DataAccess API Reference

Programmatically access data in FrameworX.

DataAccess is an API that empowers you with the capability to access classes and methods, facilitating communication between projects.

This API allow to create external programs within Visual Studio, enabling them to effectively interact with a specific project.

Example 1
Example 2
- Summary
- Technical Information
- Setting and Getting a Value:
- Running a method in the server:
- Running Event Handlers on the Server:

Example 1

```
        string remoteIP = "127.0.0.1";
        object server = null;
                DataAccess access = new DataAccess();

                 server = access.SetServer(remoteIP, "Administrator", "");
                 if (server is string)
                 {
                        @Info.Trace("OfficeServices::Connect - Error: " + server);
                        access = null;
                 }

    bool sus = access.SetCurrentValue("Tag1", 123, 192, DateTime.UtcNow, server, 0, false);
```

Where:

- remoteIP can be substituted with the IP address of the remote computer, making it versatile and adaptable to different connections.
- DataAccess instantiates a new object to access methods provided by this API, granting access to its functionalities.
- access.SetServer establishes a connection to the associated project while utilizing the "Administrator" user. An empty password ("") signifies that no password authentication is required for this user.
- access.SetCurrentValue assigns a value of 123 and quality of 192 to a designated "Tag1," which is typically a tag created within the client project.

Example 2

Summary

This solution is an example on how to connect two different solutions through CodeBehind.

Technical Information

Since we are connecting two solutions, one must act as the server and the other as the client, and both need to run simultaneously.

First, we need to connect to the server, which will be hosted at `127.0.0.1:3201`. Only after establishing this connection will the other buttons become accessible. As we will use the same server throughout the solution, it is set as a global variable.

```
private object server = default;

private DataAccess access = new DataAccess();

public async Task ConnectServer(object sender, System.Windows.Input.InputEventArgs e)

{

       string remoteIP = @Tag.IPAddress + ":" + @Tag.Port;

       server = access.SetServer(remoteIP, "Administrator", "");
```

```
	if (server is string) {

		@Info.Trace("OfficeServices::Connect - Error: " + server);

		access = null;

	}
}
```

Setting and Getting a Value:

Now, to change a tag's value, we need to create a getter and a setter. Creating a setter is straightforward—you simply use the `SetCurrentValue` method with the following parameters:

- **Object (As Object):** The object whose value needs to be changed, e.g., `"Tag1"`.
- **Value (As Object):** The value to be set, e.g., `254`.
- **Quality (As Integer):** The quality status of the tag, typically set to `192` for success and `64` for failure.
- **Timestamp (As DateTime):** The time when the tag was changed, usually set to the current time, e.g., `DateTime.UtcNow`.

Note: This method has different overloads and accepts additional optional parameters, such as:

- **Tag Protection:** Determines whether the tag has specific protection settings.
- **Server's IP:** Required if connecting to a second server or if the connection hasn't been established previously.
- **Force Value:** Specifies whether the value should be forced regardless of conditions.

The implementation is shown in the code snippet below:

```
access.SetCurrentValue("Tag1", @Tag.Setter, 192, DateTime.UtcNow);
```

And for the getter, the parameters used are:

- **Object (As Object):** The object whose value will be retrieved.
- **Out Output (As Object):** The retrieved value.
- **Out Quality (As Integer):** The quality status of the retrieved value.
- **Out Timestamp (As DateTime):** The timestamp of the tag when it was retrieved.

Note: This method also allows the server's IP as an optional parameter.

Running a method in the server:

To execute a method from a class on the server, you first need to create a class containing the desired method on the server. In this example, we'll create a class named `Math` with a method called `Counter`.

To run this server method, we'll be using the `ExecuteClassMethodAsync` method. This approach works for methods with and without a return value. The parameters used for such method are:

- **ClassName (As String):** This is where you specify the name of the class.
- **MethodName (As String):** This is where you specify the name of the method to call.
- **Parameters (As Object[]):** This is where you pass the parameters; it must be in an object array to accommodate parameters of different types.
- **Error (As TRef<string>):** This is the method's secondary return, which will provide an error message if any issues occur.

In the solution, we'll use an integer-returning method, therefore our `Counter` method will have the following structure:

```
public int Counter() { @Tag.Counter++; return @Tag.Counter; }
```

And our `ExecuteClassMethodAsync` call will look like this:

```
int methodReturn = TK.To<int>(await access.ExecuteClassMethodAsync("Math", "Counter", null, err, server));
```

Note: Since the method returns an object, it must be converted to an integer before assigning it to a tag. Alternatively, you can use `object` as your variable type to handle the returned value directly.

Running Event Handlers on the Server:

Our solution will run three different Event Handlers. Since their usage is identical, we will explain only one of them, with its code shown below:

```
public async Task EventBindToSecond(object sender, System.Windows.Input.InputEventArgs e) { if(eventIDSS ==
null) { eventIDSS = access.BindEvent("Server.Second", null, this.BindToSecond, server); @Tag.BoolSecond = 1; } }
```

The code above triggers the EventHandler *BindToSecond* every time a `Server.Second` is passed in our Server solution, and it returns an object containing the event key. This key is essential if you need to unsubscribe from the event later, as shown in the code below:

```
public async Task EventUnbindToSecond(object sender, System.Windows.Input.InputEventArgs e) { access.
UnbindEvent(eventIDSS, server); eventIDSS = null; @Tag.BoolSecond = 0; }
```

Note: Since it is likely binding and unbinding the event will happen in different scopes, it is important to ensure the event key is accessible in both places.

In this section...
