OPC UA Framework Advanced

OPC UA Client and Server development made easy

Successful with only a few lines of code

Book - The whole Manual as eBook

Download

Find Downloads on your distributors website.

Features

OPC UA Features

- Data Access (DA)
- Methods
- Events
- Alarm & Conditions
- Historian Data (HDA)
- FileType (files can be directly embedded and approached)
- Extension of the OPC UA Stack from the OPC-Foundation to comfortable and mature functionality
- no difficult “code joggling” as necessary in direct use of the Foundation Stack
- intuitive and easy usability of the framework guarantee the best possible benefit
- Client / Server application with only a few lines of code
- support of the Microsoft OPC UA Stack in the nearby future
- Configuration
    - directly in the code
    - XML-based
    - through external file
- Access / Policies
    - integrateable user management
    - User Token Policy configuration via ACL (Access Control Lists)
    - allround authentification possibilities
    - User / Password
    - Certificates
    - White / Blacklisting of user rights
    - automatic generation of Client- / Server certificates
    - free choice of certificate storage (Cerificate Stores): file system, operating system or application
- Nodes
    - “easy to use” Node management
    - user-dependent access to Nodes possible
    - Comfort Browsing
    - Callbacks / events triggered by Node access (read / write / subscribe)
- unlimited number of connections

Requirements

Operating system

- Windows 32 / 64 Bit with .NET Framework (minimum 4.0)
- also Linux / macOS with support of the Microsoft Stack

Languages

- C#
- VB.NET

OPC UA Client

OPC UA Client Development Guide

Example C# Code OPC UA Client

```
namespace Client
{
    using System;
    using System.Threading;

    using Opc.UaFx.Client;

    public class Program
    {
        public static void Main()
        {
            using (var client = new OpcClient("opc.tcp://localhost:4840/")) {
                client.Connect();
                Console.WriteLine("OPC UA Client is connected...");

                using (new Timer(ReadTemperature, client, TimeSpan.Zero,
TimeSpan.FromSeconds(1)))
                    Console.ReadKey();
            }
        }

        private static void ReadTemperature(object state)
        {
            var client = (OpcClient)state;

            var temperature = client.ReadNode("opc.node://localhost/nodes/#Temperature");
            Console.WriteLine($"Current Temperature is {temperature.Value} °C");
        }
    }
}
```

OPC UA Server

OPC UA Server Development Guide

Excample C# Code OPC UA Server

```
namespace Server
{
    using System;
    using System.Threading;

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

    public static class Program
    {
        private static OpcDataVariableNode<int> temperatureNode;

        public static void Main()
        {
            temperatureNode = new OpcDataVariableNode<int>("Temperature");

            using (var server = new OpcServer("opc.tcp://localhost:4840/", temperatureNode))
{
                server.Start();
                Console.Write("OPC UA Server is running...");

                using (new Timer(UpdateTemperature, server, TimeSpan.Zero,
TimeSpan.FromSeconds(1)))
                    Console.ReadKey();
            }
        }

        private static void UpdateTemperature(object state)
        {
            var server = (OpcServer)state;

            temperatureNode.Value = DateTime.Now.Second / 3;
            temperatureNode.ApplyChanges(server.SystemContext);
        }
    }
}
```

Class Library

Find the complete Class Library in Evaluation Paket as download on your distributors web site.

General

Terms

OPC UA

OPC UA stands for OPC Unified Architecture, shortened OPC UA. Contrasting to the predecessor OPC, OPC UA especially differenciates itself through the ability to not only transport machine data (meassurements, parameters etc.), but also describe the data semantically in order for machines to read it. OPC UA means: **O**peness **P**roductivity **C**onnectivity **U**nified **A**rchitechture.

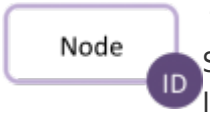
Node

The “Node” is the most basic element of the OPC UA. Nearly every element is “reduced” to one “Node”, so to say. Hearby the Nodes stand within direct relation to each other.

The Wikipedia definition about the OPC Unified Architecture contains a fitting description for the term “Node”:
“The OPC information model is a so-called Full Mesh Network based on nodes. Nodes hold process data as well as all other types of metadata.” Source: wikipedia.org/wiki/OPC_Unified_Architecture

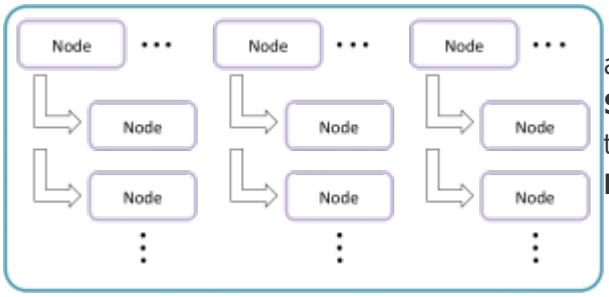
- A Node resembles an object from object oriented programming.
- A Node has attributes, which can be read (Data Access (DA), Historical Data Access (HDA)).
- Nodes are used for process data as well as for all other types of metadata.
- The therefore modelled OPC Address Space contains a type model with which all data types are specified.

NodeId



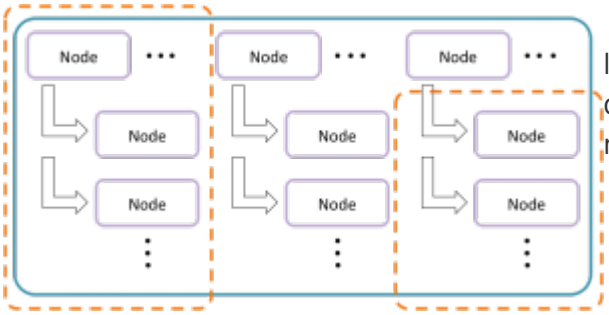
The OPC specification defines that every Node can be uniquely identified in the Adress Space via an Identifier (=**NodeId**). The **NodeId** is defined either by a GUID (Global Unique Identifier), a numeric expression, an array of bytes or a string value. In general but not necessarily, the NodeId contains the “**Namespace**”.

Address Space



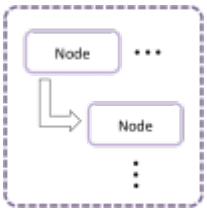
All **Nodes** supplied and processed in the OPC UA are administrated in a so-called **Address Space**. The **Address Space** depicts a kind of logical storage. In this “storage” the contained **Nodes** can logically refer to one or more **Nodes** in the same or another **Address Space**.

View



The “Address Space” mentioned / visualized earlier can be logically segmented into one or more Views. While there is one **Default View**, **Custom Views** can contain one or more Nodes.

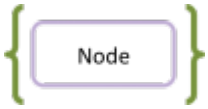
NodeManager



The **Node Manager** supplies one or more Nodes and defines their relationships towards each other. Predefined **System Node Managers** are:

- Core Node Manager (defines i.a. Type Nodes and System Nodes)
- Diagnostics Node Manager (supplies Nodes for diagnostics)
- Master Node Manager (the “administrator” of all Node Managers, it delegates calls to the concerning Node Managers)

Service



OPC UA defines a series of different **Services** by means of which the Client interacts with the Server. Those **Services** are server-sided implemented as Methods and are used for:

- reading and writing Note attributes / values
- administrating Node References
- browsing of Nodes
- reading and writing historical values
- calling Methods
- administrating subscriptions
- e.a.

Client Development Guide

OPC UA Framework Advanced

With only a few lines of code to the OPC UA Client

Introduction

Licensing

The OPC UA Framework Advanced comes with a **license for Client and Server development valid for 14 days**. This license allows you to fully test the **entire framework without restrictions**. Once the evaluation phase has expired, you have the option to apply for another test license. Just ask our support team or directly seek advice from us and let open questions be anwered, also concretely by our developers!

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

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

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

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

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

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

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

Note that a once set **bundle license ceases to be in force by additionally stating a Client license key**!

Connection to the Server

“Connect” - *What is happening there?*

1. Checking if an **address was set** (ServerAddress property).
2. The Client changes its status (**OpcClient.State** property) to the value ***Connecting***.
3. The Client **checks its configuration** for validity and conclusiveness.
4. Next the Client tries to **find an endpoint**.
   This happens via DiscoveryClient where endpoints with the desired endpoint configuration are compared and the endpoint fulfilling or at least sufficing the configuration is chosen. An endpoint is chosen depending on the used settings of the Client.
5. Next the Client creates the **configuration for a new session**.
6. **Instance certificates** are **checked**:
   1. Client certificate (depending on security configuration)
   2. Server certificate (the certificate provided by the endpoint)
7. A **channel** acting as a connection between Client and Server is **created**.
8. Attempt to **create a session** via the channel.
9. After further exchange and checking of session data the **session** gets **activated**.
10. Finally, **available Namespaces** are **retrieved**
11. And precautions for **connection surveillance** are taken:

    1. “KeepAlive-Tracking” for detecting connection abortions
    2. “Notification-Tracking” for receiving notifications
12. The Client changes its status (**OpcClient.State** property) to the value ***Connected***.

“Disconnect” - *What is happening there?*

1. The Client changes its status (**OpcClient.State** property) to the value ***Disconnecting***.
2. The Client **releases** all **gathered resources** (e.g. File Handles for OPC UA File Nodes)
3. **Ends** the **connection surveillance**
4. The **active session is ended**.
5. The **channel** created during Connect **is closed and disposed of**.
6. The Client changes its status (**OpcClient.State** property) to the value ***Disconnected***.

“BreakDetection” - *What is it and how is it important for the connection to the Server?*

The “BreakDetection” is a mechanism responsible for the detection of connection abortions using “KeepAlive”-Tracking to **detect a timeout of the connection to the Server**. If there is a timeout, the Client **automatically tries to establish a connection to the Server**. In case of a newly created connection a **new session** often happens. While KeepAlive messages are sent between Client and Server in KeepAlive in order to “test” the connection and “hold it up”, it is assumed that the connection was interrupted when response times to a KeepAlive message are too long (= reached timeout?). In that case another KeepAlive message is sent in increasing time intervals. An aborted connection is assumed if these messages also are unanswered and the previously described mechanism to re-establish the connection is introduced. The abortion detection is active by default and can be activated via the **OpcClient.UseBreakDetection** property.

“Connection Parameters” - *Which ones are there and how important are they for me?*

In order for the Client to connect to the Server the correct parameters have to be set. **Generally** the **address of the Server** (**OpcClient.ServerAddress** property) **is needed**. The Uri (= Uniform Resource Identifier) instance feeds the Client with every primarily necessary information about the Server. The Server-Address “opc.tcp://192.168.0.80:4840” e.g. contains the information of the scheme “opc.tcp” (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”) which establishes over which protocol data is exchanged in which way. In general “opc.tcp” is recommended for OPC UA Servers in a local network. Servers outside a local network should use “http” or even better “https”. Furthermore the address defines that the Server is executed on a computer with the IP address “192.168.0.80” and listens to requests via the port numbered “4840” (which is the default port for the OPC UA, custom port numbers are also possible). Instead of a static IP address the DNS name of the computer can be used as well, so instead of “127.0.0.1” also “localhost” can be used.

If the Server does **not define an endpoint** whose policy uses the security mode **“None”** (also possible are “Sign” and “SignAndEncrypt”) for data exchange, **this Endpoint-Policy has to be configured** manually (**OpcClient.Security.EndpointPolicy** property). If, however, an **endpoint with the policy “None”** is provided by the Server, **the Client automatically chooses it**. This **behavior** is activated by default and **can be deactivated** (**OpcClient.Security.UseOnlySecureEndpoints** property). The

automatic choice of the endpoint **can be done according to the OPC Foundation** by configuring the Client in a way that **the endpoint defining the highest Policy-Levels per definition** (a number) automatically is the “best” for data exchange. This behavior is deactivated by default but **can be activated** (**OpcClient.Security.UseHighLevelEndpoint** property).

If the Server uses an access control, for example via an ACL (= Access Control List), valid **user data for identifying the user has to be given** to the Server before a connection can be established. The user identity can either be varified through a username-password pair (**OpcClientIdentity** class) or through a certificate (**OpcCertificateIdentity** class). Then the identity has to be **mentioned to the Client** (**OpcClient.Security.UserIdentity** property) in order for it to deliver the identity to the Server while connecting.

“Endpoints” - *What is this and why are they needed?*

Endpoints result from the cross product of the used base addresses of the Server and the security strategies supported by the Server. The results are the base addresses of every scheme-port pair supported, while several schemes (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”) can be determined for data exchange on different ports. The hereby linked policies determine the procedure during the data exchange. Consisting of the Policy Level, the Security-Mode and the Security-Algorithm, every policy determines the kind of secure data exchange.

For example, when two Security-Policies are followed, they can be defined as follows:

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

When furthermore, for example, three Base-Addresses are combined for different schemes as follows:

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

The results will be the following endpoint descriptions through the cross product:

- 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

Here the address part of the endpoint is always needed by the Client (via constructor or via **ServerAddress** property). While the Client tries to find an endpoint with the Security-Mode “None” by default, the policy of the endpoint has to be configured manually (**OpcServer.Security.EndpointPolicies**

property) when none exists.

Information about Certificates

Certificates are used to **ensure** the **authenticity** and **integrity** of Client and Server applications. Therefore they act as a kind of identity card for Client as well as Server application. This “identification card” has to be stored somewhere as it exists as a form of file. The decision on where certificates are stored is individual. On Windows, every certificate can be passed to the **system** and Windows takes care of the **Store**. As an alternative **custom Stores** (= directories) can be set.

There are different types of Stores for certificates:

- **Store for application certificates**
  The Store also called **Application Certificate Store** exclusively contains certificates of those applications that use this Store as an Application Certificate Store. Here a Client / Server application saves its own certificate.
- **Store for certificates from trustworthy certificate issuers**
  The Store also called **Trusted Issuer Certificate Store** exclusively contains certificates from certificate issuers that issue further certificates. Here a Client / Server application saves all certificates from issuers whose certificates shall be treated as trusted by default.
- **Store for trustworthy certificates**
  The Store also called **Trusted Peer Store** exclusively contains certificates treated as trusted. Here a Client saves the **certificates from trusted Servers** and a Server saves the **certificates from trusted Clients**.
- **Store for rejected certificates**
  The Store also called **Rejected Certificate Store** exclusively contains certificates that are decreed as not trusted. Here a Client saves the **certificates from untrusted Servers** and a Server saves the **certificates from untrusted Clients**.

Regardless of the Store being located somewhere in the system or in the file system via a directory, generally certificates in the **Trusted Store** are **trusted** and certificates in the **Rejected Store** are **untrusted**. Certificates not belonging to either of the former are automatically saved in the Trusted Store, if the certificate of the certificate issuer mentioned in the certificate is deposited in the Trusted Issuer Store; otherwise it is automatically saved in the Rejected Store. Even if a trustworthy certificate has expired or if its deposited information cannot be successfully verified through the certification center the certificate is graded as not trustworthy and saved in the Rejected Store. During this process it also is removed from the Trusted Peer Store. A certificate can also expire when it is listed in a CRL (=Certificate Revocation List), which can be kept separately in the concerning store.

A certificate that the Client receives from the Server or the other way around is **for the moment always** classified as *unknown* and therefore also treated as **untrusted**. In order for a certificate to be treated as trusted it must be declared as such. This happens by saving the certificate of the Client in the Trusted Store of the Server and the certificate of the Server in the Trusted Store of the Client.

Dealing with a Server certificate at the Client:

1. The Client establishes the certificate of the Server on whose endpoint it shall connect with.
2. The Client verifies the certificate of the Server.
    1. Is the certificate valid?
        1. Has the effective date expired?
        2. Is the issuer's certificate valid?
    2. Does the certificate exist in the Trusted Peer Store?
        1. Is it listed in a CRL?

        3. Does the certificate exist in the Rejected Store?
    3. When the certificate is trusted, the Client establishes a connection to the server.

Dealing with a Client certificate at the Server:

1. The Server receives the Client's certificate from the Client while connecting.
2. The Server verifies the certificate of the Client.
    1. Is the certificate valid?
        1. Has the effective date expired?
        2. Is the issuer's certificate valid?
    2. Does the certificate exist in the Trusted Peer Store?
        1. Is it listed in a CRL?
    3. Does the certificate exist in the Rejected Store?
3. When the certificate is trusted, the Server allows the connection of the Client and operates it.

In case the verification of the certificate from the respective counterpart fails, the verification can be extended by custom mechanisms and still decided on user scale, if the certificate gets accepted or not.

Self-Signed Certificates vs. Signed Certificates
A certificate is comparable to a document. A document can be issued by everybody and can also be signed by everybody. However, the main difference here is, if the signee of a document really vouches for its correctness (like a notary) or if the signee is the owner of the document itself. Especially documents of the latter are not really inspiring confidence because no (legally) recognized instance as e.g. a notary vouches for the owner of the document.

As certificates are comparable to documents and also have to show a (digital) signature, the situation here is the same. The signature of a certificate has to tell the recipient of the certificate copy, who vouches for this certificate. Herefore it always applies that the issuer of a certificate also signs it. When the **issuer of a certificate equals the subject** of the certificate, you call this a **self-signed certificate** (subject equals issuer). When the **issuer of a certificate does not equal the subject** of the certificate, you call this a **(simple / normal / signed) certificate** (subject does not equal issuer).

As certificates are used especially in the context of the OPC UA authentication of an identity (of a certain Client or Server application), signed certificates should be used as application certificates for the own application. If, however, the issuer of the certificate also its owner, this self-signed certificate should only be trusted when the owner is rated as trusted. Such certificates were, as described, signed by the issuer of the certificate. Therefore, the issuer certificate has to be located in the **Trusted Issuer Store** of the application. When the issuer certificate cannot be found there , the certificate chain is declared incomplete and the certificate is not accepted by the counterpart. Yet, if the issuer certificate of the issuer of the application certificate is not a self-signed certificate, the certificate of its issuer has to be available in the **Trusted Issuer Store**.

User Identification through Certificates
Next to the use of a certificate as an *identification card* for Client / Server applications, a certificate can also be used to identify a user. A Client application is always operated by a certain user by whom it operates with the Server. Depending on the Server configuration a Server can request additional information about the identity of the Client's user from the Client. The user has the possibility to prove his identity through a certificate. How thoroughly a Server is examining the certificate on validity, authenticity and confidentiality depends on the Server. The Server provided by the Framework exclusively checks, if

the Thumbprint information of the user identity can be found in its ACL (=Access Control List) for certificate-based user identities.

Aspects of Security

The primary goal of the Framework is to make getting the grips of the OPC UA as easy as possible. This basic thought sadly also leads to the fact that without secondary configuration of the Server a completely save connection / communication between Client and Server does not occur. Yet, if the final Spike has been implemented and tested, second thought should be given to the *aspects of security*.

Even if one is dependant on the security mechanisms provided by the Server while developing a Client, one should always make the best choice possible. In general, the choice should always be the endpoint (**OpcClient.ServerAddress** property and **OpcClient.Security.EndpointPolicy** property) that provides the most secure connection (e.g. “https” instead of “http” as a scheme). This includes endpoints that follow the best Security-Policy possible. Keep an eye on the Security-Mode and the Security-Algorithm. According to the OPC Foundation, if you want to have the savest endpoint, refer to the endpoint with the highest Security-Level (**OpcClient.Security.UseHighLevelEndpoint** property).

For simplified handling of certificates the Client accepts every certificate by default (**OpcClient.Security.AutoAcceptUntrustedCertificates** property), also those it should deny under productive conditions because only certificates known to the Client (located in the Trusted Peer Store) apply as truly trusted. Apart from that the validity of a certificate should always be verified, including the “expiration date” of the certificate, for example. Furthermore it is advisable to check the domains referenced in the certificate (**OpcClient.Security.VerifyServersCertificateDomains** property). Other properties of the certificate or looser rules for the validity and trustworthiness of a Server certificate can be furthermore carried out manually (**OpcClient.CertificateValidationFailed** event).

If the Server uses a security process which controls the access via user identities, a concrete user identity should always be chosen (**OpcClient.Security.UserIdentity** property). Apart from the fact that anonymous identities almost always have only limited access, the Client can be granted access to more sensitive data when a concrete identity (e.g. a certificate or a username-password pair) is used. At the same time security is higher at e.g. a signed data transmission using a Certificate Identity.

Step by Step

The Client Frame

1. Add reference to the **Opc.UaFx.Advanced** Client Namespace:
   ```
   using Opc.UaFx.Client;
   ```
2. Create an instance of the OpcClient class with the address of the Server:
   ```
   var client = new OpcClient("opc.tcp://localhost:4840/");
   ```
3. Build a connection to the Server and start a session:
   ```
   client.Connect();
   ```
4. Your code to interact with the Server:
   ```
   // Your code to interact with the server.
   ```
5. Close all sessions before closing the application:
   ```
   client.Disconnect();
   ```

6. Using the using block this looks as follows:

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

Reading Values of Node(s)

The following types are used: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.**OpcValue**, Opc.UaFx.**OpcAttribute** and Opc.UaFx.Client.**OpcReadNode**.

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

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

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

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

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

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

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

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

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

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

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

Writing Values of Node(s)

The following types are used: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.**OpcValue**, Opc.UaFx.**OpcAttribute**, Opc.UaFx.**OpcStatus**, Opc.UaFx.**OpcStatusCollection** and Opc.UaFx.Client.**OpcWriteNode**.

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

- Write the value of a single Node:

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

- Write the values of several Nodes:

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

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

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

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

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

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

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

Processing Values of Node(s)

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.**OpcValue**, Opc.UaFx.**OpcStatus** and Opc.UaFx.**OpcStatusCollection**.

The **ReadNode** methods always provide an **OpcValue** instance, while the **ReadNodes** methods provide a list of **OpcValue** instances (one **OpcValue** per read Node). The actual value read is in the *Value* property of the **OpcValue** instances. The result of the read request can be checked via the *Status* property. The timestamp at which the read value has been detected at the sourcecan be retrieved via the *SourceTimestamp* property. Correspondingly, the timestamp at which the read value was detected by the Server can be retrieved via the *ServerTimestamp* property.

- Read the value of a single Node:

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

- Chech the result of the read request:

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

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

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

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

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

The **WriteNode** methods always provide an **OpcStatus** instance, while the **WriteNodes** methods provide an **OpcStatusCollection** instance (that contains an **OpcStatus** for every written Node). The result of the

write request can thereby be checkt via the properties of the **OpcStatus** instance(s).

- Write the scalar value of a single Node:

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

- Write the array value of a single Node:

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

- Check the result of a write request:

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

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

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

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

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

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

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

Browsing Nodes

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.Client.**OpcNodeInfo**, Opc.UaFx.**OpcAttribute**, Opc.UaFx.Client.**OpcAttributeInfo**, Opc.UaFx.Client.**OpcMethodNodeInfo**, Opc.UaFx.Client.**OpcArgumentInfo** and Opc.UaFx.Client.**OpcObjectTypes**.

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

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

1. Determine the OpcNodeInfo of the desired Node:

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

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

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

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

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

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

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

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

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

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

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

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

    level++;

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

Working with Subscriptions

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.Client.**OpcSubscribeDataChange**, Opc.UaFx.Client.**OpcSubscribeEventRaise**, Opc.UaFx.Client.**OpcSubscription**, Opc.UaFx.Client.**OpcMonitoredItem**, Opc.UaFx.Client.**OpcMonitoredItemCollection**, Opc.UaFx.Client.**OpcDataChangeCallback**, Opc.UaFx.Client.**OpcDataChangeEventArgs**, Opc.UaFx.Client.**OpcEventRaiseCallback**, Opc.UaFx.Client.**OpcEventRaiseEventArgs** and Opc.UaFx.**OpcNotification**.

Subscriptions in the OPC UA can be compared to subscriptions to one or more magazines in a bundle. Instead of magazines one subscribes to Node events (= in the OPC UA: Monitored Item) or changes of the Node data (= in the OPC UA: Monitored Item). In every subscription the interval can be chosen in which

notifications (**OpcNotification** instances) of the Monitored Items shall be published (**OpcSubscription.PublishingInterval**). The **OpcNodeId** of the Node determines which Node shall be subscribed to. By default the *Value Attribute* is monitored. If another attribute of the Node shall be monitored the according **OpcAttribute** has to be mentioned at the call of the **SubscribeDataChange** or at the concerning **OpcSubscribeDataChange** instance.

- Subscribe to notifications about changes of the Node value:

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

- Subscribe to notifications about changes of several Node values:

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

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

- Treat notifications about changes of Node values:

```
private void HandleDataChanged(object sender, OpcDataChangeEventArgs e)
{
    // Your code to execute on each data change.
}
```

- Subscribe to notifications about Node events:

```
OpcSubscription subscription = client.SubscribeEventRaise("2:Machine", HandleEvent);
```

- Subscribe to notifications about Node events of several Nodes:

```
OpcSubscribeEventRaise[] commands = new OpcSubscribeEventRaise[] {
    new OpcSubscribeEventRaise("2:Machine", HandleEvent),
    new OpcSubscribeEventRaise("2:Machine/Job", HandleEvent)
};

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

- Treat notifications about Node events:

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

- Configuration of the **OpcSubscription**:

```
subscription.PublishingInterval = 2000;

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

Working with Historical Data

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.Client.**IOpcNodeHistoryNavigator**, Opc.UaFx.**OpcHistoryValue** and Opc.UaFx.**OpcAggregateType**.

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

In order to read historical data the Client can:

- read all values within an **open time frame** (= non-defined StartTime or EndTime)
  - read all values **from a particular timestamp forward** (= StartTime):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and EndTime):

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

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

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

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

historyNavigator.Close();
```

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

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

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

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

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

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

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

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

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

Working with Method Nodes

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId** and Opc.UaFx.Client.**OpcCallMethod**.

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

- Call a single method Node:

```
object[] result = client.CallMethod("2:Machine", "2:Machine/StartMachine");
```

- Call several method Nodes:

```
OpcCallMethod[] commands = new OpcCallMethod[] {
    new OpcCallMethod("2:Machine", "2:Machine/StopMachine", "Job Change" /* reason
*/),
    new OpcCallMethod("2:Machine", "2:Machine/ScheduleJob", "MAN_F01_78910" /* job
serial */),
    new OpcCallMethod("2:Machine", "2:Machine/StartMachine")
};

object[][] results = client.CallMethods(commands);
```

Working with File Nodes

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.Client.**OpcFile**, Opc.UaFx.**OpcFileMode**, Opc.UaFx.Client.**OpcFileStream**, Opc.UaFx.Client.**OpcFileInfo**, Opc.UaFx.Client.**OpcFileMethods** and Opc.UaFx.Client.**SafeOpcFileHandle**.

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

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

Data access with the **OpcFile** class:

- Reading the entire content of a text file:

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

- Appending further text data to a text file:

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

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

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

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

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

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

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

Data access with the **OpcFileInfo** class:

- Creating an **OpcFileInfo** instance:

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

- Working with the **OpcFileInfo** instance:

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

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

Data access with the **OpcFileMethods** class:

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

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

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

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

- via numeric File Handle:

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

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

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

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

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

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

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

Client Configuration

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcCertificateStores** und Opc.UaFx.**OpcCertificateStoreInfo**.

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

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

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

- Loading the configuration file via the App.config

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

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

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

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

- Configuring the application
  - via Code:

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

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

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

```
<ApplicationName>MyClientAppName</ApplicationName>
<ApplicationUri>http://my.clientapp.uri/</ApplicationUri>
```

- Configuring the session parameters
  - via Code:

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

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

```
  <ClientConfiguration>
    <DefaultSessionTimeout>30000</DefaultSessionTimeout>
  </ClientConfiguration>
```

- Configuring the connection parameters
  - via Code:

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

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

```
  <TransportQuotas>
    <OperationTimeout>10000</OperationTimeout>
  </TransportQuotas>
```

- Configuring the Certificate Store
  - via Code:

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

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

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

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

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

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

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

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

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

Client Certificate Configuration

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcCertificateManager**, Opc.UaFx.Client.**OpcClientSecurity**, Opc.UaFx.**OpcCertificateStores** and Opc.UaFx.**OpcCertificateStoreInfo**.

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

1. An **existing certificate** is loaded from any path:
   ```
   var certificate = OpcCertificateManager.LoadCertificate("MyClientCertificate.pfx");
   ```
2. A **new certificate** is generated (in memory):
   ```
   var certificate = OpcCertificateManager.CreateCertificate(client);
   ```
3. Save a certificate in any path:
   ```
   OpcCertificateManager.SaveCertificate("MyClientCertificate.pfx", certificate);
   ```
4. Set the Client certificate:
   ```
   client.Certificate = certificate;
   ```
5. The certificate has to be stored in the **Application Store**:
   ```
   if (!client.CertificateStores.ApplicationStore.Contains(certificate))
       client.CertificateStores.ApplicationStore.Add(certificate);
   ```
6. If **no or an invalid certificate** is used, a new certificate is generated / used by default. If the Client shall only use the mentioned certificate this function has to be deactivated. For **deactivating the**

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

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

Configuring Client User Identities

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.**OpcUserIdentity**, Opc.UaFx.Client.**OpcClientIdentity**, Opc.UaFx.**OpcCertificateIdentity**, Opc.UaFx.Client.**OpcClientSecurity**, Opc.UaFx.Client.**OpcCertificateStores** and Opc.UaFx.**OpcCertificateStoreInfo**.

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

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

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

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

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

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

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

Configuring Server Endpoints

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.Client.**OpcClientSecurity**, Opc.UaFx.**OpcSecurityPolicy**, Opc.UaFx.**OpcSecurityMode**, Opc.UaFx.**OpcSecurityAlgorithm** and Opc.UaFx.Client.**OpcDiscoveryClient**.

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

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

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

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

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

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

```
client.Security.EndpointPolicy = new OpcSecurityPolicy(
        OpcSecurityMode.None, OpcSecurityAlgorithm.Basic256, 12);
```

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

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

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

Configuring further Security Settings

The following types are used here: Opc.UaFx.Client.**OpcClient**, Opc.UaFx.Client.**OpcClientSecurity**, Opc.UaFx.**OpcCertificateValidationFailedEventArgs**, Opc.UaFx.**OpcCertificateStores** and Opc.UaFx.**OpcCertificateStoreInfo**.

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

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

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

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

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

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

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

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

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

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

Preparing Client Configuration via XML

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

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

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

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

Creating and preparing the App.config of the application:

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

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

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

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

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

Creating and preparing the XML configuration file:

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

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

3. Save the changes to the XML file.

Client Application Delivery

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

Application certificates - *Using a concrete certificate*
For productive use don't use a certificate automatically generated by the Framework.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Server Development Guide

OPC UA Framework Advanced

With only a few lines of code to the OPC UA Server

Introduction

Licensing

The OPC UA Framework Advanced comes with a **license for Client and Server development valid for 14 days**. This license allows you to fully test the **entire framework without restrictions**. Once the evaluation phase has expired, you have the option to apply for another test license. Just ask our support team or directly seek advice from us and let open questions be anwered, also concretely by our developers!

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

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

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

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

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

```
LicenseInfo license = Opc.UaFx.Server.Licenser.LicenseInfo;

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

Note that a once set **bundle license ceases to be in force by additionally stating a Server license key**!

The Server

“Start” - *What is happening there??*

1. Checking, if an **address was set** (Address Property).
2. The Server changes its status (**OpcServer.State** property) to the value ***Starting***.
3. The Server **checks its configuration** for validity and conclusiveness.
4. Next the Server tries to **create a host for every endpoint description**.
5. What follows is the **start of all Managers** (NodeManager, SessionManager, ...)
6. Finally, **every created host** responsible for the endpoint-specific communication with the Client gets **started**.
7. The Server changes its status (**OpcServer.State** property) to the value ***Started***.

“Stop” - *What is happening there?*

1. The Server changes its status (**OpcServer.State** property) to the value ***Stopping***.
2. **Closing all Managers** (NodeManager, SessionManager, ...)
3. The Server **releases** all **gathered resources**.
4. **Closing** the **hosts of endpoint descriptions**.

5. The Server changes its status (**OpcServer.State** property) to the value ***Stopped***.

“Parameter” - *Which ones exist and how important are they for me?*

In order for the Server to be able to give the Clients access to its OPC UA Services, the right parameters have to be determined. **In general** the **server address** (**OpcServer.Address** property) **is needed**. The Uri instance (= Uniform Resource Identifier) supplies all Clients the primarily necessary information via the server. For example, the server address “opc.tcp://192.168.0.80:4840” contains the information of the concept “opc.tcp” (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”), which determines via which protocol the data shall be exchanged. Generally, “opc.tcp” is advisable for OPC UA Servers in a local network. Servers out of the local network should use “http”, even better “https”. Furthermore the address defines that the server is carried out on the computer with the IP address “192.168.0.80” and listens to requests via the port with the number “4840” (which is default for OPC UA, customized port numbers are also possible). Instead of the static IP address the DNS name of the computer can also be used, so instead of “127.0.0.1” you could also use “localhost”.

If the server shall provide **no endpoint** for data exchange with savety mode **“None”** (also additionally possible are “Sign” and “SignAndEncrypt”) as its policy, **at least one Endpoint-Policy has to be** manually **configured** (**OpcServer.Security.EndpointPolicies** property). If, however, an **endpoint with the Property “None”** is supplied by the Server, a Client can select this one automatically for an easy and quick connection. When a policy level (a number) is **assigned according to the OPC Foundation** to the separate Endpoint Policies during the definition of the endpoints, Clients can handle those appropriately. Here the OPC intends that the higher the level of the Policy of an endpoint, the “better” that endpoint (note that this is merely a neither watched nor imposed guideline).

If the Server shall use an access control, for example via an ACL (= Access Control List), **the user data has to be determined for identification of possible / valid identities of users** of the Server (this also works in a running system). Here it is possible to determine the identities of users through a username-password pair (**OpcServerIdentity** class) or through a certificate (**OpcCertificateIdentity** class). Those identities have then to be **communicated to the Server** (**OpcServer.Security.UserNameAcl/CertificateAcl** property). Those access control lists have to be activated in order for the server to recognize them (**OpcServer.Security.UserNameAcl/CertificateAcl.IsEnabled** property).

“Endpoints” - *What is this and why are they needed?*

Endpoints result from the crossing of the used Base-Addresses of the Server and the security strategies supported by the Server. The results are the Base-Addresses of every scheme-port pair supported, while several schemes (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”) can be determined for data exchange on different ports. The hereby linked policies determine the procedure during the data exchange. Consisting of the Policy Level, the Security-Mode and the Security-Algorithm, every policy determines the kind of secure data exchange.

For example, when two Security-Policies are followed, they can be defined as follows:

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

When furthermore, for example, three Base-Addresses are combined for different schemes as follows:

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

The results will be the following endpoint descriptions through the crossing:

- 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

Here the address part of the endpoint is always determined by the Server (via constructor or via **Address** property). While the Server defines an endpoint with the Security-Mode “None” by default, the policy of the endpoint has to be configured manually (**OpcServer.Security.EndpointPolicies** property) when none of this kind or additional ones shall be used.

Information about Certificates

Certificates are used to **ensure** the **authenticity** and **integrity** of Client and Server applications. Therefore they act as a kind of identity card for Client as well as Server application. This “identification card” has to be stored somewhere as it exists as a form or data. The decision on where certificates are stored is individual.

There are different types of Stores for certificates:

- **Store for user certificates**
  The Store also called **Application Certificate Store** exclusively contains certificates of those applications that use this Store as an Application Certificate Store. Here a Client / Server application saves its own certificate.
- **Store for certificates from trustworthy certificate issuers**
  The Store also called **Trusted Issuer Certificate Store** exclusively contains certificates from certificate issuers that issues further certificates. Here a Client / Server application saves all certificates from issuers whose certificates shall be treated as trusted by default.
- **Store for trustworthy certificates**
  The Store also called **Trusted Peer Store** exclusively contains certificates treated as trusted. Here a Client saves the **certificates from trusted Servers** and a Server saves the **certificates from trusted Clients**.
- **Store for rejected certificates**
  The Store also called **Rejected Certificate Store** exclusively contains certificates that are decreed as not trusted. Here a Client saves the **certificates from not trusted Servers** and a Server saves the **certificates from not trusted Clients**.

Regardless of the Store being located somewhere in the system or in the data system via a list, generally certificates in the **Trusted Store** are **trusted** and certificates in the **Rejected Store** are untrusted.

Certificates not belonging to eeither of the former are automatically saved in the Trusted Store, if the certificate of the certificate issuer mentioned in the certificate is deposited in the Trusted Issuer Store; otherwise it is automatically saved in the Rejected Store. Even if a trustworthy certificate has expired or if its deposited information cannot be successfully verified through the certification center the certificate is graded as not trustworthy and saved in the Rejected Store. During this process it also is removed from the Trusted Peer Store. A certificate can also expire when it is listed in a CRL (=Certificate Revocation List), which can be kept separately in the concerning store.

A certificate that the Client receives from the Server or the other way around is **for the moment always** classified as *unknown* and therefore also treated as **untrusted**. In order for a certificate to be treated as trusted it must be declared as such. This happens by saving the certificate of the Client in the Trusted Store of the Server and the certificate of the Server in the Trusted Store of the Client.

Dealing with a Server certificate at the Client:

1. The Client establishes the certificate on the Server on whose Endpoint it shall connect with.
2. The Client verifies the certificate of the Server.
    1. Is the certificate valid?
        1. Has the effective date expired?
        2. Is the issuer's certificate valid?
    2. Does the certificate exist in the Trusted Peer Store?
        1. Is it listed in a CRL?
    3. Does the certificate exist in the Rejected Store?
3. When the certificate is trusted, the Client establishes a connection to the server.

Dealing with a Client certificate at the Server:

1. The Server receives the Client's certificate from the Client while connecting.
2. The Server verifies the certificate of the Client.
    1. Is the certificate valid?
        1. Has the effective date expired?
        2. Is the issuer's certificate valid?
    2. Does the certificate exist in the Trusted Peer Store?
        1. Is it listed in a CRL?
    3. Does the certificate exist in the Rejected Store?
3. When the certificate is trusted, the Server allows the connection of the Client and operates it.

In case the verification of the certificate fails at the respective counterpart the verification can be extended by custom mechamisms and still decided on user scale, if the certificate gets accepted or not.

Self-Signed Certificates vs. Signed Certificates

A certificate is comparable to a document. A document can be issued by everybody and can also be signed by everybody. However, the main difference here is, if the signee of a document really vouches for its correctness (like a notary) or if the signee is the owner of the document itself. Especially documents of the latter are not really inspiring confidence because no (legally) recognized instance as e.g. a notary vouches for the owner of the document.

As certificates are comparable to documents and also have to show a (digital) signature, the situation here is the same. The signature of a certificate has to tell the recipient of the certificate copy, who vouches for this certificate. Herefore it always applies that the issuer of a certificate also signs it. When the **issuer of a certificate equals the subject** of the certificate, you call this a **self-signed certificate** (subject equals

issuer). When the **issuer of a certificate does not equal the subject** of the certificate, you call this a **(simple / normal / signed) certificate** (subject does not equal issuer).

As certificates are used especially in the context of the OPC UA authentication of an identity (of a certain Client or Server application), signed certificates should be used as application certificates for the own application. If, however, the issuer of the certificate also its owner, this self-signed certificate should only be trusted when the owner is rated as trusted. Such certificates were, as described, signed by the issuer of the certificate. Therefore, the issuer certificate has to be located in the **Trusted Issuer Store** of the application. When the issuer certificate cannot be found there , the certificate chain is declared incomplete and the certificate is not accepted by the counterpart. Yet, if the issuer certificate of the issuer of the application certificate is not a self-signed certificate, the certificate of its issuer has to be available in the **Trusted Issuer Store**.

User identification through certificates
Next to the use of a certificate as an *identification card* for Client / Server applications, a certificate can also be used to identify a user. A Client application is always operated by a certain user by whom it operates with the Server. Depending on the Server configuration a Server can request additional information about the identity of the Client's user from the Client. The user has the possibility to prove his identity through a certificate. How thoroughly a Server is examining the certificate on validity, authenticity and confidentiality depends on the Server. The Server provided by the Framework exclusively checks, if the Thumbprint information of the user identity can be found in its ACL (=Access Control List) for certificate-based user identities.

Aspects of Security

The primary goal of the Framework is to make getting the grips of the OPC UA as easy as possible. This basic thought sadly also leads to the fact that without secondary configuration of the Server a completely save connection / communication between Client and Server does not occur. Yet, if the final Spike has been implemented and tested, second thought should be given to the *aspects of security*.

Even if the Server is *only* run within a local network one should consider the use of access control lists (**OpcServer.Security.UserNameAcl/CertificateAcl** property). Here user identities can be defined via a certificate or a username-password pair. A Certificate Identity increases security in signed data transmission, for example.

Especially in cases of the Server being accessible publicly, other Security-Policies with appropriately higher Security-Mode and a matching Security-Algorithm should be negotiated. The Security-Policy-Mode “None” used by default is in this matter literally the “Great Wide Open” into your Server (**OpcServer.Security.EndpointPolicies** Properties). Last but not least one should consider the access via an anonymous user identity (**OpcServer.Security.AnonymousAcl.IsEnabled** property). According to the OPC Foundation every Endpoint Policy used above its level should stress its “quality”, in which the rule applies that the higher the level, the “better” the endpoint this policy uses.

For simplified handling of certificates the Server accepts every certificate by default (**OpcServer.Security.AutoAcceptUntrustedCertificates** property), also those it should deny under productive conditions because only certificates known to the Server (located in the Trusted Peer Store) apply as truly trusted. Apart from that the validity of a certificate should always be verified, including the “expiration date” of the certificate, for example. Other properties of the certificate or looser rules for the validity and trustworthiness of a Client certificate can be furthermore carried out manually (**OpcServer.CertificateValidationFailed** event).

Step by Step

The Server Frame

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

Node Management

The following types are used: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.**OpcNode**, Opc.UaFx.**OpcNodeId**, Opc.UaFx.Server.**OpcNodeManager**, Opc.UaFx.**OpcFolderNode**, Opc.UaFx.**OpcVariableNode**, Opc.UaFx.**OpcDataVariableNode**, Opc.UaFx.**OpcDataVariableNode<T>** and Opc.UaFx.**OpcFileNode**.

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

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

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

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

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

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

- Define a custom Node-Manager:

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

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

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

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

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

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

- Introduce a custom Node-Manager to the Server:

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

Reading Values of Node(s)

The following types are used: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.**OpcVariableNode**, Opc.UaFx.**OpcDataVariableNode**, Opc.UaFx.**OpcDataVariableNode<T>**, Opc.UaFx.**OpcReadAttributeValueCallback**, Opc.UaFx.**OpcAttributeValue<T>**, Opc.UaFx.**OpcReadAttributeValueContext**, Opc.UaFx.**OpcReadVariableValueCallback**, Opc.UaFx.**OpcReadVariableValueContext** and Opc.UaFx.**OpcVariableValue<object>**.

An **OpcNode** defines its metadata by *attributes*. Contrasting the generally always provided *attributes* like *Name*, *DisplayName* or *Description*, the *Value Attribute* is only available on *Variable-Nodes*. The values of *attributes* are saved by the concerning Node-Instances internally by default. If the value of another source of data is to be established, appropriate Callback-Methods for provision of the values can be defined. Here the signature of the ReadVariableValue-Callback-Method differenciates from the other ReadAttributeValue-Callback-Methods. In case of the *Value Attribute* instead of an **OpcAttributeValue** instance a **OpcVariableValue** instance is expected. This instance consists, additionally to the actual value, of a time

stamp at which the value was identified at the source of the value (**SourceTimestamp**) and of status information about the quality of the value. Note that the Read-Callbacks are retrieved at every read operation of the metadata by a Client. This is the case when using the services Read and Browse.

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

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

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

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

- Set the value of a Description Attribute:

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

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

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

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

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

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

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

Writing Values of Node(s)

The following types are used: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.**OpcVariableNode**, Opc.UaFx.**OpcDataVariableNode**, Opc.UaFx.**OpcDataVariableNode<T>**, Opc.UaFx.**OpcWriteAttributeValueCallback**, Opc.UaFx.**OpcAttributeValue<T>**, Opc.UaFx.**OpcWriteAttributeValueContext**, Opc.UaFx.**OpcWriteVariableValueCallback**, Opc.UaFx.**OpcWriteVariableValueContext** and Opc.UaFx.**OpcVariableValue<object>**.

An **OpcNode** defines its metadata by *attributes*. Contrasting the generally always provided *attributes* like *Name*, *DisplayName* or *Description*, the *Value Attribute* is only available on *Variable-Nodes*. The values of *attributes* are saved by the concerning Node-Instances internally by default. If the value is to be saved into another data source, appropriate Callback-Methods for saving the values can be defined. Here the signature of the WriteVariableValue-Callback-Method differenciates from the other WriteAttributeValue-

Callback-Methods. In case of the *Value Attribute* instead of an **OpcAttributeValue** instance a **OpcVariableValue** instance is expected. This instance consists, additionally to the actual value, of a time stamp at which the value was identified at the source of the value (**SourceTimestamp**) and of status information about the quality of the value. Note that the Write-Callbacks are retrieved at every write operation of the metadata by a Client. This is the case when using the Write service.

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

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

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

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

- Set the value of a Description Attribute:

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

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

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

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

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

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

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

Working with Historical Data

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

According to the OPC UA specification every Node of the category **Variable** supports the historical logging of the values of its *Value Attribute*. With every change in value of the *Value Attribute* the new value is saved together with the time stamp of the *Value Attribute*. These **pairs consisting of value and time stamp** are called **historical data**. The Server decides on where to save the data. However, the Client can

determine via the *IsHistorizing Attribute* of the Node, if the Server supplies historical data for a Node and / or historically saves value changes. A Client can read, update, replace, delete or create historical data. Mostly, historical data is read by the Client.

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

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

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

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

  - Manual activation and release of the historical data history:

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

machineIsRunningNode.IsHistorizing = true;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Szenario: **Create History**:

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

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

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

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

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

return results;
```

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

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

int index = ;

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

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

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

return results;
```

  - Via values:

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

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

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

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

return results;
```

- Via time span:

```
var results = new OpcStatusCollection();

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

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

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

        results.Add(OpcStatusCode.Good);
    }
}

return results;
```

- Scenario: **Replace History**:

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

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

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

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

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

return results;
```

- Scenario: **Update History**:

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

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

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

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

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

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

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

return results;
```

- Scenario: **Read History**:

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

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

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

Providing Method Nodes

The following types are used here: Opc.UaFx.Server.**OpcNodeManager**, Opc.UaFx.**OpcMethodNode** and Opc.UaFx.**OpcMethodContext**.

Code sections that fulfill an isolated task are called subprograms in programming. Those subprograms are often described simply as functions or methods. Those kind of methods can be called via method Nodes in the OPC UA. A method Node is defined by the **OpcMethodNode** class. Method Nodes are called by an OPC UA Client via the server-sided **Call** service.

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

Define a method Node in the Node manager:

1. through a method without a parameter:

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

2. through a method with a parameter:

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

3. through a method with a callback value:

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

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

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

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

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

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

    return statusCode;
}
```

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

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

Providing File Nodes

The following types are used: Opc.UaFx.Server.**OpcNodeManager** and Opc.UaFx.**OpcFileNode**.

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

Define a File Node in the Node-Manager:

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

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

Server Configuration

The following types are used here: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.**OpcCertificateStores** and Opc.UaFx.**OpcCertificateStoreInfo**.

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

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

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

- Loading the configuration file via App.config

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

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

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

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

- Configurating the application
  - via Code:

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

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

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

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

- Configuring the Certificate Store
  - via Code:

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

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

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

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

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

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

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

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

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

Server Certificate Configuration

The following types are used here: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.**OpcCertificateManager**, Opc.UaFx.Server.**OpcServerSecurity**, Opc.UaFx.**OpcCertificateStores** and Opc.UaFx.**OpcCertificateStoreInfo**.

Recommended are certificates of types .der, .pem, **.pfx** and **.p12**. If the Server shall provide a secure endpoint (in which the **OpcSecurityMode** equals *Sign* or *SignAndEncrypt*), the certificate has to have a private key.

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

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

2. A **new certificate** is generated (in storage):
```
var certificate = OpcCertificateManager.CreateCertificate(server);
```
3. Save a certificate in any path:
```
OpcCertificateManager.SaveCertificate("MyServerCertificate.pfx", certificate);
```
4. Set the Server certificate:
```
server.Certificate = certificate;
```
5. The certificate has to be stored in the **Application Store**:
```
if (!server.CertificateStores.ApplicationStore.Contains(certificate))
    server.CertificateStores.ApplicationStore.Add(certificate);
```
6. If **no or an invalid certificate** is used, a new certificate is generated / used by default. If the Server shall only use the mentioned certificate this function has to be deactivated. For **deactivating the function** set the property **AutoCreateCertificate** to the value *false*:
```
server.CertificateStores.AutoCreateCertificate = false;
```

Configuring Server User Identities

The following types are used here: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.**OpcUserIdentity**, Opc.UaFx.Server.**OpcServerIdentity**, Opc.UaFx.**OpcCertificateIdentity**, Opc.UaFx.Server.**OpcServerSecurity**, Opc.UaFx.Server.**OpcAccessControlList**, Opc.UaFx.Server.**OpcAnonymousAcl**, Opc.UaFx.Server.**OpcUserNameAcl**, Opc.UaFx.Server.**OpcCertifciateAcl**, Opc.UaFx.Server.**OpcAccessControlEntry**, Opc.UaFx.Server.**OpcOperationType**, Opc.UaFx.Server.**OpcRequestType** and Opc.UaFx.Server.**OpcAccessControlMode**.

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

- Deactivating the anonymous ACL:
```
server.Security.AnonymousAcl.IsEnabled = false;
```
- Configuring the **username-password pair**-based ACL:
```
var acl = server.Security.UserNameAcl;

acl.AddEntry("username1", "password1");
acl.AddEntry("username2", "password2");
acl.AddEntry("username3", "password3");
...
acl.IsEnabled = true;
```
- Configuring the **certificate**-based ACL:

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

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

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

1. Remember an Access Control Entry:
   ```
   var user1 = acl.AddEntry("username1", "password1");
   ```
2. Deny the Access Control Entry two rights:
   ```
   user1.Deny(OpcRequestType.Write);
   user1.Deny(OpcRequestType.HistoryUpdate);
   ```
3. Allow a previously denied right:
   ```
   user1.Allow(OpcRequestType.HistoryUpdate);
   ```

Configuring Server Endpoints

The following types are used here: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.Server.**OpcServerSecurity**, Opc.UaFx.**OpcSecurityPolicy**, Opc.UaFx.**OpcSecurityMode** and Opc.UaFx.**OpcSecurityAlgorithm**.

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

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

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

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

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

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

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

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

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

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

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

Define two more Base-Addresses:

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

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

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

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

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

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

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

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

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

Configuring further Security Settings

The following types are used here: Opc.UaFx.Server.**OpcServer**, Opc.UaFx.Server.**OpcServerSecurity**, Opc.UaFx.**OpcCertificateValidationFailedEventArgs**, Opc.UaFx.**OpcCertificateStores** und Opc.UaFx.**OpcCertificateStoreInfo**.

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

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

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

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

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

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

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

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

Preparing Server Configuration via XML

If the Server shall also be configurable via XML the Server configuration can be loaded either from a specific or a random XML file.

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

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

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

Compiling and preparing the App.config of the application:

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

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

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

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

4. The value of the *FilePath* element can show to a random data path where you will find the XML configuration file needed. The value shown here would show to a configuration file lying next to the application.
5. Save the changes to App.config

Creating and preparing the XML configuration file:

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

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

3. Save changes to XML file

Server Application Delivery

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

System Configuration - *Administrative Setup*

Execute the application in the target system once only with administrative rights to ensure that the Server

has permission to access the network resources. This is necessary, if e.g. the Server shall use a Base-Address with the scheme “http” or “https”.

Table of Contents