 'Arbeiten mit Ereignissen').

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

Selbstverständlich können auch lokale Ereignisse, wie im Abschnitt 'Arbeiten mit Ereignissen' gezeigt, gefiltert werden.

Nachdem ein Abonnent (ein Client) generell nur über Ereignisse informiert wird, solange er mit dem Server in Verbindung steht und ein Abonnement veranlasst hat, weiß ein Abonnent nicht welche Ereignisse bereits vor dem Aufbau einer Verbindung zum Server aufgetreten sind. Soll der Server Abonnenten nachträglich über vergangene Ereignisse informieren, dann können Abonnenten diese vom Server wie folgt anfordern. **Generell steht ein Server aber nicht in der Pflicht vergangene Ereignisse bereitzustellen.**

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

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

Ereignisknoten mit Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcEvent, OpcCondition und OpcEventSeverity.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Spezialisierung des **OpcEvent**'s (vorgestellt im Abschnitt 'Arbeiten mit Ereignissen') ist die Klasse **OpcCondition**. Sie dient der Bereitstellung von Ereignisdaten an die bestimmte Bedingungen geknüpft sind. Nur im Falle, dass eine einem Ereignisknoten zugesprochene Bedingung zutrifft, wird ein solches Ereignis ausgelöst. Zu den Informationen des Ereignisses gehören Informationen über den Zustand der Bedingung, wie auch Informationen die an die Auswertung der Bedingung geknüpft sind. Da diese Informationen je nach Szenario unterschiedlich komplex sein können stellt die **OpcCondition** die Basisklasse aller Ereignisdaten dar an denen eine Bedingung geknüpft ist.

Neben den allgemeinen Eigenschaften eines Ereignisses (bereitgestellt durch die **OpcEvent**-Basisklasse) liefert eine Instanz der **OpcCondition** Informationen die für die weitere Verarbeitung des Ereignisses für den Client von Interesse sein können. So kann der Client prüfen, ob im Allgemeinen der Server das Ereignis für einen Client als relevant beurteilt (siehe *IsRetained*-Eigenschaft). Ebenso kann der Client den Zustand der Bedingung, also ob sie aktiv oder inaktiv ist auswerten (siehe *IsEnabled*-Eigenschaft).

```
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);
    }
}
```

Wünscht der Client keine weitere Auswertung der Bedingung, dann kann der Client diese deaktivieren:

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

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

Zusätzlich kann der Client auch einen Kommentar zum aktuellen Zustand der Bedingung hinzufügen:

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

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

Ereignisknoten mit Dialog-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcCondition, OpcDialogCondition, OpcEventSeverity und OpcDialogRequestedEventArgs.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Spezialisierung der **OpcCondition** ist die **OpcDialogCondition**. Die mit diesem Ereignis verbundene Bedingung ist ein Dialog mit den Abonnenten. Dabei besteht eine solche Bedingung aus einer Meldung (engl. Prompt), Antwortoptionen (engl. Response Options) sowie Informationen, welche Option standardmäßig ausgewählt werden sollte (*DefaultResponse*-Eigenschaft), welche Option zur Bestätigung des Dialoges (*OkResponse*-Eigenschaft) und welche zum Abbruch des Dialoges (*CancelResponse*-Eigenschaft) verwendet wird. Wird ein solches Dialogbedingtes Ereignis ausgelöst, wartet der Server darauf, dass einer der Abonnenten ihm auf das Ereignis eine Antwort in Form der getroffenen Auswahl anhand der vorgegebenen Antwortoptionen liefert. Die Bedingung zur weiteren Verarbeitung, der Operationen die an den Dialog geknüpft sind, ist somit die Antwort auf eine Aufgabenstellung, eine Frage, eine Information oder eine Warnung. Die hierfür vom Ereignis bereitgestellten Informationen können wir folgt verarbeitet und an den Server zurückgemeldet werden:

```
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);
    }
}
```

Vom Standardverfahren zur Ereignisbehandlung abgesehen bietet die **OpcClient** Klasse zudem das *DialogRequested*-Event. Werden **OpcDialogCondition** Ereignisdaten vom Client empfangen und durch keinen Handler beantwortet, dann führt der Client das *DialogRequested*-Event aus um darüber eine Dialog-Verarbeitung durchzuführen. Hierbei vereinfacht sich auch etwas die Handhabung der Ereignisdaten, da diese typisiert an den Eventhandler übergeben werden können:

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

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

Es muss lediglich die *SelectedResponse*-Eigenschaft der Ereignisargumente festgelegt werden. Den Aufruf der *Respond(...)*-Methode der **OpcDialogCondition** wird nach der Ausführung des Eventhandlers durch den **OpcClient** erledigt.

Ereignisknoten mit Feedback-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcEvent, OpcCondition, OpcAcknowledgeableCondition und OpcEventSeverity.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Basierend auf **OpcCondition** Ereignissen stellt die **OpcAcknowledgeableCondition** eine Spezialisierung dar die als Basisklasse für Bedingungen mit Feedback-Anforderung zum Einsatz kommt. Ereignisse dieser Art definieren, dass bei Erfüllung ihrer Bedingung quasi eine „Meldung mit Rückschein“ abgesetzt wird. Der „Rückschein“ - also das Feedback - kann dabei sowohl zur Steuerung weiterer Abläufe als auch zur einfachen Quittierung von Hinweisen und Warnungen dienen. Der dafür von der Spezifikation vorgesehene Feedback-Mechanismus ist in zwei Stufen unterteilt. Während die erste Stufe eine Art „Lesebestätigung“ darstellt, stellt die zweite Stufe eine Art „Lesebestätigung mit Abnicken“ dar. OPC UA definiert die Lesebestätigung als einfache Bestätigung (engl. Confirm) und die Lesebestätigung mit Abnicken als Zustimmung (engl. Acknowledge). Für beide Bestätigungsweisen stellen die Ereignisse entsprechende *Confirm*- und *Acknowledge*-Methoden bereit. Per Definition soll die Ausführung des „Acknowledge“-Vorgangs ein explizites Ausführen des „Confirm“-Vorgangs unnötig machen. Dem gegenüber ist es aber möglich zuerst eine Confirm- und anschließend und somit getrennt davon eine Acknowledge-Bestätigung zu senden. Unabhängig von der Reihenfolge und der Art des Feedbacks kann optional beim Confirm beziehungsweise beim Acknowledge ein Kommentar des Sachbearbeiters angegeben werden. Ein Acknowledgement als Feedback könnte wie folgt implementiert werden:

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

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

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

Zusätzlich kann bei der Verarbeitung dieser Art von Ereignissen geprüft werden, ob das Ereignis bereits mittels Confirm (siehe *IsConfirmed*-Eigenschaft) oder mittels Acknowledge (siehe *IsAcked*-Eigenschaft) bereits bestätigt wurde. **Zu beachten ist aber, dass ein Server stets die Interpretation wie auch die auf das jeweilige Feedback folgende Logik selbst definieren muss.** Ob also ein Server Gebrauch von beiden Feedback Optionen macht oder nur von einer ist dem jeweiligen Entwickler überlassen. Im besten Fall macht ein Server zumindest von der *Acknowledge*-Methode Gebrauch, da diese von der Spezifikation als „stärker“ definiert ist.

Ereignisknoten mit Alarm-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcAlarmCondition, OpcAcknowledgeableCondition und OpcEventSeverity.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm &

Conditions.

Die in der OPC UA wohl wichtigste Implementierung der **OpcAcknowledgeableCondition** ist die **OpcAlarmCondition**. Mit Hilfe von **OpcAlarmCondition**-Ereignissen ist es möglich Ereignisse zu definieren deren Verhalten mit einem Nachttischwecker vergleichbar sind. Dementsprechend wird ein solches Ereignis aktiv (siehe *IsActive*-Eigenschaft), wenn die mit diesem verknüpfte Bedingung erfüllt ist. Im Falle eines Weckers also zum Beispiel das „Erreichen der Weckzeit“. Ein Alarm der hingegen zum Beispiel mit einer Weckzeit eingestellt wurde, aber nicht beim Erreichen dieser aktiv werden soll wird als unterdrückter Alarm bezeichnet (engl. suppressed alarm, siehe *IsSuppressed*- und *IsSuppressedOrShelved*-Eigenschaft). Wird aber ein Alarm aktiv, kann dieser zurückgestellt (engl. shelved) werden (siehe *IsSuppressedOrShelved*-Eigenschaft). Dabei kann ein Alarm einmalig („One Shot Shelving“) oder zeitlich („Timed Shelving“) zurückgestellt werden (siehe *Shelving*-Kindknoten). Alternativ kann ein zurückstellter Alarm auch wieder „vorgestellt“ (engl. unshelved) werden (siehe *Shelving*-Kindknoten).

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

    if (alarm != null) {
        Console.Write($"Alarm {alarm.ConditionName} is");
        Console.WriteLine($"{(alarm.IsActive ? "active" : "inactive")}!");
    }
}
```

Ereignisknoten mit diskreten Alarm-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcDiscreteAlarm, OpcOffNormalAlarm und OpcTripAlarm.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Ausgehend von den **OpcAlarmCondition**-Ereignisdaten gibt es mehrere Spezialisierungen die explizit für bestimmte Arten von Alarmen definiert wurden um die Form, den Grund oder den Inhalt eines Alarms bereits durch die Art des Alarms genauer zu spezifizieren. Eine Unterklasse solcher selbstbeschreibenden Alarme sind die diskreten Alarme. Als Basis eines diskreten Alarms dient die Klasse **OpcDiscreteAlarm**. Sie definiert einen Alarmzustand, der verwendet wird, um Typen in Alarmzustände zu klassifizieren, wobei der Eingang für den Alarm nur eine bestimmte Anzahl von möglichen Werten annehmen kann (z.B. wahr / falsch, läuft / angehalten / beendet). Stellt ein Alarm einen diskreten Zustand dar, der als nicht normal angesehen wird, kommt der **OpcOffNormalAlarm** oder eine Unterklasse von diesem zum Einsatz. Ausgehend von dieser Alarmklasse bietet das Framework eine weitere Konkretisierung mit dem **OpcTripAlarm**. Der **OpcTripAlarm** wird aktiv, wenn zum Beispiel an einem überwachten Gerät ein anomaler Fehler auftritt, z.B. wenn der Motor aufgrund einer Überlastung abgeschaltet wird.

```
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!");
    }
}
```

Ereignisknoten mit Alarm-Bedingungen für Grenzwerte

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs und OpcLimitAlarm.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Werden vom Server Prozessspezifische Grenzwerte geprüft und wird dann der Ausgang der Prüfung bei Grenzwertüberschreitungen / -unterschreitungen publiziert, dann repräsentiert die **OpcLimitAlarm** Klasse den zentralen Anlaufpunkt zum Einstieg in die Klassen der Grenzwert-Alarme (engl. limit alarms). Mit Hilfe dieser Klasse werden Grenzwerte in bis zu vier Stufen unterteilt. Zur Differenzierung dieser werden sie als LowLow, Low, High und HighHigh bezeichnet (genannt in der Reihenfolge ihrer metrischen Ordnung). Per Definition muss der Server nicht alle Grenzwerte definieren.

```
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);
    }
}
```

Ereignisknoten mit Alarm-Bedingungen für ausschließliche Grenzwerte

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcLimitAlarm, OpcExclusiveLimitAlarm, OpcExclusiveDeviationAlarm, OpcExclusiveLevelAlarm und OpcExclusiveRateOfChangeAlarm.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Unterklasse der **OpcLimitAlarm**-Ereignisse ist die Klasse **OpcExclusiveLimitAlarm**. Wie ihr Name bereits verrät, dient sie der Definition von Grenzwertalarmen für ausschließliche Grenzen. Ein solcher Grenzwertalarm verwendet dabei Werte für die Grenzen, die sich gegenseitig ausschließen. Das bedeutet, dass wenn ein Grenzwert überschritten / unterschritten wurde, dass nicht zugleich ein anderer Grenzwert überschritten / unterschritten sein kann.

Im Rahmen der OPC UA gibt es drei weitere Spezialisierungen des **OpcExclusiveLimitAlarm**'s.

OpcExclusiveDeviationAlarm

Diese Art von Alarm wird eingesetzt, wenn eine geringfügige Abweichung von definierten Grenzwerten festgestellt wird.

OpcExclusiveLevelAlarm

Diese Art von Alarm wird verwendet, wenn ein Grenzwert überschritten wird. Das betrifft typischerweise ein Instrument - wie zum Beispiel einen Temperatursensor. Diese Art von Alarm wird aktiv, wenn der beobachtete Wert über einem oberen Grenzwert oder unter einem unteren Grenzwert liegt.

OpcExclusiveRateOfChangeAlarm

Diese Art von Alarm wird verwendet, um eine ungewöhnliche Änderung oder fehlende Änderung eines gemessenen Werts in Bezug auf die Geschwindigkeit, mit der sich der Wert geändert hat, zu melden. Der Alarm wird aktiv, wenn die Rate, mit der sich der Wert ändert, einen definierten Grenzwert über- oder unterschreitet.

Ereignisknoten mit Alarm-Bedingungen für nicht-ausschließliche Grenzwerte

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcEventReceivedEventArgs, OpcLimitAlarm, OpcNonExclusiveLimitAlarm, OpcNonExclusiveDeviationAlarm, OpcNonExclusiveLevelAlarm und OpcNonExclusiveRateOfChangeAlarm.

Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Unterklasse der **OpcLimitAlarm**-Ereignisse ist die Klasse **OpcNonExclusiveLimitAlarm**. Wie ihr Name bereits verrät, dient sie der Definition von Grenzwertalarmen für nicht-ausschließliche Grenzen. Ein solcher Grenzwertalarm verwendet dabei Werte für die Grenzen, die sich gegenseitig nicht ausschließen. Das bedeutet, dass wenn ein Grenzwert überschritten / unterschritten wurde, dass zugleich ein anderer Grenzwert überschritten / unterschritten sein kann. Die dabei verletzten Grenzen können mit den Eigenschaften *IsLowLow*, *IsLow*, *IsHigh* und *IsHighHigh* der Ereignisdaten geprüft werden.

Im Rahmen der OPC UA gibt es drei weitere Spezialisierungen des **OpcNonExclusiveLimitAlarm**'s.

OpcNonExclusiveDeviationAlarm

Diese Art von Alarm wird eingesetzt, wenn eine geringfügige Abweichung von definierten Grenzwerten festgestellt wird.

OpcNonExclusiveLevelAlarm

Diese Art von Alarm wird verwendet, wenn ein Grenzwert überschritten wird. Das betrifft typischerweise ein Instrument - wie zum Beispiel einen Temperatursensor. Diese Art von Alarm wird aktiv, wenn der beobachtete Wert über einem oberen Grenzwert oder unter einem unteren Grenzwert liegt.

OpcNonExclusiveRateOfChangeAlarm

Diese Art von Alarm wird verwendet, um eine ungewöhnliche Änderung oder fehlende Änderung eines gemessenen Werts in Bezug auf die Geschwindigkeit, mit der sich der Wert geändert hat, zu melden. Der Alarm wird aktiv, wenn die Rate, mit der sich der Wert ändert, einen definierten Grenzwert über- oder unterschreitet.

Bearbeitung des Adressraums

Erstellen von Knoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcAddNode, OpcAddNodeResult, OpcAddNodeResultCollection, OpcAddDataItemNode, OpcAddAnalogItemNode, OpcAddObjectNode, OpcAddFolderNode, OpcAddDataVariableNode und OpcAddPropertyNode.

Während ein Server einen vordefinierten Satz an „Standardknoten“ seinen Clients zur Verfügung stellt, können die Clients den Server dazu veranlassen weitere Knoten bereitzustellen. Hierzu dient die „AddNodes“-Schnittstelle des Servers. Ausgehend von der **OpcAddNode** Klasse bietet das Framework zahlreiche weiter Unterklassen welche zum Anlegen von Typ-spezifischen Knoten verwendet werden können. Ein neuer Ordnerknoten kann somit auf folgende Weise angelegt werden:

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

Die verwendeten Parameter stellen das notwendige Minimum der benötigten Informationen bereit. Der erste Parameter „name“ wird für den Namen (Name Eigenschaft), den Anzeigename (DisplayName Eigenschaft), den symbolischen Namen (SymbolicName Eigenschaft) und die Beschreibung (Description Eigenschaft) der Node verwendet. Der zweite Parameter „nodeId“ dient dem Server als Vorgabe, welchen Identifier er dem Knoten zuweisen soll. Ist der Identifier bereits im Adressraum der Knoten des Servers für einen anderen Knoten vergeben, wird der Knoten nicht angelegt und der Client erhält als Ergebnnis den Code „BadNodeIdRejected“. Wird stattdessen, wie im Beispiel gezeigt, OpcNodeId.Null verwendet, wird der Server selbstständig für den Knoten einen neuen Identifier erstellen und ihm diesen zuweisen. Über den Parameter „parentNodeId“ wird schließlich noch der Identifier des Elternknotens festgelegt, unter dem der neue Knoten im Baum angelegt werden soll.

Als Ergebnis liefert der Aufruf der AddNode-Methode eine Instanz der OpcAddNodeResult Klasse. Diese liefert neben den Informationen über den Ausgang der Operation auch den Identifier, der schlussendlich für den neuen Knoten verwendet wurde:

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

Neben der Möglichkeit einen einzelnen Knoten anzulegen, können auch mehrere Knoten gleichzeitig angelegt werden:

```
OpcNodeId jobsNodeId = result.NodeId;

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

Als Ergebnis liefert der Aufruf der AddNodes-Methode eine Instanz der OpcAddNodeResultCollection Klasse. Diese enthält OpcAddNodeResult Instanzen die auf die gleiche Weise wie zuvor beschrieben ausgewertet und weiter verarbeitet werden können.

Neben der Möglichkeit einen oder mehrere Knoten zugleich anzulegen, besteht die Möglichkeit ganze Bäume von Knoten an die Methoden AddNode und AddNodes zu übergeben:

```
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"))));
```

Ein derartiger Baum kann auch über die entsprechenden Eigenschaften konstruiert werden:

```
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);
```

Sollen andere Typ-Definitionen für die Knoten verwendet werden, als die die durch die entsprechenden Unterklassen von OpcAddNode bereitgestellt werden, dann besteht die Möglichkeit Objekt- und Variablen-Knoten anhand ihrer Typ-Definition anzulegen:

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

Im Gegensatz zum Anlegen von Knoten anhand von durch die Foundation definierten Typ-Definitionen ist es auch möglich Knoten anhand des Identifiers ihrer Typ-Definition anzulegen. Hierzu muss der zu verwendende Typ über eine Object- beziehungsweise Variablen-spezifische Typ-Definition im Voraus deklariert werden. Dieser kann dann immer wieder zum Erstellen von entsprechenden Knoten verwendet werden:

```
// 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")));
```

Löschen von Knoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcDeleteNode, OpcStatus und OpcStatusCollection.

Vom Server bereitgestellte wie auch von einen Client erstellte Knoten können über die „DeleteNodes“-Schnittstelle des Server gelöscht werden. Benötigt wird hierzu primär der Identifier des Knotens der gelöscht werden soll:

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

Die im obigen Beispiel gezeigte Möglichkeit verwendet zum Löschen eine Instanz der OpcDeleteNode Klasse, welche standardmäßig auch das Löschen aller Verweise, die auf den Knoten zeigen, einschließt. Sollen jedoch die Verweise auf den Knoten bestehen bleiben, muss zusätzlich der Parameter „includeTargetReferences“ mit dem Wert „false“ belegt werden:

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

Neben der Möglichkeit einen einzelnen Knoten zu löschen, können auch mehrere Knoten gleichzeitig gelöscht werden:

```
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"));
```

Erstellen von Verweisen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcAddReference, OpcStatus und

OpcStatusCollection.

Knoten die bereits im Adressraum eines Servers existieren, können unterschiedliche Beziehungen zueinander besitzen. Diese Beziehungen werden über so genannte Verweise (engl. References) im Adressraum des Servers beschrieben. Während Knoten physikalisch in der Rolle des Eltern- beziehungsweise Kindknotens eingeordnet werden, können zwischen ihnen noch weitere logische Beziehungen existieren. Diese Beziehungen dienen somit der genaueren Definition der Funktion und der Abhängigkeiten der Knoten zueinander. Darüber hinaus können über derartige Verweise weitere Bäume im Adressraum des Servers definiert werden, ohne bereits vorhandene Knoten neu organisieren zu müssen.

Zum Anlegen eines Verweises bedient sich der Client der „AddReferences“-Schnittstelle des Servers wie folgt:

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

Das gezeigte Beispiel erzeugt eine Beziehung ausgehend vom Knoten mit dem Identifier „ns=2;s=Plant“ zum Knoten „ns=2;s=Machines/MAC01“ vom Typen „Organizes“. Die gezeigte Anweisung entspricht demnach 1:1 dem Ergebnis des folgenden Beispiels:

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

Während der erste Identifier immer den Quellknoten angibt, gibt der Parameter „targetNodeId“ den Identifier des Zielknotens an. Der zusätzliche Parameter „targetNodeCategory“ muss der Category Eigenschaft (= NodeClass laut Foundation) des Zielknotens entsprechen, denn anhand dieser versichert sich der Server, dass der gewünschte Zielknoten ausreichend bekannt ist.

Neben der Möglichkeit einen einzelnen Verweis anzulegen, können auch mehrere Verweise gleichzeitig angelegt werden:

```
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));
```

Das oben gezeigte Beispiel organisiert beispielhaft drei Knoten die jeweils eine Maschine für sich unterhalb des Knotens „Machines“ repräsentieren unterhalb der Knoten „Plant01“ und „Plant02“. Die hier standardmäßig verwendete Beziehung „Organizes“ führt dazu, dass die Knoten weiterhin unterhalb des „Machines“ Knoten verfügbar sind, zusätzlich aber auch unterhalb des „Plant01“ Knotens die Knoten „MAC01“ und „MAC02“, wie auch der Knoten „MAC03“ unterhalb des Knotens „Plant02“.

Löschen von Verweisen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcDeleteReference, OpcStatus und

OpcStatusCollection.

Bereits bestehende Verweise zwischen den Knoten im Adressraum des Servers können über die „DeleteReferences“-Schnittstelle des Server gelöscht werden:

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

Das gezeigte Beispiel löscht dabei alle „Organizes“-Verweise die in Richtung des Knotens mit dem Identifier „ns=2;s=Machines/MAC03“ als auch in Richtung des Knotens mit dem Identifier „ns=2;s=Plant“ zwischen den beiden Knoten existieren. Sollen hingegen nur „Organizes“-Verweise ausgehend von Quell- zum Zielknoten in nur eine bestimmte Richtung gelöscht werden, kann das wie folgt durchgeführt werden:

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

Sollen hingegen Verweise die nicht vom Typen „Organizes“ sind gelöscht werden, dann können diese über den zusätzlichen „referenceType“ beziehungsweise „referenceTypeId“ Parameter wie folgt angegeben werden:

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

Neben der Möglichkeit einzelne Verweise zu löschen, können auch mehrere Verweise gleichzeitig gelöscht werden:

```
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"));
```

Konfiguration des Clients

Allgemeine Konfiguration

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcCertificateStores und OpcCertificateStoreInfo.

In allen hier gezeigten Codeausschnitten wird stets der Client über den Code konfiguriert (wenn nicht mit der Standardkonfiguration des Clients gearbeitet wird). Zentrale Anlaufstelle für die Konfiguration der Clientanwendung, der Sitzungsparameter sowie Verbindungsparameter ist die **OpcClient** Instanz. Alle Einstellungen zum Thema Sicherheit finden sich als Instanz der **OpcClientSecurity** Klasse über die *Security* Eigenschaft des Clients. Alle Einstellungen zum Thema Zertifikatspeicher finden sich als Instanz der **OpcCertificateStores** Klasse über die *CertificateStores* Eigenschaft des Clients.

Soll der Client auch per XML konfigurierbar sein, dann besteht die Möglichkeit, die Konfiguration des

Clients entweder direkt aus einer bestimmten oder aus einer beliebigen XML Datei zu laden. Welche Schritte dazu nötig sind, sehen Sie im Abschnitt „Vorbereiten der Clientkonfiguration via XML“.

Sobald die entsprechenden Vorbereitungen zur Konfiguration der Clientkonfiguration via XML getroffen wurden, können die Einstellungen wie folgt geladen werden:

- Laden der Konfigurationsdatei über die App.config

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

- Laden der Konfigurationsdatei über den Pfad zur XML Datei

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

Zur Konfiguration der Clientanwendung stehen unter anderem die folgenden Möglichkeiten zur Verfügung:

- Konfiguration der Anwendung
  - 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 (unterhalb des *OpcApplicationConfiguration* Elements):

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

- Konfiguration der Sitzungsparameter
  - via Code:

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

  - via XML (unterhalb des *OpcApplicationConfiguration* Elements):

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

- Konfiguration der Verbindungsparameter
  - via Code:

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

  - via XML (unterhalb des *OpcApplicationConfiguration* Elements):

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

- Konfiguration der Zertifikatspeicher
  - 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 (unterhalb des *OpcApplicationConfiguration* Elements):

```
  <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>
```

Konfiguration des Zertifikats

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcCertificateManager, OpcClientSecurity, OpcCertificateStores und OpcCertificateStoreInfo.

Empfohlen werden Zertifikate vom Typen .der, .pem, **.pfx** und **.p12**. Soll der Client einen sicheren Serverendpunkt verwenden (bei dem der **OpcSecurityMode** gleich *Sign* oder *SignAndEncrypt* ist), muss das Zertifikat über einen privaten Schlüssel verfügen.

1. Ein **vorhandenes Zertifikat** wird wie folgt aus einen beliebigen Pfad geladen:

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

2. Ein **neues Zertifikat** kann wie folgt (im Speicher) erzeugt werden:
```
var certificate = OpcCertificateManager.CreateCertificate(client);
```
3. Gespeichert werden kann das Zertifikat unter einem beliebigen Pfad über:
```
OpcCertificateManager.SaveCertificate("MyClientCertificate.pfx", certificate);
```
4. Das Clientzertifikat festlegen:
```
client.Certificate = certificate;
```
5. Das Zertifikat muss im Zertifikatstore für **Anwendungszertifikate** (= ApplicationStore) gespeichert sein:
```
if (!client.CertificateStores.ApplicationStore.Contains(certificate))
    client.CertificateStores.ApplicationStore.Add(certificate);
```
6. Wird **kein oder ein ungültiges Zertifikat** verwendet, wird standardmäßig automatisch ein neues Zertifikat erzeugt/verwendet. Soll zudem sichergestellt sein, dass der Client nur das angegebene Zertifikat verwendet, muss diese Funktion deaktiviert werden. Zum **Deaktivieren der Funktion** die Eigenschaft **AutoCreateCertificate** auf den Wert *false* stellen:
```
client.CertificateStores.AutoCreateCertificate = false;
```

Konfiguration der Benutzeridentität

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcUserIdentity, OpcClientIdentity, OpcCertificateIdentity, OpcClientSecurity, OpcCertificateStores und OpcCertificateStoreInfo.

Erwartet ein Server zusätzlich zum Clientzertifikat Informationen über die Identität des Benutzers, muss dieser über die *UserIdentity* Eigenschaft festgelegt werden. Zur Auswahl stehen dabei Identitäten basierend auf Benutzername-Passwort oder einem Zertifikat. Im Falle dessen, dass der Server eine anonymisierte Benutzeridentität unterstützt, muss keine spezielle Identität festgelegt werden.

- Festlegen einer Benutzeridentität bestehend aus **Benutzername-Passwort**:
```
client.Security.UserIdentity = new OpcClientIdentity("userName", "password");
```
- Festlegen einer Benutzeridentität mittels **Zertifikat (mit privatem Schlüssel)**:
```
client.Security.UserIdentity = new OpcCertificateIdentity(new
X509Certificate2("Doe.pfx"));
```
- Festlegen einer **anonymen** Benutzeridentität (standardmäßig bereits so vorkonfiguriert):
```
client.Security.UserIdentity = null;
```

Konfiguration des Serverendpunkts

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcClientSecurity, OpcSecurityPolicy, OpcSecurityMode, OpcSecurityAlgorithm und OpcDiscoveryClient.

Standardmäßig wählt der Client aus den angebotenen Serverendpunkten den mit der **einfachsten Sicherheitskonfiguration**. Dabei fällt seine Wahl auf einen Endpunkt, dessen **OpcSecurityMode** entweder gleich *None*, *Sign* oder *SignAndEncrypt* ist. Laut OPC Foundation dient das Level der Policy eines

Endpunkts als relatives Maß der über den Endpunkt verwendeten Sicherheitsmechanismen. So ist per Definition ein Endpunkt mit einem höheren Level sicherer als ein Endpunkt mit einem niedrigeren Level. Standardmäßig ignoriert der Client das Level der Policy der Endpunkte.

1. Soll der Client ausschließlich sichere Endpunkte berücksichtigen, dann muss die **UseOnlySecureEndpoints** Eigenschaft auf den Wert *true* gestellt werden:
```
client.Security.UseOnlySecureEndpoints = true;
```
2. Soll der Client einen Endpunkt auswählen, der das höchste Policy-Level definiert, dann muss die **UseHighLevelEndpoint** Eigenschaft auf den Wert *true* gestellt werden:
```
client.Security.UseHighLevelEndpoint = true;
```
3. Soll der Client einen Endpunkt auswählen, der eine bestimmte Sicherheitskonfiguration aufweist, dann muss die **EndpointPolicy** Eigenschaft wie folgt festgelegt werden:
```
client.Security.EndpointPolicy = new OpcSecurityPolicy(
        OpcSecurityMode.None, OpcSecurityAlgorithm.Basic256);
```
4. Um die vom Server bereitgestellten Endpunkte zu untersuchen, verwenden Sie den **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.
    }
}
```

Weitere Sicherheitseinstellungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcClient, OpcClientSecurity, OpcCertificateValidationFailedEventArgs, OpcCertificateStores und OpcCertificateStoreInfo.

Ein Server sendet beim Verbindungsaufbau sein Zertifikat zur Authentifizierung an den Client. Anhand des Serverzertifikats kann der Client entscheiden, ob er wirklich eine Verbindung zu diesem Server aufbauen möchte und ihm somit vertraut.

- Zur zusätzlichen Prüfung der im Serverzertifikat hinterlegten Domänen kann die Eigenschaft **VerifyServersCertificateDomains** verwendet werden (welche standardmäßig deaktiviert ist):
```
client.Security.VerifyServersCertificateDomains = true;
```
- Soll der Client **nur vertrauenswürdige** Zertifikate akzeptieren, dann muss die standardmäßige Akzeptanz aller Zertifikate wie folgt deaktiviert werden:
```
client.Security.AutoAcceptUntrustedCertificates = false;
```
- Sobald die standardmäßige Akzeptanz aller Zertifikate deaktiviert wurde, sollte an eine benutzerdefinierte Prüfung der Zertifikate gedacht werden:

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

- Ist das Serverzertifikat als **nicht vertrauenswürdig** eingestuft, kann dieses manuell als **vertrauenswürdig** deklariert werden. Dazu muss es im TrustedPeerStore gespeichert werden:

```
// 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);
```

Konfiguration via XML

Soll der Client auch per XML konfigurierbar sein, dann besteht die Möglichkeit, die Konfiguration des Clients entweder direkt aus einer bestimmten oder aus einer beliebigen XML Datei zu laden.

Unter Einsatz einer bestimmten XML Datei muss diese den folgenden Standard XML Baum aufweisen:

```
<?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">
```

Im Falle dessen, dass eine beliebige XML Datei zur Konfiguration verwendet werden soll, muss dazu eine .config Datei erstellt werden, welche auf eine XML Datei verweist, aus der die Konfiguration für den Client geladen werden soll. Welche Einträge die .config Datei dazu enthalten und welchen Aufbau die XML Datei aufweisen muss, sehen Sie in diesem Abschnitt.

Die App.config der Anwendung anlegen und vorbereiten:

1. Eine App.config (soweit nicht schon vorhanden) zum Projekt hinzufügen
2. Das folgende *configSections* Element unterhalb des *configuration* Elements einfügen:

```
<?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. Das folgende *Opc.UaFx.Client* Element ebenso unterhalb des *configuration* Elements einfügen:

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

4. Der Wert des *FilePath* Elements kann auf einen beliebigen Dateipfad zeigen, unter dem die zu verwendende XML Konfigurationsdatei gefunden werden kann. Der hier gezeigte Wert würde so zum Beispiel auf eine Konfigurationsdatei verweisen, die neben der Anwendung liegt.
5. Die Änderungen an der App.config speichern

Die XML Konfigurationsdatei anlegen und vorbereiten:

1. Eine XML Datei mit dem in der App.config verwendeten Dateinamen anlegen und unter dem in der App.config verwendeten Pfad speichern.
2. Den folgenden Standard XML Baum für XML Konfigurationsdateien einfügen:

```
<?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. Die Änderungen an der XML Datei speichern

Auslieferung einer Clientanwendung

So bereiten Sie Ihre OPC UA Clientanwendung für den Einsatz in produktiven Umgebungen vor.

Anwendungszertifikat - *Ein konkretes Zertifikat verwenden*
Verwenden Sie für den produktiven Einsatz kein Zertifikat, das automatisch durch das Framework erzeugt wird.

Verfügen Sie bereits über ein passendes Zertifikat für Ihre Anwendung, dann können Sie Ihr PFX-basiertes Zertifikat über den **OpcCertificateManager** aus einem beliebigen Speicherort laden und der Clientinstanz zuweisen:

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

Beachten Sie, dass der Name der Anwendung, welcher im Zertifikat als „Common Name“ (CN) enthalten sein muss, mit dem Wert des *AssemblyTitle* Attributs der Anwendung übereinstimmen muss:

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

Ist das nicht der Fall, dann müssen Sie den im Zertifikat verwendeten Namen über die **ApplicationName** Eigenschaft der Clientinstanz festlegen. Wird zudem im Zertifikat der „Domain Component“ (DC) Teil verwendet, dann muss der Wert der **ApplicationUri** Eigenschaft der Anwendung den gleichen Wert aufweisen:

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

Falls Sie nicht bereits über ein passendes Zertifikat verfügen, welches Sie als Anwendungszertifikat für

Ihren Client verwenden können, sollten Sie zumindest ein selbstsigniertes (engl. self-signed) Zertifikat mittels Certificate Generator der OPC Foundation erstellen und verwenden. Der im SDK des Frameworks enthaltene Certificate Generator (Opc.Ua.CertificateGenerator.exe) wird dazu wie folgt aufgerufen:

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

Dabei legt der erste Parameter (-sp) fest, dass im aktuellen Verzeichnis das Zertifikat gespeichert werden soll. Mit dem zweiten Parameter (-an) wird der Name der Clientanwendung, die das Zertifikat als Anwendungszertifikat verwenden soll, festgelegt. Ersetzen Sie dementsprechend „MyClientAppName“ durch den Namen Ihrer Clientanwendung. Zu beachten ist dabei, dass **das Framework zur Auswahl des Anwendungszertifikats den Wert des *AssemblyTitle* Attributs verwendet und deshalb für „MyClientAppName“ der selbe Wert wie in diesem Attribut angegeben verwendet wird**.
Alternativ zum Wert im *AssemblyTitle* Attribut kann auch über die **ApplicationName** Eigenschaft der Clientinstanz der Wert festgelegt werden, der im Anwendungszertifikat verwendet wurde:

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

Wichtig ist, dass entweder der Wert des *AssemblyTitle* Attributs oder der Wert der **ApplicationName** Eigenschaft mit dem Wert des zweiten Parameters (-an) übereinstimmt. Sollten Sie noch weitere Eigenschaften des Zertifikats festlegen wollen, wie zum Beispiel die Gültigkeit in Monaten (die standardmäßig 60 Monate beträgt) oder den Namen des Unternehmens wie auch den Namen der Domänen, in denen der Client eingesetzt wird, dann rufen Sie den Generator mit dem Parameter „/?“ auf, um eine Liste von allen weiteren / möglichen Parameter(werten) zu erhalten:

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

Nachdem der Certificate Generator mit den entsprechenden Parametern aufgerufen wurde, befinden sich im aktuellen Verzeichnis die Ordner „certs“ und „private“. Ohne den Namen der Ordner und ohne den Namen der Dateien in den Ordnern zu ändern kopieren Sie die beiden Ordner in das Verzeichnis, welches Sie als Speicherort für die Anwendungszertifikate festgelegt haben. Standardmäßig ist das der Ordner im „CertificateStores“ Ordner, welcher sich neben der Anwendung befindet, der den Namen „Trusted“ trägt.

Sollten Sie den Parameter „ApplicationUri“ (-au) festgelegt haben, dann müssen Sie den gleichen Wert auf der **ApplicationUri** Eigenschaft der Clientinstanz festlegen:

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

Konfigurationsumgebung - *Alle notwendigen Dateien bei einer XML-basierten Konfiguration*

Soll die Anwendung über eine beliebige XML Datei konfigurierbar sein, welche in der App.config referenziert wird, dann muss sich die App.config im selben Verzeichnis, wie die Anwendung, befinden und den Namen der Anwendung als Präfix tragen:

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

Wird die Anwendung über eine (bestimmte) XML Datei konfiguriert, dann muss diese für die Anwendung erreichbar sein, stellen Sie auch das sicher.

Lizenzierung

Das OPC UA SDK kommt mit einer **Testlizenz die je Anwendungsstart 30 Minuten uneingeschränkt zur Client- und Serverentwicklung** verwendet werden kann. Sollte diese Einschränkung ihre Evaluationsmöglichkeiten einschränken, besteht die Möglichkeit eine alternative Testlizenz bei uns zu beantragen.

Fragen Sie einfach unseren Support (via support@traeger.de) oder lassen Sie sich gleich direkt von uns beraten und offene Fragen durch unsere Entwickler klären!

Nach Erhalt Ihres personalisierten **Lizenzschlüssels zur OPC UA Cliententwicklung** muss dieser dem Framework mitgeteilt werden. Fügen Sie hierzu die folgende Codezeile in Ihre Anwendung ein, **bevor** Sie das erste Mal auf die **OpcClient Klasse** zugreifen. Ersetzen Sie hierbei *<insert your license code here>* durch den von uns erhaltenen Lizenzschlüssel.

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

Haben Sie einen **Bundle-Lizenschlüssel zur OPC UA Client- und Serverentwicklung** bei uns erworben, muss dieser wie folgt dem Framework mitgeteilt werden:

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

Zudem erhalten Sie Informationen über die aktuell vom Framework verwendete Lizenz über die *LicenseInfo* Eigenschaft der **Opc.UaFx.Client.Licenser Klasse** für Client-Lizenzen und über die **Opc.UaFx.Licenser Klasse** für Bundle-Lizenzen. Das funktioniert wie folgt:

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

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

Beachten Sie, dass eine einmal festgelegte **Bundle-Lizenz durch die zusätzliche Angabe des Client-Lizenzschlüssels außer Kraft tritt**!

Im Laufe der Entwicklung/Evaluation ist es häufig egal, ob gerade die Testlizenz oder bereits die erworbene Lizenz verwendet wird. Sobald aber die Anwendung in den produktiven Einsatz geht, ist es ärgerlich, wenn die Anwendung während der Ausführung aufgrund einer ungültigen Lizenz nicht mehr funktioniert. Aus diesem Grund empfehlen wir den folgenden Codeausschnitt in die Clientanwendung zu implementieren und diesen zumindest beim Start der Anwendung auszuführen:

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

Weitere Informationen zur Lizenzierung, dem Erwerb oder anderen Fragen erhalten Sie direkt auf unserer Produktseite unter: opcua.traeger.de.

[1] Der *OPC UA Wrapper Server* wird automatisch vom OPC UA Client gestartet beziehungsweise ein bereits vorhandener wiederverwendet. Der *OPC UA Wrapper Server* 'verpackt' die Zugriffe des OPC UA Clients und sendet diese an den *OPC Classic Server*. Die erhaltenen Antworten des OPC Classic Servers sendet der *OPC UA Wrapper Server* als OPC UA Antworten an den *OPC UA Client* wieder zurück.

Inhaltsverzeichnis

Getestet? Du willst es? 1
Der Client Frame 2
Ein Client für OPC UA 2
Ein Client für OPC Classic 2
Werte von Node(s) 4
Werte lesen 4
Werte schreiben 5
Werte verarbeiten 5
Browsen von Nodes 7
Welche Nodes hat der Server? 7
Node für Node 'besuchen' 8
High-Speed Browsen 8
High-Speed Browsen - Details 9
Subscriptions 12
Subscriptions erstellen 12
Subscriptions filtern 14
Strukturierte Daten 15
Einfachster Zugriff 16
Namen-basierter Zugriff 16
Dynamischer Zugriff 17
Typisierter Zugriff 18
Datentypen generieren 19
Datentypen definieren 19
Datentypen mit optionalen Feldern 20
Historische Daten 21
Nodes 24
Methodenknoten 24
Dateiknoten 25
Datentypknoten 27
Datenknoten 28
Datenpunktknoten 28
Datenpunktknoten für analoge Werte 29
Ereignisse 29
Ereignisknoten 31
Ereignisknoten mit Bedingungen 32
Ereignisknoten mit Dialog-Bedingungen 33
Ereignisknoten mit Feedback-Bedingungen 35
Ereignisknoten mit Alarm-Bedingungen 35
Ereignisknoten mit diskreten Alarm-Bedingungen 36
Ereignisknoten mit Alarm-Bedingungen für Grenzwerte 37
Ereignisknoten mit Alarm-Bedingungen für ausschließliche Grenzwerte 37
Ereignisknoten mit Alarm-Bedingungen für nicht-ausschließliche Grenzwerte 38
Bearbeitung des Adressraums 39
Erstellen von Knoten 39
Löschen von Knoten 42
Erstellen von Verweisen 42
Löschen von Verweisen 43
Konfiguration des Clients 44
Allgemeine Konfiguration 44
Konfiguration des Zertifikats 46
Konfiguration der Benutzeridentität 47

Konfiguration des Serverendpunkts 47
Weitere Sicherheitseinstellungen 48
Konfiguration via XML 49
Auslieferung einer Clientanwendung 50
Lizenzierung 52