OPC UA SDK
Client + Server Development

Server Development Guide

Der Server Frame

1. Verweis zum **Opc.UaFx.Advanced** Server Namespace hinzufügen:

```
using Opc.UaFx.Server;
```

2. Eine Instanz der OpcServer Klasse mit der gewünschter Standard-Basis-Adresse:

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

3. Server starten und Clients bedienen:

```
server.Start();
```

4. Ihr Code zur Bearbeitung von Clientanfragen:

```
// Your code to process client requests.
```

5. Vor dem Beenden der Anwendung alle Sitzungen beenden und den Server herunterfahren:

```
server.Stop();
```

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

```
using (var server = new OpcServer("opc.tcp://localhost:4840/")) {
    server.Start();
    // Your code to process client requests.
}
```

Node Management

Nodes Erstellen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNode, OpcNodeId, OpcNodeManager, OpcFolderNode, OpcVariableNode, OpcDataVariableNode, OpcDataVariableNode und OpcFileNode.

Ein **OpcNode** definiert einen Datenpunkt des Servers. Dieser kann ein logischer Ordner (**OpcFolderNode**), eine Variable (**OpcVariableNode**), eine Methode (**OpcMethodNode**), eine Datei (**OpcFileNode**) und vieles mehr sein. Ein **OpcNode** wird über eine **OpcNodeId** eindeutig identifiziert. Sie besteht aus einem Wert (einem Text, einer Zahl, ...) - der eigentlichen ID - und einem Index des Namensraums (engl. Namespace), dem ein Node zugeordnet ist. Der Namensraum wird durch eine Uri (= Uniform Resource Identifier) festgelegt. Die verfügbaren Namespaces werden durch die vom Server verwendeten Node-Manager bestimmt.

Jeder Node-Manager definiert mindestens einen Namespace. Diese Namespaces werden zur Einordnung der Nodes eines Node-Managers verwendet, wodurch ein Node wieder einem bestimmten Node-Manager zugeordnet werden kann. Der Standard-Namensraum (engl. Default-Namespace) eines Node-Managers wird immer dann verwendet, wenn einem Node kein anderer Namespace zugeteilt wird. Die von einen Node-Manager definierten Nodes werden auch als *Nodes im Adressraum (engl. Address Space) des Node-Managers* bezeichnet.

Während des Startvorgangs des Servers fordert der Server seine Node-Manager dazu auf ihren Address Space, also ihre (statischen) Nodes, zu erstellen. Weitere (dynamische) Nodes können auch während der Ausführung des Servers zum Address Space eines Node-Managers hinzugefügt oder auch aus diesem wieder entfernt werden. Ein stets statischer Address Space kann auch ohne expliziten benutzerdefinierten Node-Manager erzeugt werden, indem dem Server die statischen Nodes für den Namespace *http://{host}/{path}/nodes/* direkt mitgeteilt werden. Anstelle der Nodes des statischen Address Spaces können auch benutzerdefinierte Node-Manager definiert werden.

- Einen benutzerdefinieren Address Space mit einem Root Node für den Default-Namespace *http://{host}/{path}/nodes/* erstellen:

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

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

- Einen benutzerdefinierten Node-Manager definieren:

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

- Einen benutzerdefinierten Address Space mit einem Root Node mittels benutzerdefiniertem Node-Manager erstellen:

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

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

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

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

- Einen benutzerdefinieren Node-Manager dem Server bekanntmachen:

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

Node Sichtbarkeit

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, IOpcNode, OpcNodeId, OpcNodeManager, OpcFolderNode, OpcVariableNode, OpcDataVariableNode und OpcDataVariableNode.

Nicht immer sollen alle Nodes für alle Benutzer des Servers sichtbar sein. Zur Einschränkung der Sichtbarkeit der Nodes können beliebige Kriterien über den eigenen Node-Manager angewendet werden. Häufig reicht auch die Einschränkung über die Identität und eines bestimmten Nodes. Der folgenden einfache Baum soll das veranschaulichen.

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

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

Soll nun ein Benutzer Zugriff auf alle Nodes, bis auf den „Speed“ Node zugreifen dürfen, dann kann die mögliche Implementierung der `IsNodeAccessible`-Methode wie folgt aussehen:

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

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

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

Nodes Aktualisieren

Die folgenden Typen kommen hierbei zum Einsatz: OpcStatusCode und OpcVariableNode.

Während der Ausführung des Servers müssen häufig diverse Nodes, entsprechend dem zugrundeliegenden System, aktualisiert werden. Einfache Object-, Folder- oder Method-Nodes müssen eher selten aktualisiert werden – Variablen-Nodes hingegen regelmäßig. Je nachdem, welche Informationen vorliegen können neben den eigentlichen Wert einer Variablen-Node, auch ein Zeitstempel und die Güte des Wertes im Variablen-Node hinterlegt beziehungsweise aktualisiert werden. Das folgende Beispiel skizziert kurz die Handhabung.

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

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

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

Zu beachten ist, dass erst mit dem Aufruf von `ApplyChanges` auch Client-Anwendungen über die Änderung der Node informiert werden (soweit diese eine aktive Subscription für die Node und das Value-Attribut abgeschlossen haben).

Werte von Node(s)

Werte lesen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcVariableNode, OpcDataVariableNode, OpcDataVariableNode, OpcReadAttributeValueCallback, OpcAttributeValue, OpcReadAttributeValueContext, OpcReadVariableValueCallback, OpcReadVariableValueContext und OpcVariableValue.

Ein **OpcNode** definiert seine Metadaten durch *Attribute*. Neben den im Allgemeinen immer bereitgestellten *Attributen* wie *Name*, *DisplayName* oder *Description* ist das *Value Attribut* nur auf *Variablen-Nodes* verfügbar. Die Werte der *Attribute* speichern die jeweiligen Node-Instanzen standardmässig intern. Soll der Wert aus einer anderen Datenquelle ermittelt werden, können hierzu entsprechende Callback-Methoden zur Bereitstellung der Werte definiert werden. Hierbei unterscheidet sich die Signatur der ReadVariableValue-Callback-Methode von den anderen ReadAttributeValue-Callback-Methoden. Im Falle des *Value Attributs* wird anstelle einer **OpcAttributeValue** Instanz eine **OpcVariableValue** Instanz erwartet. Diese besteht neben dem eigentlichen Wert aus einem Zeitstempel, zu dem der Wert an der Quelle des Wertes festgestellt worden ist (**SourceTimestamp**), aus Statusinformationen über die Qualität des Wertes. Zu beachten ist, dass die Read-Callbacks bei jedem Lesevorgang der Metadaten durch einen Client aufgerufen werden. Das ist der Fall bei Verwendung der Services Read und Browse.

- Den Initialwert des Value Attributs eines Variable-Nodes festlegen:
  ```
  var machineIsRunningNode = new OpcDataVariableNode<bool>("IsRunning", false);
  ```
- Den Wert des Value Attributs eines Variable-Nodes festlegen:
  ```
  machineIsRunningNode.Value = true;
  ```
- Den Wert des Description Attributs festlegen:
  ```
  machineIsRunningNode.Description = "My description";
  ```
- Alle Clients (im Falle einer aktiven Subscription) über die Attributänderungen informieren und Änderungen übernehmen:
  ```
  machineIsRunningNode.ApplyChanges(server.SystemContext);
  ```
- Den Wert des Description Attributs aus einer anderen Datenquelle als der internen ermitteln:

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

- Den Wert des Value Attributs eines Variable-Nodes aus einer anderen Datenquelle als der internen ermitteln:

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

Werte schreiben

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcVariableNode, OpcDataVariableNode, OpcDataVariableNode, OpcWriteAttributeValueCallback, OpcAttributeValue, OpcWriteAttributeValueContext, OpcWriteVariableValueCallback, OpcWriteVariableValueContext und OpcVariableValue.

Ein **OpcNode** definiert seine Metadaten durch *Attribute*. Neben den im Allgemeinen immer bereitgestellten *Attributen* wie *Name*, *DisplayName* oder *Description* ist das *Value Attribut* nur auf *Variablen-Nodes* verfügbar. Die Werte der *Attribute* speichern die jeweiligen Node-Instanzen standardmässig intern. Soll der Wert in eine andere Datenquelle gespeichert werden, können hierzu entsprechende Callback-Methoden zur Speicherung der Werte definiert werden. Hierbei unterscheidet sich die Signatur der WriteVariableValue-Callback-Methode von den anderen WriteAttributeValue-Callback-Methoden. Im Falle des *Value Attributs* wird anstelle einer **OpcAttributeValue** Instanz eine **OpcVariableValue** Instanz verwendet. Diese besteht neben dem eigentlichen Wert aus einem Zeitstempel, zu dem der Wert an der Quelle des Wertes festgestellt worden ist (**SourceTimestamp**), aus Statusinformationen über die Qualität des Wertes. Zu beachten ist, dass die Write-Callbacks bei jedem Schreibevorgang der Metadaten durch einen Client aufgerufen werden. Das ist der Fall bei Verwendung des Write Services.

- Den Initialwert des Value Attributs eines Variable-Nodes festlegen:

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

- Den Wert des Value Attributs eines Variable-Nodes festlegen:

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

- Den Wert des Description Attributs festlegen:

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

- Alle Clients (im Falle einer aktiven Subscription) über die Attributänderungen informieren und

Änderungen übernehmen:

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

- Den Wert des Description Attributs in einer anderen Datenquelle als der internen speichern:

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

- Den Wert des Value Attributs eines Variable-Nodes in einer anderen Datenquelle als der internen speichern:

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

Historische Daten

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager, IOpcNode, OpcHistoryValue, OpcHistoryModificationInfo, OpcValueCollection, OpcStatusCollection, OpcDeleteHistoryOptions, OpcReadHistoryOptions, IOpcNodeHistoryProvider und OpcNodeHistory<T>.

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. Dabei kann ein 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.

Die vom Server bereitgestellten historischen Daten können wahlweise direkt im Node-Manager des jeweiligen Nodes, über den in-memory basieren Node-Historian oder über einen benutzerdefinierten Historian verwaltet werden. Zu beachten ist, dass laut OPC UA historische Werte stets als Schlüssel ihren Zeitstempel verwenden. Dementsprechend gilt, dass unter allen historischen Werten eines Nodes ein Zeitstempel immer eindeutig ist und somit nur einen bestimmten Wert und dessen Qualitätsinformationen (= Statusinformationen) identifiziert. Die auf diese Weise gespeicherten historischen Daten werden zudem in reine historische Daten und modifizierte (engl. modified) historische Daten unterschieden. Letztere repräsentieren Datenbank-technisch eine Art Changelog (= Änderungsprotokoll). Dieses Changelog kann dazu verwendet werden, historische Daten zu verarbeiten, die vor einer Manipulation der ursprünglichen historischen Daten gültig waren. Zugleich kann über das Changelog jede Änderung der historischen Daten

nachvollzogen werden. Wird zum Beispiel ein historischer Wert ersetzt, wird der vorherige Wert in die modified Historie gespeichert. Ein historischer Wert, der aus der Historie entfernt wird, wird ebenfalls in der modified Historie gespeichert. Zusätzlich wird in der modified Historie die Art der Änderung, der Zeitstempel der Änderung und der Name des Benutzers, der die Änderung veranlasste, gespeichert.

Möchte ein Client die (modifizierten) historischen Daten eines Nodes lesen:

- Muss der entsprechende Node eine Variable-Node sein und die Aufzeichnung historischer Daten aktiviert und der Zugriff darauf freigegeben werden.
  - Wird ein **OpcNodeHistorian** verwendet, dann übernimmt dieser die Aktivierung und Freigabe der historischen Datenaufzeichnung:

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

  - Manuelle Aktivierung und Freigabe der historischen Datenaufzeichnung:

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

    machineIsRunningNode.IsHistorizing = true;
    ```

- Müssen Änderungen des *Value Attributs* des Variable-Nodes überwacht werden und in einen Speicher für die historischen Werte überführt werden.
  - Wird ein **OpcNodeHistorian** verwendet, dann kann dieser zur automatischen Aktualisierung der Historie eingestellt werden:

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

  - Zur manuellen Überwachung der Änderungen des *Value Attributs* sollte das *BeforeApplyChanges* Ereignis des Variable-Nodes abonniert werden:

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

- Müssen die historischen Daten dem Client zur Verfügung gestellt werden.
  - Wird ein **IOpcNodeHistoryProvider** wie der **OpcNodeHistorian** verwendet, dann muss dieser dem Server durch den Node-Manager mitgeteilt werden:

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

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

  - Wird ein benutzerdefinierter **IOpcNodeHistoryProvider** verwendet, dann wird dessen *ReadHistory* Methode dazu verwendet:

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

- Soll der Node-Manager sich selbst um die Historie seiner Nodes kümmern, dann muss die *ReadHistory* Methode implementiert werden:

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

Möchte ein Client die historischen Daten eines Nodes erzeugen, müssen die neuen Werte in der Historie als auch in der modifizierten Historie abgelegt werden:

- Wird ein **IOpcNodeHistoryProvider** wie der **OpcNodeHistorian** verwendet, dann muss dieser dem Server durch den Node-Manager mitgeteilt werden:

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

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

- Wird ein benutzerdefinierter **IOpcNodeHistoryProvider** verwendet, dann wird dessen *CreateHistory* Methode dazu verwendet:

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

- Soll der Nodemanager sich selbst um die Historie seiner Nodes kümmern, dann muss die *CreateHistory* Methode implementiert werden:

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

Möchte ein Client die historischen Daten eines Nodes löschen, müssen die zu löschenden Werte in die

modifizierte Historie übertragen und aus der eigentlichen Historie gelöscht werden. Sollen modifizierte Historienwerte gelöscht werden, können diese direkt aus der modifzierten Historie entfernt werden:

- Wird ein **IOpcNodeHistoryProvider** wie der **OpcNodeHistorian** verwendet, dann muss dieser dem Server durch den Node-Manager mitgeteilt werden:

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

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

- Wird ein benutzerdefinierter **IOpcNodeHistoryProvider** verwendet, dann wird eine seiner *DeleteHistory* Methoden dazu verwendet:

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

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

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

- Soll der Node-Manager sich selbst um die Historie seiner Nodes kümmern, dann müssen die *DeleteHistory* Methoden implementiert werden:

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

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

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

Möchte ein Client die historischen Daten eines Nodes ersetzen, müssen die zu ersetzenden Werte in die modifizierte Historie übertragen und in der eigentlichen Historie ersetzt werden:

- Wird ein **IOpcNodeHistoryProvider** wie der **OpcNodeHistorian** verwendet, dann muss dieser dem Server durch den Node-Manager mitgeteilt werden:

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

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

- Wird ein benutzerdefinierter **IOpcNodeHistoryProvider** verwendet, dann wird die *ReplaceHistory* Methode dazu verwendet:

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

- Soll der Nodemanager sich selbst um die Historie seiner Nodes kümmern, dann muss die *ReplaceHistory* Methode implementiert werden:

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

Möchte ein Client die historischen Daten eines Nodes erzeugen - wenn diese noch nicht existieren - oder ersetzen - wenn diese bereits existieren, im OPC UA-Sinne also aktualisieren (engl. updaten), müssen im Falle von nicht existenten Einträgen diese in die Historie und modifizierte Historie eingetragen werden und im Falle von existenten Einträgen in der Historie ersetzt und in der modifzierten Historie eingetragen werden:

- Wird ein **IOpcNodeHistoryProvider** wie der **OpcNodeHistorian** verwendet, dann muss dieser dem Server durch den Node-Manager mitgeteilt werden:

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

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

- Wird ein benutzerdefinierter **IOpcNodeHistoryProvider** verwendet, dann wird die *UpdateHistory* Methode dazu verwendet:

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

- Soll der Nodemanager sich selbst um die Historie seiner Nodes kümmern, dann muss die *UpdateHistory* Methode implementiert werden:

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

Unter Verwendung der Klasse **OpcNodeHistory<T>** können die Daten der Historie als auch die der modifizierten Historie im Speicher verwaltet werden. Neben diversen Methoden, die die üblichen Zugriffsszenarien auf historische Daten bedienen, erlauben die einzelen Konstruktoren der Klasse die Größe (= Kapazität) der Historie festzulegen. Zugleich kann die Historie bereits „vorausgeladen“ und über diverse Ereignisse überwacht werden.

Definition einer Historie abhängig von der Art der historischen Daten:

- Zum Einsatz als einfache Historie wird als Typparameter die Klasse **OpcHistoryValue** verwendet:

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

- Zum Einsatz als modifizierte Historie wird als Typparameter die Klasse **OpcModifiedHistoryValue** verwendet:

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

Bei Einsatz der Klasse **OpcNodeHistory<T>** können die üblichen History-Szenarien wie Read, Create, Delete, Replace und Update wie folgt implementiert werden:

- Szenario: **Create History**:

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

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

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

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

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

return results;
```

- Szenario: **Delete History**
  - Mittels Zeitstempel:

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

int index = ;

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

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

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

return results;
```

- Mittels Werten:

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

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

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

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

return results;
```

- Mittels Zeitspanne:

```
var results = new OpcStatusCollection();

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

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

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

        results.Add(OpcStatusCode.Good);
    }
}

return results;
```

- Szenario: **Replace History**:

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

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

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

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

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

return results;
```

- Szenario: **Update History**:

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

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

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

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

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

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

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

return results;
```

- Szenario: **Read History**:

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

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

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

Nodes

Methodenknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager, OpcMethodNode und OpcMethodContext.

Codeabschnitte, die eine für sich abgeschlossene Aufgabe erfüllen, werden in der Programmierung als Unterprogramme bezeichnet. Diese Unterprogramme werden häufig auch einfach als Funktionen beziehungsweise Methoden beschrieben. Derartige Methoden lassen sich in der OPC UA über Methodenknoten aufrufen. Zur Definition eines Methodenknotens wird die **OpcMethodNode** Klasse verwendet. Aufgerufen werden diese über den serverseitigen **Call** Service durch einen OPC UA Client.

Das Framework definiert einen Methodenknoten durch die 1:1 Umsetzung eines Funktionszeigers (in C# Delegaten) in ein Node der Kategorie *OpcNodeCategory.Method*. Hierzu wird per .NET Reflections die Struktur des Delegaten untersucht und basierend darauf der Methodenknoten mit seinen *IN* und *OUT* Argumenten definiert.

Einen Methodenknoten (engl. Method Node) im Nodemanager definieren:

1. durch eine Methode ohne Parameter:

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

2. durch eine Methode mit Parameter:

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

3. durch eine Methode mit Rückgabewert:

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

4. durch eine Methode mit Parameter und Rückgabewert:

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

5. durch eine Methode, die Zugriff auf kontextbezogene Informationen des aktuellen „Call“-Aufrufs benötigt (hierbei muss der erste Parameter vom Typen **OpcMethodNodeContext** sein):

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

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

    return statusCode;
}
```

Zusätzlich besteht die Möglichkeit, dem Framework über das **OpcArgument-Attribut** zusätzliche Informationen über die Argumente (Rückgabewerte und Parameter) einer Methode bereitzustellen. Diese Informationen werden bei der Definition der Argumente des Methodenknotens berücksichtigt und jedem Client beim Browsen des Nodes bereitgestellt. Ein derartige Definition von zusätzlichen Informationen sieht dann wie folgt aus:

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

Dateiknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager und OpcFileNode.

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 kann, 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 OPC UA beim Aufruf von Open ein Modus angeben, der die Art des geplanten Dateizugriffs vorgibt. Auch in 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.

Einen Dateiknoten (engl. File Node) im Node-Manager definieren:

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

Alle weiteren Operationen, um mit der repräsentierten Datei zu arbeiten, werden bereits durch die **OpcFileNode** Klasse bereitgestellt.

Datentypknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager, OpcNodeId, OpcDataTypeAttribute, OpcDataTypeNode und OpcEnumMemberAttribute.

In manchen Szenarios ist es notwendig die vom Server bereitgestellten Daten mit einen benutzerdefinierten Datentypen zu beschreiben. Ein solcher Datentyp kann zum Beispiel eine Enumeration sein. Je nach Eintrag (namentlich von allen anderen differenzierbar) steht für diesen ein anderer Wert oder gar eine Kombination von Werten die wiederum durch andere Einträge repräsentiert werden (können). Im letzteren Fall spricht man dann von Flag-Enumerationen. Sind die Bits für einen Enum-Eintrag bitweise in einem Flag-Enumerationswert gesetzt, dann gilt dieser auch wenn der gesamte Wert nicht genau dem Wert des Eintrages entspricht (weil im Moment auch andere Enum-Einträge Anwendung finden sollen). Die dabei gültigen (Kombinationen von) Werte müssen somit als Name-Wert-Paare unter einer bestimmten ID vom Server bereitgestellt werden, damit Lese- und Schreibzugriffe auf Nodes - die den Typen der Enumerationen verwenden - auch gültige Werte an den Server übermitteln. Damit eine benutzerdefinierte Enumeration auch als Enumeration im Adressraum des Servers publiziert wird, muss die Enumeration über das **OpcDataTypeAttribute** verfügen. Über dieses Attribut werden der Enumeration die Daten der für den Typen gültigen **OpcNodeId** festgelegt. Abschließend muss dann noch der benutzerdefinierte Datentyp über einen der Node-Manager des Servers veröffentlicht werden. Wie das im Detail aussieht, kann dem folgenden Code Beispiel entnommen werden:

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

...

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

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

Weitere Informationen über die einzelnen Enum-Einträge können über das **OpcEnumMemberAttribute** festgelegt werden. Die dabei angebotene optionale *Description* Eigenschaft kommt aber nur bei Einträge einer Flag-Enumeration zum Einsatz. Die oben dargestellte Enumeration könnte dann wie folgt aussehen:

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

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

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

Datenknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager, OpcDataVariableNode und OpcDataVariableNode.

Mit Hilfe der **OpcDataVariableNode** können einfache skalare als auch komlexe Datenstrukturen bereitgestellt werden. Diese selbstbeschreibenden Knoten stellen dabei neben dem Wert an sich auch Informationen über den für den Wert gültigen Datentypen bereit. Dazu gehört unter anderen zum Beispiel die Länge eines Arrays. Angelegt wird ein solcher Datenknoten wie folgt:

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

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

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

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

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

Datenpunktknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager, OpcDataItemNode und OpcDataItemNode.

Die durch einem OPC UA Server bereitgestellten Daten kommen häufig nicht direkt 1:1 aus dem des Servers zugrundeliegendenen System. Auch wenn diese Daten-Variablen mittels Instanzen von **OpcDataVariableNodes** bereitgestellt werden können, ist der Ursprung beziehungsweise die *Definition* - wie ein Wert eines Datenpunktes zustande kommt - für die korrekte Weiterverarbeitung und Interpretation von Interesse. Insbesondere beim Einsatz durch Dritte ist diese Information nicht nur ein Teil der Dokumentation, sondern auch ein hilfreicher Apsekt auch bei der internen Datenverarbeitung. Genau hier setzt die **OpcDataItemNode** an und stellt über die Eigenschaft *Definition* die notwenigen Information über das Zustandekommen der Werte des Datenpunktknotens bereit. Zusätzlich bietet die *ValuePrecision* Eigenschaft einen Wert der darüber Auskunft gibt, wie genau die Werte sein können. Definiert wird dieser Knoten wie folgt:

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

Generell gilt, dass der Wert der *Definition* Eigenschaft abhängig vom Hersteller ist.

Datenpunktknoten für analoge Werte

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager, OpcAnalogItemNode, OpcAnalogItemNode, OpcValueRange und OpcEngineeringUnitInfo.

Nodes vom Typen **AnalogItemType** stellen im Wesentlichen eine Spezialisierung der **OpcDataItemNode** dar. Die dabei zusätzlich angebotenen Eigenschaften erlauben es die bereitgestellten analogen Werte genauer zu spezifizieren. Dabei dient die *InstrumentRange* der Definition des Wertebereiches der von der Quelle der analogen Daten verwendet wird. Die *EngineeringUnit* dient der Klassifizierung der mit dem Wert in Verbindung stehenden Maßeinheit gemäß der UNECE Empfehlungen N° 20. Diese Empfehlungen basieren auf dem internationalen System für Maßeinheiten (engl. International System of Units, kurz SI Units). Ergänzt werden diese beiden Eigenschaften um die *EngineeringUnitRange* über die gemäß *EngineeringUnit* der 'im normalen Betrieb' gültige Wertebereich bereitgestellt werden kann. Eine solcher Knoten kann dann wie folgt im Node-Manager definiert werden:

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

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

Die dabei im Konstruktor der **OpcEngineeringUnitInfo** erwartete UnitID kann aus der UNECE Tabelle für Maßeinheiten bei der OPC Foundation entnommen werden: UNECE Maßeinheiten in OPC UA

NodeSets

NodeSets beschreiben den Inhalt des Adressraumes eines Servers in Form der XML (eXtensible Markup Language). Bestandteil der Beschreibung ist die Definition von Datentypen und deren transportierte logische (Daten-abhängige Informationen) als auch physische (im Speicher) Struktur. Zudem finden sich die Definitionen von Node-Typen wie auch konkreten Node-Instanzen in einem NodeSet. Das Stammelement „UANodeSet“ beschreibt zudem die Beziehungen (*engl. References*) zwischen den einzelnen im NodeSet definierten, sondern auch in anderen NodeSets, wie auch in der Spezifikation, definierten Nodes.

Companion Spezifikationen erweitern nicht nur die allgemeine Definition des Adressraumes, sondern liefern unter anderen auch eigene Datentypen und Nodetypen. Wodurch zu jeder Companion Spezifikation wiederum ein oder gar mehrere NodeSets existieren. Damit ein Server eine Companion Spezifikation erfüllen kann, muss der Server die in dieser Spezifikation definierten Typen und Verhaltensweisen entsprechend umsetzen.

Ein weiterer Fall, in dem NodeSets als Beschreibung des Adressraumes verwendet werden, ist die Projektierung von Steuerungen. So besteht die Möglichkeit die gesamte projektierte Konfiguration einer Steuerung aus einer Projektierungssoftware wie etwa TIA Portal zu exportieren und mit dieser den OPC UA Server einer Steuerung zu initialisieren.

Unabhängig von der Quelle eines NodeSets gilt: Soll ein NodeSet von einen Server verwendet werden, muss dieser die notwendige Logik zur Importierung und Implementierung des im NodeSets beschriebenen Adressraumes bereitstellen. Wie das funktioniert zeigen die nächsten Abschnitte.

NodeSets Importieren

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeSet, OpcNodeSetManager und OpcNodeManager.

Nodes welche in einem NodeSet beschrieben sind, können über den **OpcNodeSetManager** 1:1 importiert werden:

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

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

Beim Aufruf von **OpcNodeSetManager.Create(...)** können 1-n NodeSets angegeben werden. Der beim

Aufruf von OpcNodeSetManager.Create erstellte OpcNodeManager kümmert sich dabei um den Import der NodeSets und erzeugt somit beim Start des Servers die im NodeSets definierten Nodes innerhalb des Adressraumes des Servers.

Soll hingegen ein benutzerdefinierter NodeManager sich um den Import eines NodeSets kümmern, kann das einfach durch das Überschreiben der ImportNodes-Methode der OpcNodeManager-Klasse getan werden:

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

NodeSets Implementieren

Die folgenden Typen kommen hierbei zum Einsatz: OpcNodeManager und IOpcNode.

Häufig genügt der einfache Import eines NodeSets nicht, da die zugehörige Logik zur Anbindung des zugrundeliegenden Systems noch fehlt. Diese Logik ist zum Beispiel notwendig, um das Lesen einer Node auf das Lesen von zum Beispiel eines Wortes in einem Datenbaustein abzubilden. Gleiches gilt wiederum für das Schreiben einer Node.

Zur Implementierung dieser Logik wird die Methode **OpcNodeManager.ImplementNode** innerhalb eines benutzerdefinierten NodeManagers wie folgt überschrieben:

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

Im Falle eines UMATI-NodeSets könnte zum Beispiel die Logik zur Simulation des Status einer Lampe wie folgt implementiert werden:

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

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

Ereignisse

Ereignisse veröffentlichen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager, OpcEventSeverity und OpcText.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Ereignisse informieren einen Abonnenten über Abläufe, Zustände und Systemspezifische Begebenheiten. Derartige Informationen können Interessenten direkt über **globale Ereignisse** zugestellt werden. Ein globales Ereignis kann entweder durch einen **OpcNodeManager**- oder über eine **OpcServer**-Instanz ausgelöst und versendet werden. Hierzu bietet das Framework diverse Methodenüberladungen der *ReportEvent(...)*-Methode. Um ein globales Ereignis unter Einsatz einer **OpcServer**-Instanz auszulösen stehen die folgenden Möglichkeiten zur Verfügung:

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

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

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

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

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

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

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

Die selbigen Methodenüberladungen finden sich auch als Instanz-Methoden eine **OpcNodeManager**-Instanz.

Ereignisknoten

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcEventNode. Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Es ist nicht immer zweckmäßig Ereignisse durchwegs global über den Server an alle Abonnenten zu senden. 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. Die Basisklasse aller Ereignisknoten stellt dabei die Klasse **OpcEventNode** dar. Unter Einsatz dieser ist es möglich einfache Ereignisse, der Art wie sie im Abschnitt 'Bereitstellen von Ereignissen' gezeigt wurden, in Form von lokalen Ereignissen bereitzustellen. Da es sich dabei um einen Knoten handelt, muss der Ereignisknoten (**OpcEventNode**) zuvor im **OpcNodeManager** wie jeder andere Knoten angelegt werden:

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

Damit ein Ereignis nun von diesem Ereignisknoten gesendet werden kann, muss dieser noch als 'Benachrichtiger' (engl. Notifier) definiert werden. Dazu wird der Ereignisknoten bei jedem Knoten als 'Benachrichtiger' eingetragen, über den ein Abonnement das lokale Ereignis empfangen können soll. Das funktioniert wie folgt:

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

Bevor nun ein Ereignis ausgelöst wird, müssen noch alle für das Ereignis relevanten Informationen auf dem Ereignisknoten eingetragen werden. Welche Informationen dabei geändert und wie festgelegt werden hängt vom einzelnen Anwendungsfall ab. Im Allgemeinen funktioniert das wie folgt:

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

Zusätzlich bietet der Ereignisknoten **OpcEventNode** noch weitere Eigenschaften:

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

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

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

Nach erfolgter Konfiguration des zu erzeugenden Ereignisses muss nur noch die *ReportEvent(...)*-Methode der **OpcEventNode**-Instanz aufgerufen werden:

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

Diese führt automatisch einen Aufruf der *ApplyChanges(...)*-Methode auf der Node aus, erzeugt eine Momentaufnahme (engl. Snapshot) der Node und sendet diese an alle Abonnementen. Nach Aufruf der *ReportEvent(...)*-Methode können die Eigenschaften der **OpcEventNode** nach Belieben weiter geändert werden.

Nachdem ein Abonnent 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 ein Server Abonnenten nachträglich über vergangene Ereignisse informieren, dann können diese vom Server auf Anfrage vom Abonnenten wie folgt bereitgestellt werden:

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

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

Anzumerken ist an dieser Stelle, dass jeder Knoten unter dem ein Ereignisknoten als 'Benachrichtiger' eingetragen wurde separat einen solchen Callback festlegen muss. **Generell steht der Server aber nicht in der Pflicht vergangene Ereignisse bereitzustellen.** Darüberhinaus besteht jederzeit die Möglichkeit mit Hilfe der *CreateEvent(...)*-Methode der **OpcEventNode** einen Snapshot des Knotens anzulegen, diesen zwischenzuspeichern und die zwischengespeicherten Snapshots beim Aufruf des *QueryEventsCallback*'s bereitzustellen.

Ereignisknoten mit Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcConditionNode. Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Spezialisierung der **OpcEventNode** (vorgestellt im Abschnitt 'Bereitstellen von Ereignisknoten') ist die Klasse **OpcConditionNode**. Sie dient der Definition von Ereignissen an die bestimmte Bedingungen geknüpft sind. Nur im Falle, dass die dem Ereignisknoten zugesprochene Bedingung zutrifft, sollte ein Ereignis ausgelöst werden. Zu dem Knoten zugehörige Informationen gehören auch Informationen über den Zustand dieser Bedingung, wie auch Informationen die an die Auswertung der Bedingung geknüpft sind. Da diese Informationen je nach Szenario unterschiedlich komplex sein können stellt die **OpcConditionNode** die Basisklasse aller Ereignisknoten dar an denen eine Bedingung geknüpft ist. Erzeugt wird ein solcher Knoten wie ein **OpcEventNode**. Im Folgenden sollen deshalb nur die spezifischen weiteren Eigenschaften gezeigt werden:

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

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

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

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

Mit Hilfe der Methode *AddComment(...)* und dem gleichnamigen Kindknoten kann die *Comment*-Eigenschaft des Knotens geändert werden. Das Ergebnis aus der Änderung kann über die folgenden Eigenschaften ausgewertet werden:

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

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

Soll derselbe Ereignisknoten mehrgleisig bearbeitbar sein, dann kann ein neuer Ereigniszweig (engl. event branch) eröffnet werden. Hierzu kann die *CreateBranch(...)*-Methode des Knotens verwendet werden. Der dabei für den Branch eindeutige Schlüssel wird in der *BranchId*-Eigenschaft gespeichert. Das folgende Snippet zeigt die für Branches wichtigsten Bestandteile einer **OpcConditionNode**:

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

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

...

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

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

Ereignisknoten mit Dialog-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcDialogConditionNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Spezialisierung der **OpcConditionNode** ist die **OpcDialogConditionNode**. Die mit diesem Knoten 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 wird (*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. Wurde ein Knoten wie gehabt erstellt, können die entsprechenden Eigenschaften je nach Szenario definiert werden:

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

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

Ein durch einen Abonnenten beantwortete Dialog-Bedingung wird dann mittels *RespondCallback* des Knotens wie folgt behandelt.

```
outOfMaterial.RespondCallback = this.HandleOutOfMaterialResponse;

...

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

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

    return OpcStatusCode.Good;
}
```

Ereignisknoten mit Feedback-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcAcknowledgeableConditionNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Basierend auf der **OpcConditionNode** stellt die **OpcAcknowledgeableConditionNode** 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 stellt der Knoten zwei Kindknoten *Confirm* und *Acknowledge* 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. Erstellt wird

ein solcher Knoten wie bereits bekannt:

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

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

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

Während der weiteren Prozessabläufe kann ein eingegangenes Feedback mittels *IsAcked*- und *IsConfirmed*-Eigenschaft des Knotens geprüft werden:

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

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

Zu beachten ist, 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 sollte ein Server zumindest von der *Acknowledge*-Methode Gebrauch machen, da diese von der Spezifikation als „stärker“ definiert ist.

Ereignisknoten mit Alarm-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcAlarmConditionNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Die in der OPC UA wohl wichtigste Implementierung der **OpcAcknowledgeableConditionNode** ist die **OpcAlarmConditionNode**. Mit Hilfe der **OpcAlarmConditionNode** ist es möglich Ereignisknoten zu definieren deren Verhalten mit einem Nachttischwecker vergleichbar ist. Dementsprechend wird dieser Knoten aktiv (siehe *IsActive*-Eigenschaft), wenn die mit ihm 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*-Eigenschaft). Alternativ kann ein zurückstellter Alarm auch wieder „vorgestellt“ (engl. unshelved) werden (siehe *Shelving*-Eigenschaft). Ein Beispiel für die API der **OpcAlarmConditionNode** zeigt der folgende Code:

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

...

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

...

if (overheating.IsActive)
    CancelJob();

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

Ereignisknoten mit diskreten Alarm-Bedingungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager, OpcDiscreteAlarmNode, OpcOffNormalAlarmNode und OpcTripAlarmNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Ausgehend von der **OpcAlarmConditionNode** 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 **OpcDiscreteAlarmNode**. Sie definiert einen Alarmzustandsknoten, 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). Soll ein Alarm einen diskreten Zustand darstellen, der als nicht normal angesehen wird, sollte der Einsatz der **OpcOffNormalAlarmNode** oder einer ihrer Unterklassen in Betracht gezogen werden. Ausgehend von dieser Alarmklasse bietet das Framework eine weitere Konkretisierung mit der **OpcTripAlarmNode**. Der **OpcTripAlarmNode** wird aktiv, wenn zum Beispiel an einem überwachten Gerät ein anomaler Fehler auftritt, z.B. wenn der Motor aufgrund einer Überlastung abgeschaltet wird. Erstellt werden die eben genannten Knoten wie folgt:

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

Ereignisknoten mit Alarm-Bedingungen für Grenzwerte

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcLimitAlarmNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Sollen Prozessspezifische Grenzwerte geprüft und der Ausgang der Prüfung bei Grenzwertüberschreitungen / -unterschreitungen publiziert werden, dann bietet die **OpcLimitAlarmNode** Klasse den zentralen Anlaufpunkt zum Einstieg in die Klassen der Grenzwert-Alarme (engl. limit alarms).

Mit Hilfe dieser Klasse können Grenzwerte in bis zu vier Stufen unterteilt werden. Zur Differenzierung dieser werden sie als LowLow, Low, High und HighHigh bezeichnet (genannt in der Reihenfolge ihrer metrischen Ordnung). Per Definition ist es nicht notwendig alle Grenzwerte zu definieren. Aus diesem Grund bietet die Klasse die Möglichkeit die gewünschten Grenzen von Anfang festzulegen:

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

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

Ereignisknoten mit Alarm-Bedingungen für ausschließliche Grenzwerte

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcNodeManager und OpcExclusiveLimitAlarmNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm & Conditions.

Eine Unterklasse der **OpcLimitAlarmNode** ist die Klasse **OpcExclusiveLimitAlarmNode**. 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. Die dabei verletzte Grenze wird mit der *Limit*-Eigenschaft des Knotens beschrieben.

Im Rahmen der OPC UA gibt es drei weitere Spezialisierungen der **OpcExclusiveLimitAlarmNode**.

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

OpcExclusiveLevelAlarmNode
Diese Art von Alarm wird normalerweise verwendet, um zu melden, 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.

OpcExclusiveRateOfChangeAlarmNode
Diese Art von Alarm wird üblicherweise 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: OpcServer, OpcNodeManager und OpcNonExclusiveLimitAlarmNode.
Dieser Abschnitt beschreibt einen Teil der API für die Themen bezüglich: Alarm & Events, Alarm &

Conditions.

Eine Unterklasse der **OpcLimitAlarmNode** ist die Klasse **OpcNonExclusiveLimitAlarmNode**. 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* des Knotens geprüft werden.

Im Rahmen der OPC UA gibt es drei weitere Spezialisierungen der **OpcNonExclusiveLimitAlarmNode**.

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

OpcNonExclusiveLevelAlarmNode
Diese Art von Alarm wird normalerweise verwendet, um zu melden, 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.

OpcNonExclusiveRateOfChangeAlarmNode
Diese Art von Alarm wird üblicherweise 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.

Überwachung von Anfrage- und Antwort-Nachrichten

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcRequestValidatingEventArgs, OpcRequestValidatingEventHandler, OpcRequestProcessingEventArgs, OpcRequestProcessingEventHandler, OpcRequestProcessedEventArgs, OpcRequestProcessedEventHandler, OpcRequestValidatedEventArgs, OpcRequestValidatedEventHandler, IOpcServiceRequest und IOpcServiceResponse.

Die von Clients an einen Server gesendeten Anfragen werden vom Server als Instanzen der IOpcServiceRequest Schnittstelle verarbeitet, validiert und mittels Instanzen der IOpcServiceResponse Schnittstelle beantwortet. Die dabei vom Server empfangenen Anfragen können über die Ereignisse RequestProcessing, RequestValidating, RequestValidated und RequestProcessed über benutzerdefinierte Methoden zusätzlich überwacht, protokolliert, gelenkt oder gar verweigert werden. Dies ist besonders dann in Situationen sinnvoll, wenn die vom Framework bereitgestellten Mechanismen nicht den projektspezifischen Ansprüchen, insbesondere bestimmter Restriktionen, nicht ausreichend sind. Der Ablauf der Verarbeitung bis hin zur Beantwortung (= mittels Antworten, engl. Responses) von Anfragen (engl. Requests) durchläuft dabei die folgenden Schritte:

1. Empfang der Rohdaten einer Anfrage (Protokollebene des Frameworks)
2. Deserialisierung der Rohdaten zu einer Anfrage (Nachrichtenebene des Frameworks)
3. Vorverarbeitung der Anfrage: **RequestProcessing Ereignis**
4. Zuteilung der Anfrage zum entsprechenden Dienst (Dienstebene des Frameworks)
5. Validierung der Anfrage

    1. Standardvalidierungen (Session, Identity, ...)
    2. benutzerdefinierte Validierung: **RequestValidating Ereignis**
    3. abschließende Validierung (Prüfung der benutzerdefinierten Validierung)
    4. benutzerdefinierter Abschluss der Validierung: **RequestValidated Ereignis**
6. Verarbeitung der Anfrage (Anwendungsebene des Frameworks)
7. Erzeugen der Antwort über den entsprechenden Dienst (Dienstebene des Frameworks)
8. Nachverarbeitung der Anfrage und deren Antwort: **RequestProcessed Ereignis**
9. Serialisierung der Antwort zu Rohdaten (Nachrichtenebene des Frameworks)
10. Senden der Rohdaten der Antwort (Protokollebene des Frameworks)

Die unter den Punkten 3., 5.2, 5.4 und 8. genannten Ereignisse bieten dem Entwickler des Servers die Möglichkeit über den Ablauf der Anfrageverarbeitung über benutzerdefinierten Code zum entsprechenden Zeitpunkt zu überwachen oder zu beeinflussen. Direkt nach Empfang und der Aufbereitung der Nutzdaten in Form einer IOpcServiceRequest Instanz wird der benutzerdefinierte Code des RequestProcessing Ereignisses ausgeführt. Die hierbei bereitgestellten Informationen dienen der primären Diagnose des Nachrichtenverkehrs zwischen Client und Server. Ein hier eingetragener Eventhandler sollte keine Exception auslösen:

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

// ...

server.RequestProcessing += HandleRequestProcessing;
```

Der in den OpcRequestProcessingEventArgs bereitgestellte Kontext entspricht dabei stets einer Instanz der OpcContext Klasse, welche nur die allgemeine Umgebung der Nachrichtenverarbeitung beschreibt. Die in diesem Eventhandler bereitgestellten Informationen werden im darauf folgenden RequestValidating Ereignis zusätzlich um Informationen über die Sitzung (engl. Session) und Identität (engl. Identity) ergänzt. Im Falle von Anfragen, die eine Session voraussetzen, handelt es sich beim bereitgestellten OpcContext Objekt um die Spezialisierung OpcOperationContext. Über den OpcOperationContext können dann zusätzliche Sitzungsbezogene Validierungen durchgeführt werden:

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

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

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

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

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

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

// ...

server.RequestValidating += HandleRequestValidating;
```

Das gezeigte Beispiel beschränkt die Anzahl der „AddNodes“-Anfragen je Sitzung auf 100 „AddNode“-Befehle. Jede weitere Anfrage wird nach Erreichen der Beschränkung verweigert. Dies geschieht durch Setzen der Cancel Eigenschaft der Argumente des Ereignisses auf den Wert „true“, wodurch automatisch der Code der Result Eigenschaft der Argumente des Ereignisses auf den Wert „BadNotSupported“ festgelegt wird. Es besteht auch die Möglichkeit die Anfrage durch das (zusätzliche) Setzen eines „Bad“-Codes abzubrechen. Ein über das RequestValidating Ereignis eingetragener Eventhandler darf eine Exception auslösen. Wird jedoch eine Exception im Eventhandler ausgelöst oder die Eigenschaft Cancel auf den Wert „true“ beziehungsweise die Result Eigenschaft der Argumente des Ereignisses auf einen „Bad“-Code gesetzt, so werden die Eventhandler des RequestValidated Ereignis nicht ausgeführt (was den aus dem .NET Framework bekannten Validating-Validated-Pattern entspricht). Wird hingegen die Anfrage nicht abgebrochen, werden die Eventhandler des RequestValidated Ereignisses ausgeführt:

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

// ...

server.RequestValidated += HandleRequestValidated;
```

Auch hier gilt wie beim RequestProcessing Ereignis, dass die bereitgestellten Informationen der primären Diagnose des Nachrichtenverkehrs zwischen Client und Server dienen. Der Nutzen des Ereignisses besteht darin, dass nur nach Aufruf des Ereignisses der Server auch versucht die Anfrage zu bearbeiten und zu beantworten. Ein hier eingetragener Eventhandler sollte keine Exception auslösen. Nach Abschluss der vom Server durchgeführten Verarbeitung der Anfrage wird schlussendlich die Anfrage mit der resultierenden Ergebnisse beantwortet. Die dabei erzeuge Antwort kann zusammen mit der Anfrage im RequestProcessed Ereignis ausgewertet werden:

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

// ...

server.RequestProcessed += HandleRequestProcessed;
```

Wie im obigen Beispiel gezeigt stellen die Argumente des Ereignisses zusätzlich noch Informationen über eine gegebenenfalls aufgetretenen Ausnahme (engl. Exception) zur Verfügung. Ein hier eingetragener Eventhandler sollte keine Exception auslösen.

Konfiguration des Servers

Allgemeine Konfiguration

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

In allen hier gezeigten Codeausschnitten wird stets der Server über den Code konfiguriert (wenn nicht mit der Standardkonfiguration des Servers gearbeitet wird). Zentrale Anlaufstelle für die Konfiguration der Serveranwendung ist die **OpcServer** Instanz. Alle Einstellungen zum Thema Sicherheit finden sich als Instanz der **OpcServerSecurity** Klasse über die *Security* Eigenschaft des Servers. Alle Einstellungen zum Thema Zertifikat Speicher finden sich als Instanz der **OpcCertificateStores** Klasse über die *CertificateStores* Eigenschaft des Servers.

Soll der Server auch per XML konfigurierbar sein, dann besteht die Möglichkeit die Konfiguration des Servers entweder direkt aus einer bestimmten oder aus einer beliebigen XML Datei zu laden. Welche Schritte dazu nötig sind, sehen Sie im Abschnitt „Vorbereiten der Serverkonfiguration via XML“.

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

- Laden der Konfigurationsdatei über die App.config

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

- Laden der Konfigurationsdatei über den Pfad zur XML Datei

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

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

- Konfiguration der Anwendung
  - via Code:

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

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

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

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

- Konfiguration der Zertifikatspeicher
  - via Code:

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

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

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

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

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

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

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

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

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

Konfiguration des Zertifikats

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcCertificateManager, OpcServerSecurity, OpcCertificateStores und OpcCertificateStoreInfo.

Empfohlen werden Zertifikate vom Typen .der, .pem, **.pfx** und **.p12**. Soll der Server einen sicheren Endpunkt bereitstellen (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("MyServerCertificate.pfx");
```
2. Ein **neues Zertifikat** kann wie folgt (im Speicher) erzeugt werden:
```
var certificate = OpcCertificateManager.CreateCertificate(server);
```
3. Gespeichert werden kann das Zertifikat unter einem beliebigen Pfad über:
```
OpcCertificateManager.SaveCertificate("MyServerCertificate.pfx", certificate);
```
4. Das Serverzertifikat festlegen:
```
server.Certificate = certificate;
```
5. Das Zertifikat muss im Zertifikatstore für **Anwendungszertifikate** (= ApplicationStore) gespeichert sein:
```
if (!server.CertificateStores.ApplicationStore.Contains(certificate))
    server.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 Server nur das angegebene Zertifikat verwendet, muss diese Funktion deaktiviert werden. Zum **Deaktivieren der Funktion** die Eigenschaft **AutoCreateCertificate** auf den Wert *false* stellen:
```
server.CertificateStores.AutoCreateCertificate = false;
```

Konfiguration der Benutzeridentitäten

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcUserIdentity, OpcServerIdentity, OpcCertificateIdentity, OpcServerSecurity, OpcAccessControlList, OpcAnonymousAcl, OpcUserNameAcl, OpcCertificateAcl, OpcAccessControlEntry, OpcOperationType, OpcRequestType und OpcAccessControlMode.

Ein Server erlaubt standardmässig den Zugriff auch ohne Angabe einer konkreten Benutzeridentität. Diese Art von Benutzerauthentifizierung wird als anonyme Authentifizierung bezeichnet. Wird hingegen eine Benutzeridentität angegeben, dann muss diese dem Server bekannt sein, damit mit dieser Identität auf den Server zugegriffen werden darf. Soll zum Beispiel ein Benuztername-Passwort-Paar oder ein Zertifikat als Ausweis zur Benutzerauthenifizierung verwendet werden können, müssen die entsprechenden Zugriffskontrolllisten (engl. Access Control List - ACL) konfiguriert und aktiviert werden. Zur Konfiguration der Kontrolllisten gehört die Konfiguration der Zugriffskontrolleinträge (engl. Access Control Entries - ACE). Diese werden durch ein Prinzipal mit einer bestimmten Identität (Benutzername-Passwort oder Zertifikat) definiert und in die Listen eingetragen.

- Deaktivierung der anonymen Zugriffskontrollliste:

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

- Konfiguration der **Benutzername-Passwort**-Paar-basierten Zugriffskontrollliste:

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

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

- Konfiguration der **Zertifikat**-basierten Zugriffskontrollliste:

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

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

Alle bisher durch das Framework definierten Zugriffskontrolllisten verwenden als Zugriffskontrollmodus (engl. Access Control Mode) den Modus „Whitelist“. Unter diesem Modus besitzt allein durch die Definition eines Access Control Entries jeder Entry auf alle möglichen Arten (engl. Types) von Anfragen (engl. Requests) Zugriff, auch ohne dem Entry den Zugriff dafür explizit zu erteilen (engl. allow). Dementsprechend müssen den Entries alle nicht erlaubten Aktionen entzogen (engl. deny) werden. Die jeweiligen erlaubten und verbotenen Operationen können direkt auf dem Entry festgelegt werden, das nach dem Eintrag in die ACL zur Verfügung gestellt wird.

1. Ein Access Control Entry merken:

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

2. Dem Access Control Entry zwei Rechte entziehen:

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

3. Ein zuvor entzogenes Recht wieder erteilen:

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

Konfiguration der Serverendpunkte

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcServerSecurity, OpcSecurityPolicy, OpcSecurityMode und OpcSecurityAlgorithm.

Die Endpunkte des Servers werden durch das Kreuzprodukt der verwendeten Basis-Adressen und der konfigurierten Sicherheitsstrategien für Endpunkte definiert. Die Basis-Adressen setzen sich dabei aus den unterstützten Schema-Port-Paaren und dem Host (IP Adresse oder DNS Name) zusammen, wobei mehrere Schemen (möglich sind „http“, „https“, „opc.tcp“, „net.tcp“ und „net.pipe“) zum Datenaustausch auf unterschiedlichen Ports festgelegt werden können. Standardmäßig verwendet der Server keine spezielle Strategie (engl. policy), um einen sicheren Endpunkt (engl. endpoint) bereitzustellen. Dementsprechend gibt es genauso viele Endpunkte, wie es Basis-Adressen gibt. Definiert ein Server genau eine Basis-Adresse, gibt es nur einen Endpunkt mit eben dieser Basis-Adresse und der Sicherheitsstrategie mit dem

Modus *None*. Werden aber n verschiedene Basis-Adressen festgelegt, dann existieren ebenso nur n verschiedene Endpunkte mit genau derselben Sicherheitsstrategie. Auch wenn nur eine spezielle Sicherheitsstrategie festgelegt wird, existieren weiterhin nur n verschiedene Endpunkte mit genau dieser Sicherheitsstrategie. Gibt es hingegen m verschiedene Sicherheitsstrategien (s_1, s_2, s_3, ..., s_m), dann gibt es mit n verschiedenen Basis-Adressen (b_1, b_2, ..., b_n) die Endpunkte, die durch eine paarweise Verknüpfung aus Strategie und Basis-Adresse entstehen (s_1+b_1, s_1+b_2, ..., s_1+b_n, s_2+b_1, s_2+b_2, ..., s_2+b_n, s_3+b_1, s_3+b_2, ..., s_3+b_n, s_m+b_n, ...).

Zusätzlich zum Modus (= Security-Mode) der zu verwendenden Sicherung der Kommunikation definiert eine Endpoint-Policy einen Security-Algorithmus sowie einen Level. Laut OPC Foundation dient der 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 (zu beachten ist, dass das lediglich eine Richtlinie ist, die niemand weder prüft, noch durchsetzt).

Werden nun zum Beispiel zwei Sicherheitsstrategien (engl. Security-Policies) verfolgt, dann könnten diese wie folgt definiert sein:

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

Werden weiter zum Beispiel drei Basis-Adressen (engl. Base-Addresses) wie folgt für verschiedene Schemen festgelegt:

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

So ergeben sich daraus durch das Kreuzprodukt die folgenden Endpunktbeschreibungen:

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

Zur Konfiguration der (primären) Basis-Adresse kann entweder der Konstruktor der **OpcServer** Klasse oder die **Address** Eigenschaft einer **OpcServer** Instanz verwendet werden:

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

Soll der Server weitere Basis-Adressen unterstützen, dann können diese mit den Methoden **RegisterAddress** und **UnregisterAddress** verwaltet werden. Die in der Summe vom Server verwendeten (also registrierten) Basis-Adressen können über die **Addresses** Eigenschaft abgerufen werden. Wurde zuvor der Wert der **Address** Eigenschaft nicht festgelegt, dann wird die erste Adresse, die mittels **RegisterAddress** definiert wird, für die **Address** Eigenschaft verwendet.

Zwei weitere Basis-Adressen definieren:

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

Zwei Basis-Adressen vom Server entfernen (= deregistrieren, engl. unregister), sodass sich auch die „Haupt“-Basis-Adresse ändert:

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

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

Werden alle Adressen der **Addresses** Eigenschaft deregistriert, dann ist auch der Wert der **Address** Eigenschaft nicht festgelegt.

Definition einer sicheren Sicherheitsstrategie für die Endpunkte des Servers:

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

Durch die Definition einer konkreten Sicherheitsstrategie für Endpunkte geht die standardmäßig verwendete Strategie mit dem Modus *None* verloren. Damit auch diese (nicht für den produktiven Einsatz zu empfehlende) Strategie vom Server unterstützt wird, muss sie explizit in die Liste der Endpunkt-Strategien eingetragen werden:

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

Weiterer Sicherheitseinstellungen

Die folgenden Typen kommen hierbei zum Einsatz: OpcServer, OpcServerSecurity, OpcCertificateValidationFailedEventArgs, OpcCertificateStores und OpcCertificateStoreInfo.

Ein Client sendet beim Verbindungsaufbau sein Zertifikat zur Authentifizierung an den Server. Anhand des Clientzertifikats kann der Server entscheiden, ob er eine Verbindung mit dem Client zulassen möchte und ihm somit vertraut.

- Soll der Server **nur vertrauenswürdige** Zertifikate akzeptieren, dann muss die standardmäßige Akzeptanz aller Zertifikate wie folgt deaktiviert werden:
  ```
  server.Security.AutoAcceptUntrustedCertificates = false;
  ```
- Sobald die standardmäßige Akzeptanz aller Zertifikate deaktiviert wurde, sollte an eine benutzerdefinierte Prüfung der Zertifikate gedacht werden:
  ```
  server.CertificateValidationFailed += HandleCertificateValidationFailed;
  ...
  private void HandleCertificateValidationFailed(object sender,
  OpcCertificateValidationFailedEventArgs e)
  {
      if (e.Certificate.SerialNumber == "...")
          e.Accept = true;
  }
  ```
- Ist das Clientzertifikat 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 OpcServer.
var server = (OpcServer)sender;

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

Konfiguration via XML

Soll der Server auch per XML konfigurierbar sein, dann besteht die Möglichkeit, die Konfiguration des Servers 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">
</OpcApplicationConfiguration>
```

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 Server 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:

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

3. Das folgende *Opc.UaFx.Server* Element ebenso unterhalb des *configuration* Elements eingefügen:

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

3. Die Änderungen an der XML Datei speichern

Auslieferung einer Serveranwendung

So bereiten Sie Ihre OPC UA Serveranwendung 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 Serverinstanz zuweisen:

```
var certificate = OpcCertificateManager.LoadCertificate("MyServerCertificate.pfx");
server.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 Serverinstanz festlegen. Wird zudem im Zertifikat der „Domain Component“ (DC) Teil verwendet, dann muss der Wert der **ApplicationUri** Eigenschaft der Anwendung den gleichen Wert aufweisen:

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

Falls Sie nicht bereits über ein passendes Zertifikat verfügen, welches Sie als Anwendungszertifikat für Ihren Server 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 MyServerAppName
```

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 Serveranwendung, die das Zertifikat als Anwendungszertifikat verwenden soll, festgelegt. Ersetzen Sie dementsprechend „MyServerAppName“ durch den Namen Ihrer Serveranwendung. Zu beachten ist dabei, dass **das Framework zur Auswahl des Anwendungszertifikats den Wert des *AssemblyTitle* Attributs verwendet und deshalb für „MyServerAppName“ der selbige Wert wie in diesem Attribut angegeben verwendet wird**.
Alternativ zum Wert im *AssemblyTitle* Attribut kann auch über die **ApplicationName** Eigenschaft der Serverinstanz der Wert festgelegt werden, der im Anwendungszertifikat verwendet wurde:

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

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 Server 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, der im „CertificateStores“ Ordner den Namen „Trusted“ trägt, welcher sich neben der Anwendung befindet.

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

```
server.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:

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

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

Systemkonfiguration - *Administrativer Setup*

Führen Sie die Anwendung auf dem Zielsystem einmalig mit Administratorrechten aus, um sicherzustellen, dass der Server auf Netzwerkressourcen zugreifen darf. Das ist zum Beispiel dann notwendig, wenn der Server eine Basis-Adresse mit dem Schema „http“ oder „https“ verwenden soll.

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.

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

durch den von uns erhaltenen Lizenzschlüssel.

```
Opc.UaFx.Server.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.Server.Licenser Klasse** für Server-Lizenzen und über die **Opc.UaFx.Licenser Klasse** für Bundle-Lizenzen. Das funktioniert wie folgt:

```
ILicenseInfo license = Opc.UaFx.Server.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 Server-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 Serveranwendung zu implementieren und diesen zumindest beim Start der Anwendung auszuführen:

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

Inhaltsverzeichnis