SDK Guide
Clavister InControl

Version 1.10.02

Clavister AB
Sjögatan 6J
SE-89160 Örnsköldsvik
SWEDEN

Phone: +46-660-299200
Fax: +46-660-12250

www.clavister.com

Published 2010-04-06

SDK Guide
Clavister InControl
Version 1.10.02

Published 2010-04-06

Copyright Notice

Disclaimer

The information in this document is subject to change without notice. Clavister makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for a particular purpose. Clavister reserves the right to revise this publication and to make changes from time to time in the content hereof without any obligation to notify any person or parties of such revision or changes.

Limitations of Liability

UNDER NO CIRCUMSTANCES SHALL CLAVISTER OR ITS SUPPLIERS BE LIABLE FOR DAMAGES OF ANY CHARACTER (E.G. DAMAGES FOR LOSS OF PROFIT, SOFTWARE RESTORATION, WORK STOPPAGE, LOSS OF SAVED DATA OR ANY OTHER COMMERCIAL DAMAGES OR LOSSES) RESULTING FROM THE APPLICATION OR IMPROPER USE OF THE CLAVISTER PRODUCT OR FAILURE OF THE PRODUCT, EVEN IF CLAVISTER IS INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. FURTHERMORE, CLAVISTER WILL NOT BE LIABLE FOR THIRD-PARTY CLAIMS AGAINST CUSTOMER FOR LOSSES OR DAMAGES. CLAVISTER WILL IN NO EVENT BE LIABLE FOR ANY DAMAGES IN EXCESS OF THE AMOUNT CLAVISTER RECEIVED FROM THE END-USER FOR THE PRODUCT.

Table of Contents

Preface 4
1. An SDK Overview 5
2. Starting Coding 8
3. Opening Configurations 11
4. Editing Configurations 13
5. Check In and Deployment 18
A. Visual Studio Setup 19

Preface

Target Audience

The target audience for this publication is software developers who wish to access and manipulate Clavister Security Gateway configurations directly from custom applications.

Text Structure

The text is divided into chapters and subsections. Numbered subsections are shown in the table of contents at the beginning of the document.

Text links

Where a "See section" link is provided in the main text, this can be clicked on to take the reader directly to that reference. For example, "*see Chapter 1, An SDK Overview*".

Web links

Web links included in the document are clickable, for example http://www.clavister.com.

Notes to the main text

Special sections of text which the reader should pay special attention to are indicated by icons on the left hand side of the page followed by a short paragraph in italicized text. There are the following types of such sections:

Note
This indicates some piece of information that is an addition to the preceding text. It may concern something that is being emphasized or something that is not obvious or explicitly stated in the preceding text.

Caution
This indicates where the reader should be careful with their actions as an undesirable situation may result if care is not exercised.

Important
This is an essential point that the reader should read and understand.

Warning
This is essential reading for the user as they should be aware that a serious situation may result if certain actions are taken or not taken.

Chapter 1: An SDK Overview

The *InControl SDK* is a toolset that allows third parties to create custom clients applications which can programmatically manage and configure Clavister Security Gateways. The custom client applications achieve this by making use of the *InControl Application Programming Interface* (API). This API is the central component of the SDK.

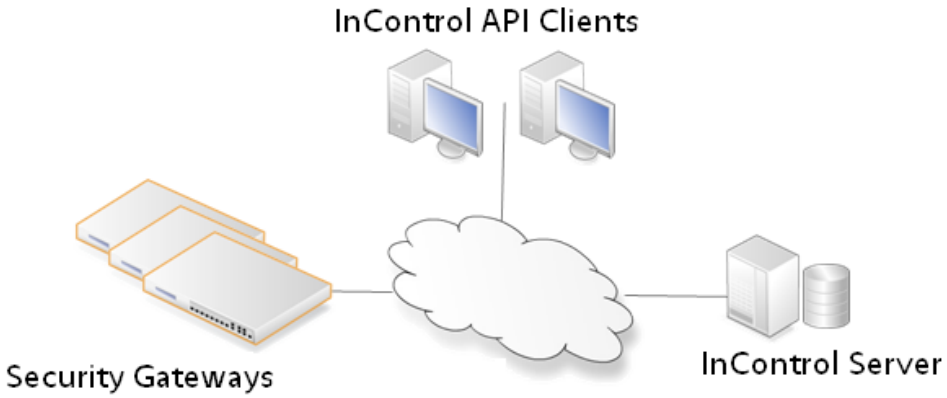
The complete SDK is included in the Clavister InControl installation package and contains the following:

- This PDF guide.
- The complete InControl API reference description in Windows Help format.

The InControl API is built into the InControl server by default and there is no need for additional binaries.

The Purpose of the InControl API

The InControl API provides the ability to create applications for managing and configuring one or a number of Clavister Security Gateways. Applications can be written using any one of a number of programming languages and can run on a variety computing platforms.



Client applications don't communicate directly with Clavister Security Gateways but instead interact with an *InControl Server* running on the same or a different computer. The server maintains a central database containing the configurations of connected security gateway and mediates all communications between clients and individual gateways.

The InControl server performs the same functions that it does with the standard Clavister InControl client. However, the Clavister client is replaced with a non-Clavister client which is written using the API.

The API Implementation

The InControl API implementation is based on the Microsoft *Windows Communication Foundation* (WCF) which makes use of *Simple Object Access Protocol* (SOAP) to pass objects between a *WCF Service* and a *WCF Client*. With the InControl API, the InControl server provides the *WCF Service* function and the application written with the API acts as the *WCF Client*.

This guide is not designed to be an introduction to WCF programming and it is therefore recommended to refer to other sources for an in-depth understanding of general WCF principles. This guide will assume that the reader understands basic WCF concepts.

InControl Server Setup and Licensing

Basic installation and administration of the InControl server is not discussed in this manual. This can be found in the separate *InControl Administration Guide*. InControl API based applications can run on either the same PC as the server or on a different computer.

The InControl server comes as standard with the API capability. However, this is only enabled if a valid *InControl Server License* is installed on the server.

The current version of the InControl API does not make use of any user authentication to access the InControl server. Therefore, the InControl API is disabled by default, and needs to be enabled in the InControl Server configuration file before any access to the API can be made.

To enable the InControl API, edit the InControl Server configuration file *ICS.exe.config* which resides in the target installation directory of the InControl Server (i.e. *"C:\Program Files\Clavister\InControl\Server"*). Locate the tag *<PluginHost API="false" />* and change *"false"* to *"true"* so that the resulting text block looks like this:

```
<PluginHost API="true">
```

Note: Avoiding unauthorized access

To avoid unauthorized access to the InControl API, it is important that you ensure that the InControl API port (TCP 33726) is not reachable from any other hosts than those where the trusted API clients reside. Future releases of the InControl API will include user authentication which removes this limitation.

The Capabilities of the InControl API

Applications that make use of the API are capable of the same range of functions that the standard InControl client is capable of. That is, they may check out, examine, edit and check in the configurations of any Clavister Security Gateway reachable via the InControl server.

API Reference Materials

Although programming examples are discussed in this guide, the full API reference documentation is not included. This is provided in a more convenient Windows Help format as a separate file in the InControl SDK package.

Compatible Development Environments

The code examples included with the SDK are written using the *C Sharp* (C#) programming language. However, since the InControl API is based on WCF, any development platform capable of supporting this standard could be used for application development with the API.

A description with screen shots for setting up the InControl API with Microsoft's *Visual Studio* development environment can be found in *Appendix A, Visual Studio Setup*.

Chapter 2: Starting Coding

This chapter starts examining coding examples which make use of the InControl API.

Similarities with the CLI

A useful principle to remember is that programming with the InControl API is very similar to constructing CorePlus *Command Line Interface* (CLI) commands. The two are similar since they both interact with the *Configuration Engine* which is the CorePlus subsystem that deals with the management of CorePlus configurations.

Similarities to the CLI will be discussed further in this guide and example CLI commands will sometimes be given following InControl API based code in order to highlight the similarities. The *CLI Reference Guide* can be an extremely useful reference document when programming with the API since it lists all possible API parameter options.

A key difference between using the CLI and the InControl API is that an equivalent to the "*cc*" (change context/category) command does not need to be used to change the current context to be a particular object such as the *main* IP rule set when editing IP rules. Instead, the InControl API uses a more direct way of referring to a particular configuration object.

A Simple Example

We will first look at a simple first example of API based source code. This example, like others used throughout this guide, is written in C Sharp (C#) and assumes that your development environment is setup accordingly with references to the WCF service (see *Appendix A, Visual Studio Setup*).

```
// Open the connection from the client to the server

ChannelFactory<IRemoteServer> channelFactory =
                new ChannelFactory<IRemoteServer>(new NetTcpBinding(),
                "net.tcp://localhost:33726/InControl");

IRemoteServer server = channelFactory.CreateChannel();

// Get the root domain which contains all defined security gateways

Domain global = server.get_Root();

// Assume we have a device in the root domain called 'sgwStockholm'
// Get this security gateway

Device sgwStockholm =
        server.GetConfigObjectByName(global, "sgwStockholm") as Device;

// Check out this security gateway
```

```
server.CheckOutConfiguration(sgwStockholm);

// Get the latest configuration for this gateway

Configuration cfg = server.GetLatestConfiguration(sgw);

// Add a new IP rule
// Start by setting the properties of the rule

Dictionary<string, string> properties = new Dictionary<string, string>();

properties.Add("Action", "NAT");
properties.Add("SourceInterface", "lan");
properties.Add("SourceNetwork", "lannet");
properties.Add("DestinationInterface", "wan");
properties.Add("DestinationNetwork","all-nets");
properties.Add("Service", "http-outbound");
properties.Add("LogEnabled", "True");
properties.Add("Comments", "Allow and NAT HTTP traffic from LAN to WAN");

// Get the main IPRuleSet folder to add the node to it
// Note how the ruleset identifier is constructed

Node folder = server.GetNode(cfg, "main.IPRuleSet");

// Add the IP Rule as a child node to this folder

server.AddChildNode(folder, "IPRule", "NAT_HTTP", properties);

// Check in the new configuration

server.CheckInConfiguration(cfg,
                "Added IP rule allowing NAT LAN to WAN HTTP traffic");

// Deploy and activate the new configuration on the security gateway

server.DeployConfiguration(sgwStockholm);

// Finished so close the connection to the InControl server

(server as IClientChannel).Close();
```

Various aspects of this code will be discussed in detail in this and subsequent chapters. In this chapter, we will look next at the code which opens and closes the connection to the InControl server.

Establishing Server Connection

To set up the initial connection to the InControl server, the server must first be reachable at a given IP address. The code for establishing communications with the server, as well as closing them, is as follows:

```
ChannelFactory<IRemoteServer> channelFactory =
                new ChannelFactory<IRemoteServer>(new NetTcpBinding(),
                "net.tcp://localhost:33726/InControl");

IRemoteServer server = channelFactory.CreateChannel();
        "
        "
(server as IClientChannel).Close();
```

The initial steps in this code are:

- Assuming that the server is running on the same Windows computer as the InControl server we can connect to *Localhost* instead of an IP address and the C# source code for this is:

```
ChannelFactory<IRemoteServer> channelFactory =
                new ChannelFactory<IRemoteServer>(new NetTcpBinding(),
```

```
                "net.tcp://localhost:33726/InControl");
```

Where the server is not on the same computer, *Localhost* should be replaced with the relevant IP address.

- Once the *ChannelFactory* object instance is defined, the InControl server itself is retrieved as an instance of *IRemoteServer*.

```
IRemoteServer server = channelFactory.CreateChannel();
```

The object named *server* will be used extensively throughout the remaining code examples in this guide.

Note

*Error handling **try** and **catch** blocks are not explicitly shown in these code examples but should be used following normal error handling practices to catch errors.*

Closing the Server Connection

Before ending an InControl API based application, the connection to the InControl server should be closed when the server is no longer required. The code to perform this operation is:

```
(server as IClientChannel).Close();
```

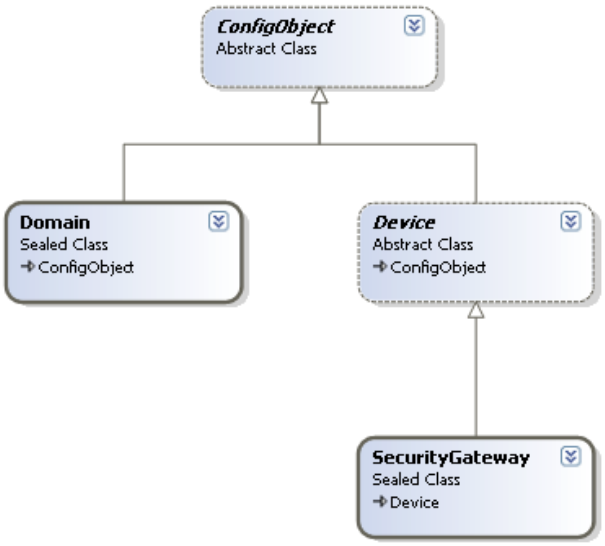
This frees up allocated resources on the server.

In the following chapter we will examine the remaining parts of the example program above and other important operations performed with the InControl API.

Chapter 3: Opening Configurations

This chapter discusses how to obtain and check out a CorePlus configuration in preparation for editing.

It is useful at this point to look at the relationships between the classes involved in obtaining a CorePlus configuration.



The C# code to retrieve a *Configuration* object is as follows:

```
Domain global = server.get_Root();

Device sgwStockholm =
       server.GetConfigObjectByName(global, "sgwStockholm") as Device;

server.CheckOutConfiguration(sgwStockholm);

Configuration cfg = server.GetLatestConfiguration(sgwStockholm);
```

Let us now break the steps in this code down.

Checking Out Gateways

Once the *Configuration* object is obtained we can perform a check out of the security gateway in order to start editing the configuration. This is a suggested ordering and the check out of the

gateway could be done before obtaining the configuration.

Note

Checking out a security gateway is not necessary if the operations performed on the configuration only involve reading the configuration's contents. If modifications are to be made then the security gateway has to be checked out.

- Get the root domain of the server which will provide the methods to access individual security gateways:

```
Domain global = server.get_Root();
```

- Assuming that there is a Clavister Security Gateway in the root domain named *sgwStockholm*, get the *Device* object for this gateway:

```
Device sgwStockholm =
        server.GetConfigObjectByName(root, "sgwStockholm") as Device;
```

  At this point the application is able to read configuration information but cannot change it until a check out operation is performed.

- Check out this security gateway by invoking the *CheckOutConfiguration* method of the *Device* class.

```
server.CheckOutConfiguration(sgwStockholm);
```

  After the checkout, the application will have exclusive access to the security gateway's configuration and no other client will be able to check it out.

Checking back in the security gateway is discussed in *Chapter 5, Check In and Deployment*.

Opening a Configuration

Before we can work with a configuration we must obtain a specific version, usually the latest, as a *Configuration* object instance. To get the latest version we would use the code:

```
Configuration cfg = server.GetLatestConfiguration(sgw);
```

This code retrieves the latest version of the specific security gateway's configuration. The InControl server maintains a database of past versions and it is also possible to get a list of all available versions and select a particular version.

Chapter 4: Editing Configurations

This chapter looks at example InControl API based code for performing typical editing operations on a CorePlus configuration. Assuming that we have checked out a configuration, we will examine how typical editing operations are can be performed on a CorePlus configuration.

Adding an IP Rule

Let us first look at how a new IP rule is defined and examine in more depth some of the code used in the code example at the beginning of *Chapter 2, Starting Coding*.

IP rules define what traffic is allowed or dropped as it enters the security gateway through a particular interface (the source interface) and exits another interface (the destination interface), and that comes from a particular network (the source network) going to a particular network (the destination network).

In this example, taken from the code example in the previous chapter, we will allow traffic from the network *lannet* which is connected to the *lan* interface to flow to the Internet. The Internet is connected to the *wan* interface and the destination network is *all-nets* (in other words, any network).

The required IP rule can be summarized as follows:

Action	Src Interface	Src Network	Dest Interface	Dest Network	Service
NAT	lan	lannet	wan	all-nets	http-outbound

The code to add this rule is:

```
Dictionary<string, string> properties = new Dictionary<string, string>();

properties.Add("Action", "NAT");
properties.Add("SourceInterface", "lan");
properties.Add("SourceNetwork", "lannet");
properties.Add("DestinationInterface", "wan");
properties.Add("DestinationNetwork", "all-nets");
properties.Add("Service","http-outbound");
properties.Add("LogEnabled","True");
properties.Add("Comments", "Allow/NAT HTTP traffic from LAN to WAN");

Node folder = server.GetNode(cfg, "main.IPRuleSet");

server.AddChildNode(folder, "IPRule", "Example_Drop_Rule", properties);
```

Breaking the code down, let us examine the individual statements:

- First, create a new *Dictionary* object called *properties*:

```
properties = new Dictionary<string,string>();
```

- We now add the various parameters to this *Dictionary* object instance starting with a comment:
- Define the source interface. In this case it is *any* network:

```
properties.Add("SourceInterface", "lan");
```

The parameter name *SourceInterface* and all possible parameters for the rule can be found under *IPrule* in the *CorePlus CLI Reference Guide*.

- Define the source network. In this example it is *all-nets* which means it can be any network:

```
properties.Add("SourceNetwork", "lannet");
```

- Define the destination interface. This could be a physical interface or perhaps a VPN tunnel (which is treated like a physical interface). In this example we will use an interface called *wan* which might be connected to the public Internet:

```
properties.Add("DestinationInterface", "wan");
```

- Specify the destination network:

```
properties.Add("DestinationNetwork", "all-nets");
```

- Then specify the service:

```
properties.Add("Service","http-outbound");
```

- Enable logging on this rule so that a log message can be generated when it triggers:

```
properties.Add("LogEnabled","true");
```

- As the last property, add a comment to say what the rule does:

```
properties.Add("Comments", "Allow/NAT HTTP traffic from LAN to WAN");
```

- In preparation for adding the rule, get the *Node* instance which is the *main* IP rule set.

```
Node folder = server.GetNode(cfg, "main.IPRuleSet");
```

Notice how we specify the rule set by using the suffix *IPRuleSet* to qualify the name. If we wanted to add a new IP rule set called, for example *User-rules*, we would use the code:

```
Node new_ruleset =
        server.AddChildNode(root, "IPRuleSet", "User-rules", null);
```

- Finally, add this rule to the default *main* rule set:

```
server.AddChildNode(folder, "IPRule", "NAT_HTTP", properties);
```

The *NAT_HTTP* parameter will be the symbolic name of the rule used in the configuration.

The equivalent CLI command would be:

```
Device:/> add IPRule Name=NAT_HTTP Action=NAT SourceInterface=lan
                SourceNetwork=lannet DestinationInterface=wan
                DestinationNetwork=all-nets Service=http-outbound
                LogEnabled=True
```

Again, we can see that referring to the CLI command can provide us with the correct parameters that need to be specified when using the InControl API.

Adding an IP4 Address Folder

Let us assume we need to create a new folder called *InternalServersFolder* to collect together in one place a group of IP4 addresses which are all related to internal servers. We can create the folder with the following code:

```
Dictionary<string, string> properties = new Dictionary<string, string>();
Node folder = server.AddChildNode(server.GetRootNode(cfg),
                "AddressFolder","InternalServersFolder", properties);
```

The *Node* object called *folder* can now be used in the next step when we add an address to it.

Let us move on to one of the more common operations performed with CorePlus configurations which is manipulating the *Address Book*. This is where all the symbolic names for IP addresses that CorePlus uses are defined along with their associated IP addresses. Some default address book objects are defined by CorePlus, others may have to be added.

Adding an IP4 Address Object

Next, let us first look at how we add a new IP4 address object to the configuration's address book. Let us assume we want to add a new IP for a web server with the symbolic name *webserver_ip* and an IP address *10.53.95.1*.

```
Dictionary<string,string> properties = new Dictionary<string,string>();

properties["Address"] = "10.53.95.1";
properties["Comments"] = "Web Server Address";

server.AddChildNode(folder, "IP4Address", "webserver_ip", properties);
```

Let us examine the individual lines in this code:

- First, define a new *Dictionary* object called *properties*

  ```
  Dictionary<string,string> properties = new Dictionary<string,string>();
  ```

- Next, define the *Address* field of the object:

  ```
  properties["Address"] = "10.53.95.1";
  ```

- Then the *Comments* field:

  ```
  properties["Comments"] = "Web Server Address";
  ```

- Finally, we perform the add itself.

  ```
  server.AddChildNode(folder, "IP4Address", "webserver_ip", properties);
  ```

Here, we use the *Node* object called *folder* which was defined at the beginning of this chapter.

Let us now examine how this would be done through the CLI to see the similarity:

```
Device:/> add Address IP4Address webserver_ip
              Address=10.53.95.1 Comments="Web Server Address"
```

Tip

Thinking about how an operation would be performed with the CLI can often provide a framework for understanding how to do the same operation using the InControl API.

Changing Configuration Settings

Let us now look at changing some existing configuration settings. In this example, we will change the current values of the settings *TCPSequenceNumbers* and *TCPAllowReopen*. The code to do this is:

```
Dictionary<string, string> properties = new Dictionary<string, string>();

properties.Add("TCPSequenceNumbers", "Ignore");
properties.Add("TCPAllowReopen", "True");

Node folder = server.GetNode(cfg, "TCPSettings");

server.SetNodeProperties(folder, properties);
```

Let us look at the individual operations in this code:

- First, create a *Dictionary* object called *properties*:

  ```
  Dictionary<string, string> properties = new Dictionary<string,string>();
  ```

- Add the pairs of setting names and values which are to be set:

  ```
  properties.Add("TCPSequenceNumbers", "Ignore");
  properties.Add("TCPAllowReopen", "True");
  ```

- Now get the *Node* object where these settings are found, in this case *TCPSettings*:

  ```
  Node folder = server.GetNode(cfg, "TCPSettings");
  ```

- Apply the new properties to this node:

  ```
  server.SetNodeProperties(folder, properties);
  ```

In this example, the *CLI Reference Guide* can once again give us the correct naming for the *Node* object and its individual settings. *TCP Settings* is listed as a node (or object) name in the guide and all related settings are listed in that section of the guide.

Listing Configuration Items

To list out the contents of a particular node we can use the following code to enumerate the values and then display them on the console as a list.

```
foreach(KeyValuePair<string,string> item in server.GetNodeProperties(node))
        {
                Console.WriteLine(item.Key + ":\t" + item.Value);
        }
```

Deleting Configuration Items

Deleting a node in the configuration is simple:

```
Server.DeleteNode(node)
```

The *Attribute* Value and Deleting Related Objects

An *Attribute* value can be assigned to configuration objects so that all items with a particular value can be deleted at once. For example, the code above to add an IP rule could become:

```
properties = new Dictionary<string,string>();

properties.Add("Attribute", "user_A");
properties.Add("SourceInterface", "any");
        "
        "
server.AddChildNode(incoming, "IPRule", "Example_Drop_Rule", properties);
```

Where the string *user_A* will be assigned as the *Attribute* for all configuration objects related to this user.

Note

*The **Attribute** value is not definable with the CLI. The InControl API must be used.*

Chapter 5: Check In and Deployment

This chapter shows how changes made to an edited configuration can be saved and then activated.

As discussed in *Chapter 3, Opening Configurations*, configurations need to be checked out for editing. After editing is complete, a configuration needs to be checked back in and/or deployed to the Clavister Security Gateway.

The code example found in *Chapter 2, Starting Coding* achieves this with the code lines:

```
// Check in the new configuration

server.CheckInConfiguration(cfg,
                "Added IP rule allowing NAT LAN to WAN HTTP traffic");

// Activate the new configuration on the Security Gateway

server.DeployConfiguration(sgwStockholm);
```

Check In

A check in is a simple operation which is performed by invoking the *CheckInConfiguration* method on the *Server* object with the: configuration as a parameter.

```
server.CheckInConfiguration(cfg,
                "Added IP rule allowing NAT LAN to WAN HTTP traffic");
```

The string parameter *Example* is a comment for the check in.

Deployment

Checking in a configuration does not mean that the changes made will come into effect on the Clavister Security Gateway. It is necessary to *deploy* a configuration for it to become the active configuration on the security gateway. Deployment is done with the following code:

```
server.DeployConfiguration(sgwStockholm);
```

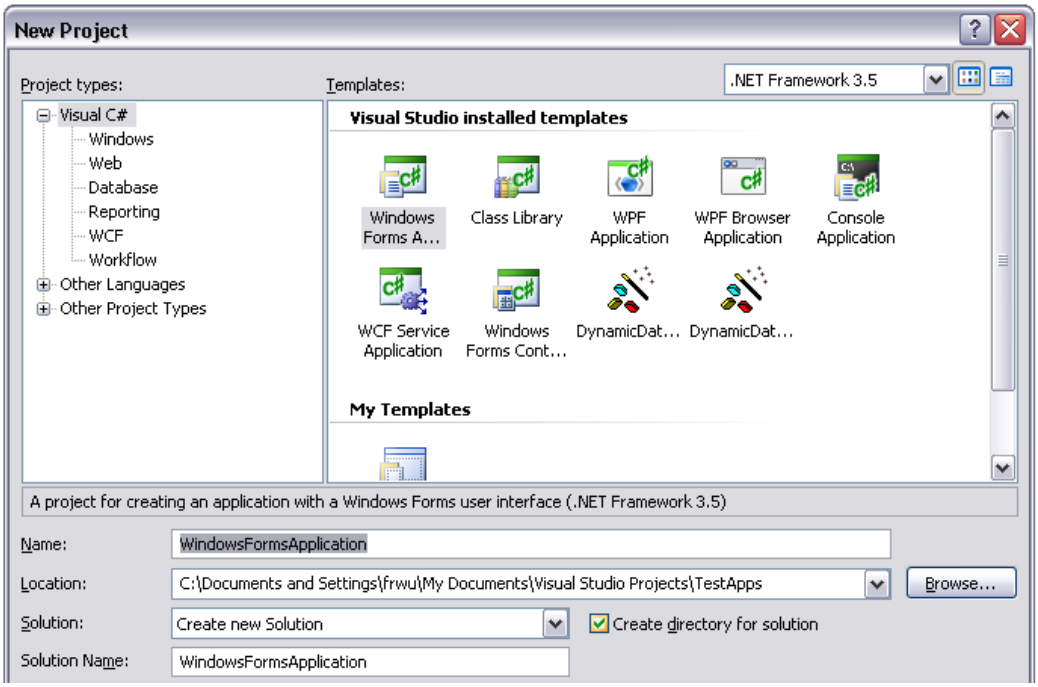
Notice that the configuration is not the parameter for the *DeployConfiguration* method but the *Device* object instance is used instead. In other words, the deployment is done for the security gateway and the most recent version of the configuration is deployed.

Appendix A: Visual Studio Setup

A common development environment that may be used to create source code with the InControl API is *Microsoft Visual Studio*. This appendix discusses how to set up Visual Studio when working with the InControl API.

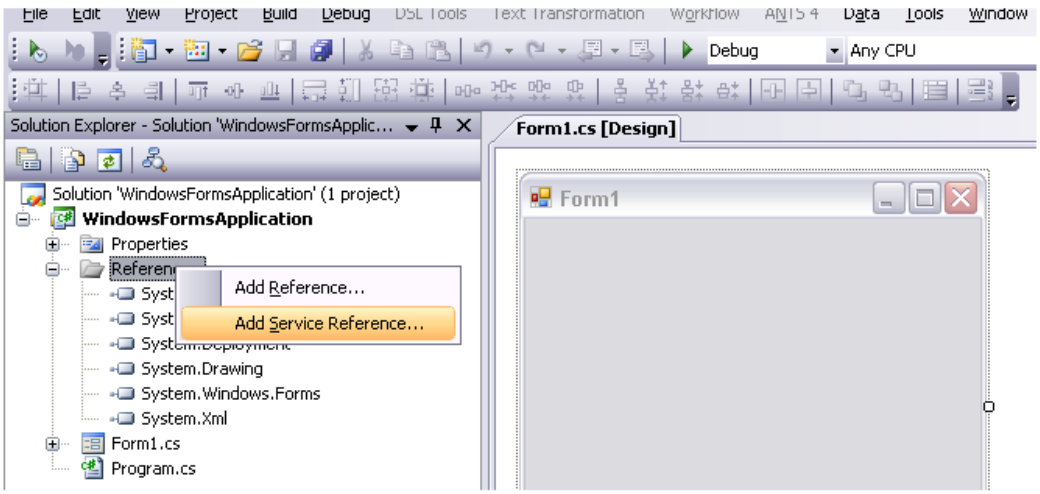
The steps for Visual Studio setup are as follows:

- Create a new Visual Studio project for code development. In this case we select a *Windows Forms Application*.



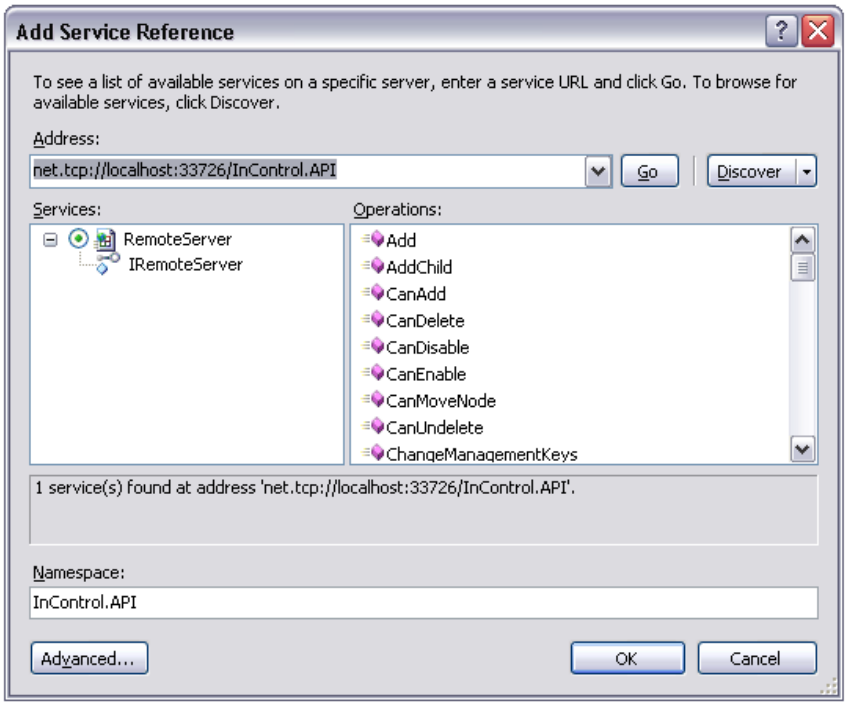
- Make sure the InControl server is running, is accessible and has an InControl server license that has the InControl API option enabled.

- Add the InControl API as a *Service* in the project. Do this by selecting the **Add Service Reference** option after right clicking the **References** tree node.

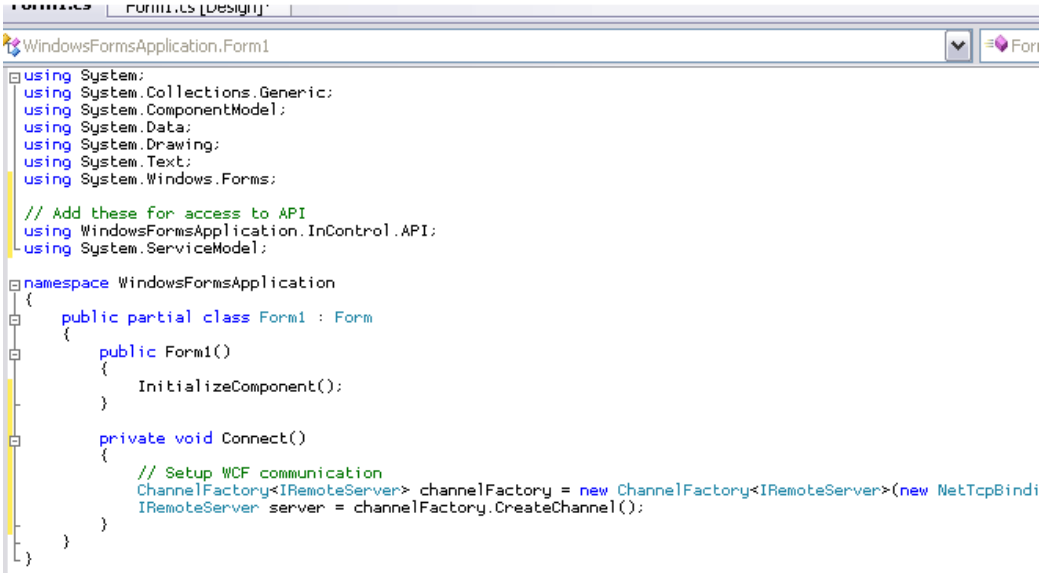


- In the *Add Service Reference* dialog, specify the full URL address of the InControl API. The

address is constructed in the form *net.tcp://localhost:33726/InControl.API* where *localhost* should be substituted with the relevant IP address or hostname of the InControl server The *Namespace* to use is called *InControl.API*.



- Visual Studio will contact the InControl server and download the relevant *Web Service Definition Language* (WSDL) file which describes the API.
- Make sure references to the new namespace and the *System.ServiceModel* is added in your source files according to the screenshot below.



Clavister AB
Sjögatan 6J
SE-89160 Örnsköldsvik
SWEDEN

Phone: +46-660-299200
Fax: +46-660-12250

www.clavister.com