Schneider Electric

SmartConnector Developers Guide

EcoBuildings EcoStruxure Labs

1 Revision History

Date	Author	Revision	Changes Made
10/21/2014	MRS	1	Initial release
02/16/2015	MRS	2	Updated for v1.3
09/30/2015	MRS	3	Updated for v2.0
04/04/2016	MRS	4	Updated for v2.1

2 Overview

2.1 Scope

This document is intended as a guide for developers authoring extensions using SmartConnector – the Windows service middleware framework developed by EcoBuildings EcoStruxure® Labs. This document assumes the reader has the requisite knowledge of C# or VB.NET® and is moderately comfortable developing class libraries in Visual Studio® .NET.

This document will not cover the details pertaining to installation, configuration, monitoring, and control of SmartConnector in a runtime environment. That information can be found in "SmartConnector Installation and Configuration Guide".

2.2 Description

When developing solutions there is frequently a need for software that can bridge the gap between Schneider Electric Building Management Systems (BMS) and third party systems and data sources. This software goes by varying names: protocol shims, glue logic or more generally, middleware. As different projects are analyzed, patterns begin to emerge where this middleware performs similar actions with only minor variations from solution to solution. SmartConnector was conceived to be this middleware framework.

3 How SmartConnector Works

SmartConnector is an extensible and configurable application framework. At its simplest, SmartConnector is a multi-threaded Windows service. Threads are configured and scheduled or commanded to execute a Processor with a predefined set of inputs; a Processor Configuration. While SmartConnector does include some sample Processor classes for operational validation, Processors are typically written by others using SmartConnector's public libraries and packaged as SmartConnector Extensions.

The SmartConnector Runtime includes an OData API which provides a full CRUD and other custom actions that allow users to configure, monitor, and control SmartConnector using any OData compliant client.

The SmartConnector Runtime includes EcoStruxure Web Service (EWS) Serve capabilities as well. SmartConnector can serve multiple logical EWS endpoints configured in a multi-tenant environment. This means that the data for one EWS server is logically separated from all other EWS servers. Also included is a data adapter framework so solutions can be developed to pull in data from one or more different sources, consolidate it, scrub it and then serve it up in the EWS v1.2 common data model – to SmartStruxure for example – via a standard EWS client connection.

Figure 1 shows the general architecture of SmartConnector.

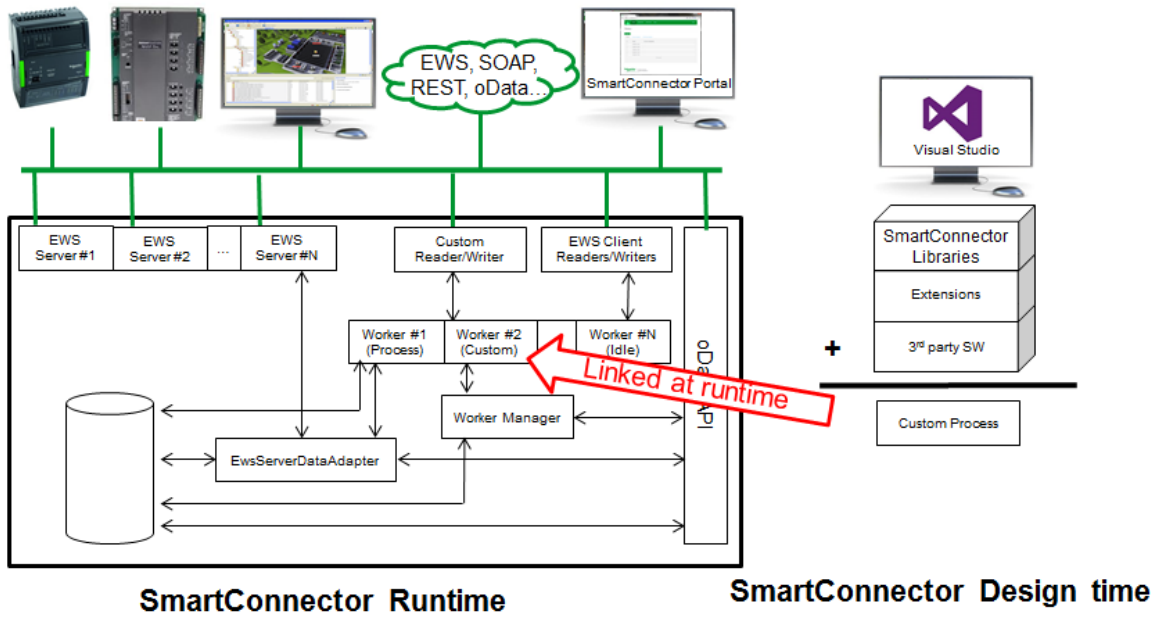

Figure 1: SmartConnector General Architecture

3.1 Processor

The Processor is the core of the runtime execution of SmartConnector. Processors are .NET classes which do the work required in an application solution. Work can be as simple or as complicated as needed to meet the application's requirements. Multiple Processors can also be combined to perform work asynchronously in a cooperative manner.

3.2 Processor Configuration

While a Processor defines how work is accomplished, a Processor Configuration specifically dictates what gets accomplished. A Processor Configuration contains all of the information needed to instantiate a class at runtime, hydrate all properties of that class, and validate that everything is correct before finally executing the Processor. A Processor Configuration also links to the requisite scheduling information which determines how often SmartConnector will run a Processor and when.

3.3 Schedules

Schedules can be defined to execute a Processor based on an interval (second, minute, hour, or day granularity), weekly, or monthly basis. One Schedule can be used by any number of Processor Configurations.

3.4 EWS Server

SmartConnector can serve multiple logical EWS v1.2 compliant servers. A data management adapter class is included in SmartConnector's core set of libraries. This adapter allows a Processor to interact with the persistent data store for all supported CRUD actions in a consistent manner.

3.5 Extension

Extensions are class assemblies, written by others, to solve an application problem. An extension may contain one or more custom Processor classes and/or one or more EWS Servers.

3.6 Workers

Workers represent the threads in the SmartConnector Runtime that execute Processors. The number of available Workers is configurable but is generally limited by the host system hardware. When not actively running a Processor, Workers are inactive, waiting for a command from the Worker Manager. In this state, Workers consume virtually no system resources.

3.7 Worker Manager

The Worker Manager is responsible for selecting a Processor Configuration, instantiating its defined Processor and passing it off to an idle Worker for execution. The Worker Manager also listens to external input to start or stop a Processor or EWS Server as required.

3.8 Persistent Data Store

SmartConnector is backed by a SQL database to persist all manner of data including setup parameters, configuration data, schedule data, and EWS server data. SmartConnector also provides a Processor Values data store that Processors can access directly. This data store can be used to save state between run iterations of the Processor or to enable collaboration between multiple Processors.

3.9 In-Memory Cache

In addition to a persistent data store, SmartConnector provides a mechanism to have a volatile in-memory cache of data. A singleton class available to any Processor and EWS Server provides strongly typed access to any data the author wishes to store.

3.10 OData REST API

Communicating with the SmartConnector Runtime is done via an OData compliant REST API. Refer to the Appendix for details; however you are encouraged to read the entire guide prior to attempting to consume the API.

3.11 Logging

SmartConnector provides an integrated logging framework. Logging levels of Info, Status, Error, Debug, and Trace are extensively used throughout the SmartConnector Runtime and public libraries. The Logger is also available to Processor authors for adding their own log information to the common log file output.

4 Obtaining SmartConnector

SmartConnector's public libraries are distributed via NuGet packages. SmartConnector's NuGet packages are hosted in a private feed on www.myget.org. To obtain access to this feed, you need to register with the www.smartconnectorserver.com. Once you have authenticated on this site, click "Send a request to become a SmartConnector Developer" link and supply the required information. Once you are granted access you will receive an email from MyGet. Follow the instructions contained in the email.

4.1 Configuring the NuGet Source

Once you have gained access to the SmartConnector Feed, you should configure Visual Studio to always include this feed when searching for NuGet packages. Perform the following:

Note steps are based on Visual Studio 2015. Older versions may have slightly different steps

1. In Visual Studio, select the Tools-Options-NuGet Package Manager option.
2. Click Package Sources.
3. Click + to add a new source.
4. At the bottom of the dialog, enter a **Name** of "SmartConnector" and a **Source** of https://www.myget.org/F/mongoose/auth/YOUR_API_KEY/api/v2 where YOUR_API_KEY is your personal API key from MyGet.
5. Click Update.
6. The dialog should look similar to Figure 2.

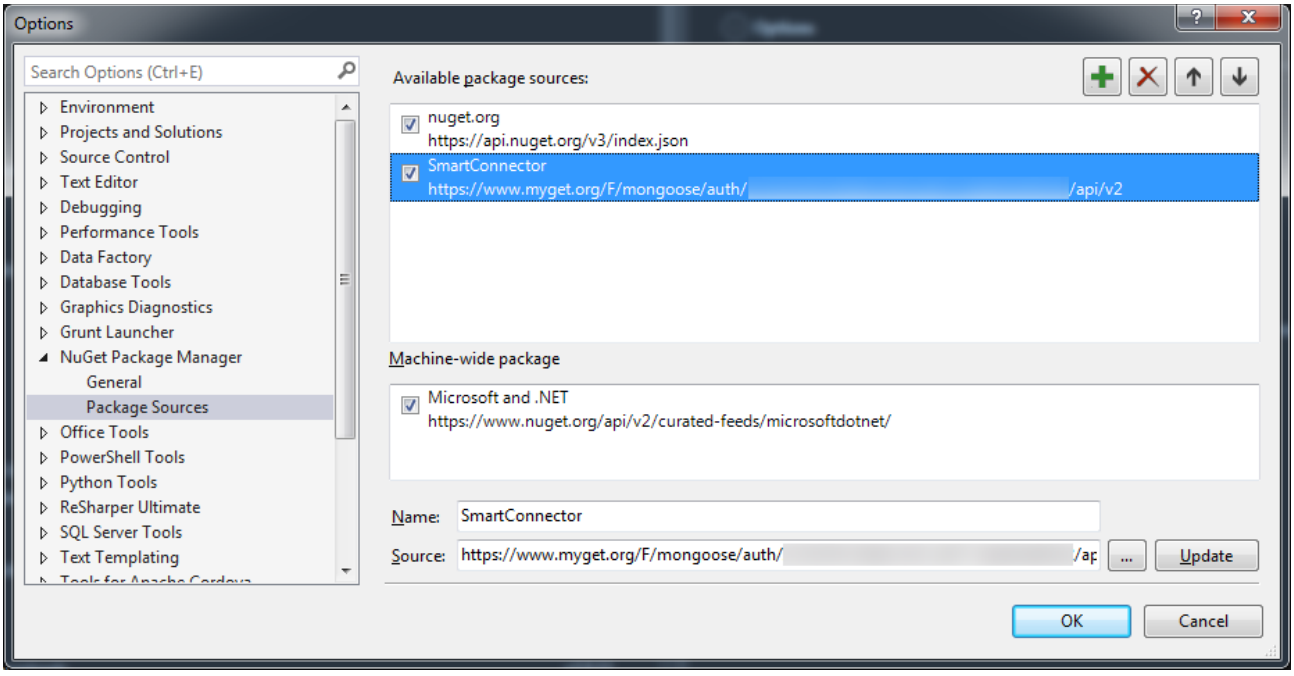

Figure 2: Package Manager Settings for SmartConnector NuGet Feed

7. Click OK.

4.2 NuGet Packages

SmartConnector currently provides numerous NuGet packages to facilitate Processor development and testing. However, most are simply dependent pieces of other core packages.

4.2.1.1 SxL.Common

This NuGet package contains low level support classes for other packages.

4.2.1.2 SxL.Licensing

This NuGet package contains helpers for licensing aspects.

4.2.1.3 Ews.Common

This NuGet package contains EWS data constructs which are commonly used when serving EWS or consuming a EWS endpoint.

4.2.1.4 Ews.Client

This NuGet package contains all classes and data proxy constructs needed to consume any EWS v1.1 (or later) compliant endpoint.

4.2.1.5 Ews.Server.Contract

This NuGet package contains a fully customizable MVC style EWS Server implementation.

4.2.1.6 Mongoose.Common

This NuGet package contains low level support classes for the SmartConnector Runtime.

4.2.1.7 Mongoose.Configuration

This NuGet package contains data classes for creating SmartConnector Processor Configurations.

4.2.1.8 Mongoose.Ews.Server

This NuGet package contains the SmartConnector standard EWS Server implementation. It can be customized as needed.

4.2.1.9 Mongose.Ews.Server.Data

This NuGet package contains classes for managing the internal data for a SmartConnector EWS Server.

4.2.1.10 Mongoose.Ews.Server.Sample

This NuGet package contains classes which illustrate how to extend and consume SmartConnector EWS Servers. This package will be reviewed in detail in a later section.

4.2.1.11 Mongoose.Process

This NuGet package contains the core SmartConnector framework library. When creating custom Processors, you will always need to include this NuGet package.

4.2.1.12 Mongoose.Process.Sample

This NuGet package contains classes which illustrate how to extend and consume SmartConnector Processors. This package will be reviewed in detail in a later section.

4.2.1.13 Mongoose.Process.Test

This NuGet package contains classes, extension methods and other helpers to aid in writing unit tests for Processors.

5 Sample NuGet Packages

In an effort to jumpstart development, special sample NuGet packages are available; these will configure a .NET class library to be a SmartConnector Extension. They will also inject a series of examples and NUnit test fixtures which can be executed to demonstrate how they operate. These packages can be used individually or in the same solution.

5.1 Installation

The following steps illustrate how to create an initial SmartConnector solution.

Note steps are based on Visual Studio 2015. Older versions may have slightly different steps

1. Start Visual Studio
2. Create a new Project. The project type should be a "Class Library".
3. After the project is initialized right click the References item in the Solution Explorer.
4. Choose Manage NuGet Packages.
5. Select the "SmartConnector" feed you configured above.
6. Confirm that "Include prerelease" is not checked.
7. Click Browse. You should see something like Figure 3.

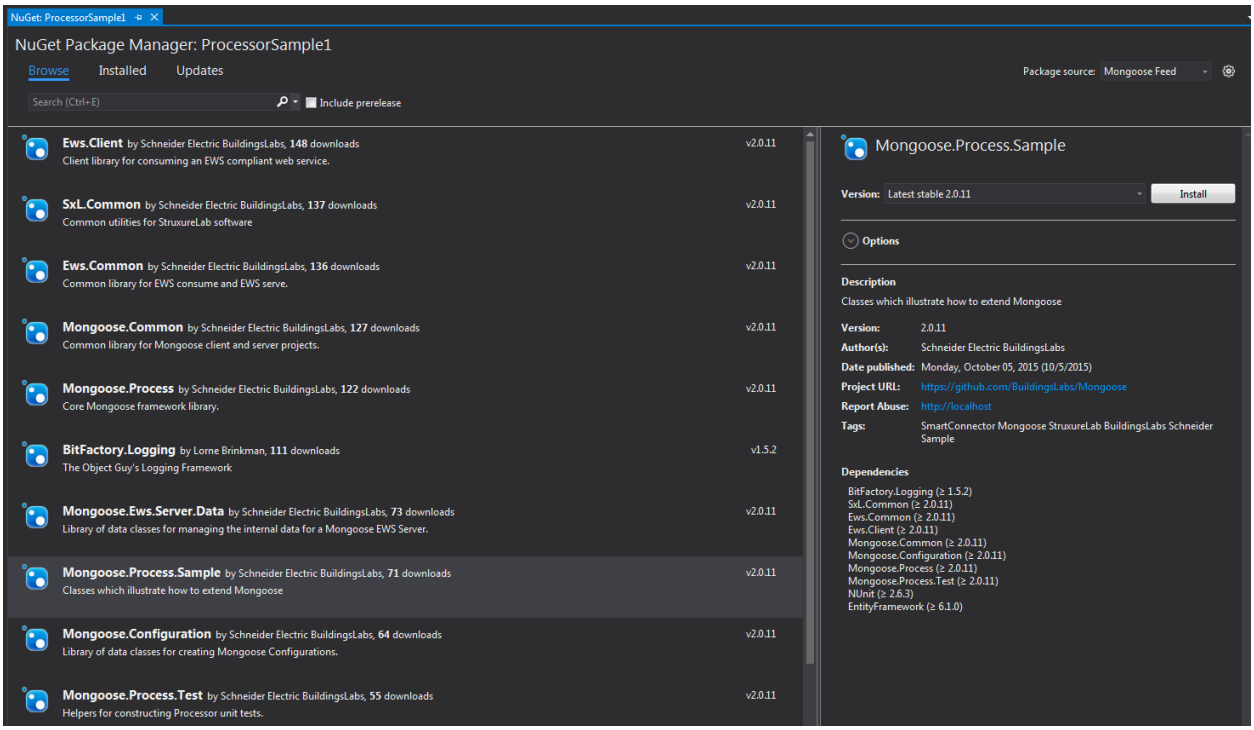

Figure 3: NuGet Package Manager Browsing SmartConnector Feed

8. Scroll down and select one of the available samples packages. You should be using the latest version of the package.
    a. "Mongoose.Process.Sample" will demonstrate how to create Processor classes.

b. "Mongoose.Ews.Server.Sample" will demonstrate how to customize the base EWS Serve capabilities.

9. Click Install.

At this point, Visual Studio will begin installing all of the referenced NuGet package and their defined dependencies (you may need to accept some license agreements for dependent packages). If all goes well, there should be a folder labeled "Samples\Mongoose". Build to confirm that everything was installed properly.

5.2 NUnit

The test fixtures distributed with the samples are based on the open source unit testing framework NUnit (www.nunit.org). This document does not describe how to write unit tests, however it does describe how to use NUnit with the samples provided. If using a third party productivity add-on for Visual Studio which includes a test runner (such as ReSharper), skip to Running the Samples.

All necessary NUnit references in the samples are included as NuGet package dependencies. However, in order to execute these tests the NUnit test runner is required. The NUnit test runner can be found at http://www.nunit.org/index.php?p=download. Only the appropriate version for one's current development environment should be downloaded and installed.

Note: At this time, SmartConnector samples support NUnit through v2.6.x.

After you have installed NUnit, you can link your Visual Studio class library project to it by configuring your project as follows:

1. Build the project. The binaries must exist in order to complete these steps.
2. Open the properties page for the class library project.
3. Click the Debug tab.
4. Select "Start External Program".
5. Click the ellipsis button and navigate to the location where NUnit was installed.
6. In the command line arguments, enter the complete path to the DLL which is generated from the project. When completed, properties should look something like Figure 4.

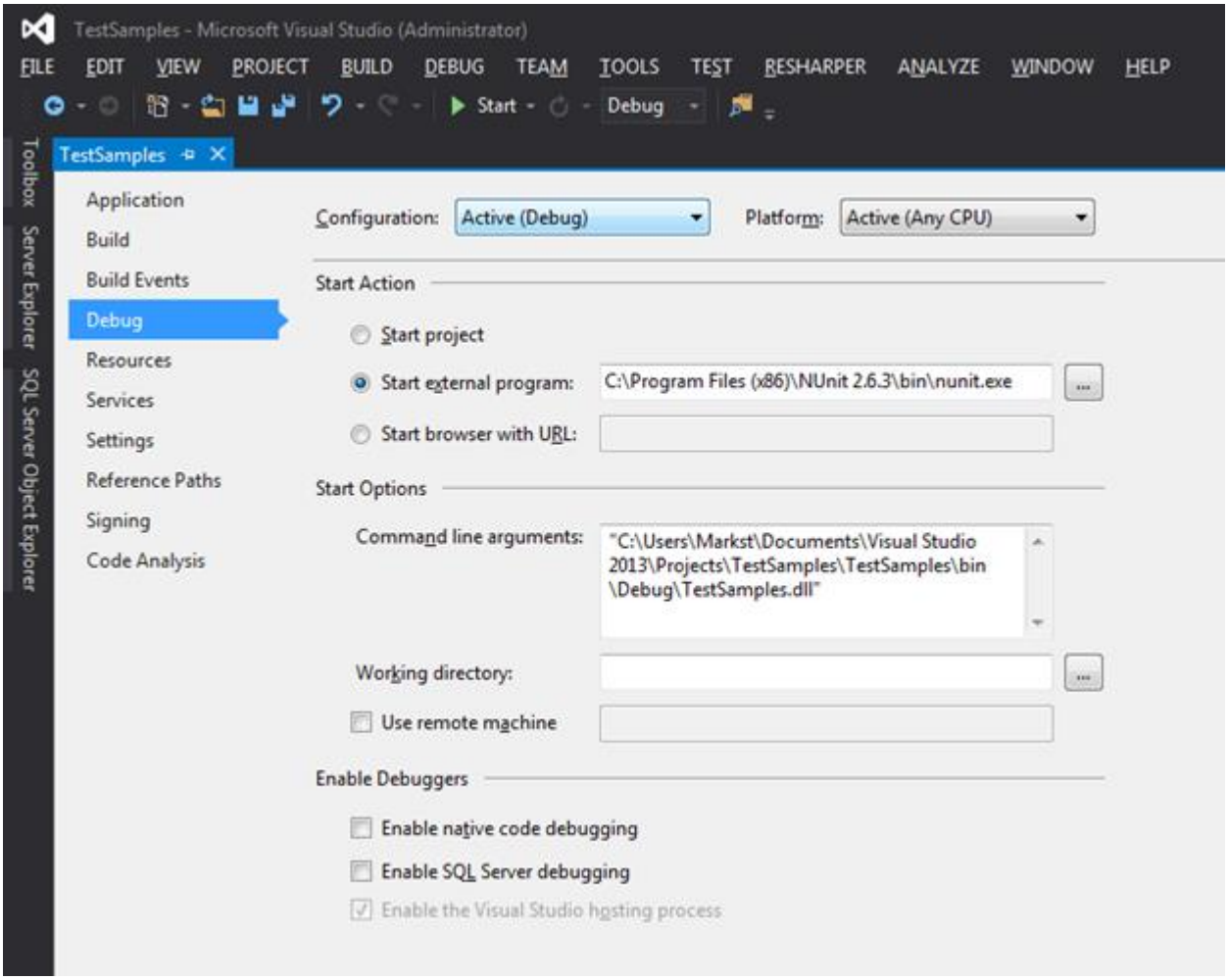

Figure 4: Test Project Debug Properties

7. Run your project.

At this point, NUnit should open and load the project into the “runner”. The result should look like Figure 5.

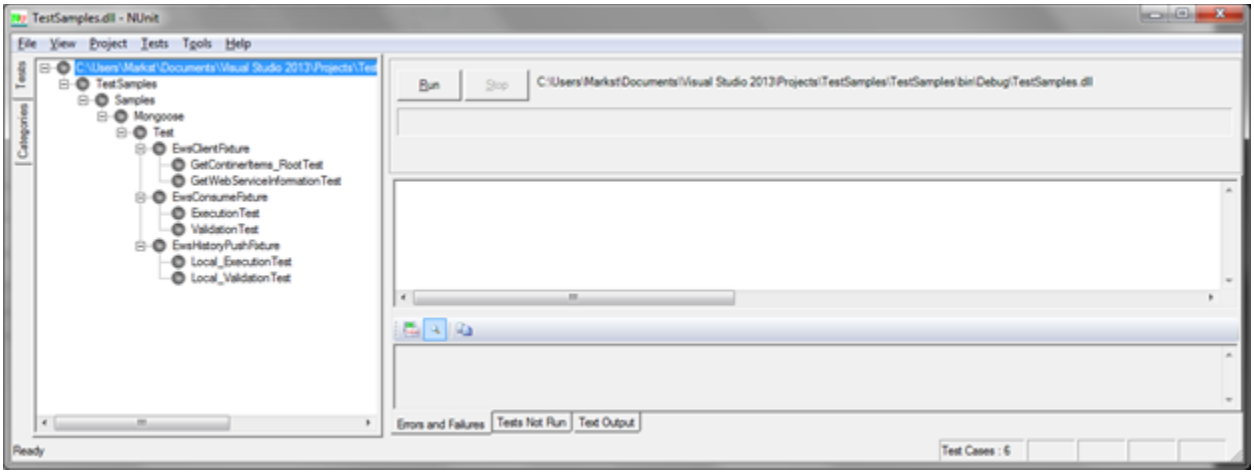

Figure 5: NUnit Runner

If your project configuration is setup for "Debug", breakpoints in your code will not be immediately honored. The issue is well documented on Stack Overflow (http://stackoverflow.com/questions/3076807/vs-2010-nunit-and-the-breakpoint-will-not-currently-be-hit-no-symbols-have-b). The solution is to add the following snippet to the app.config of NUnit. The section can be added immediately after the opening configuration tag.

```
<startup>
    <requiredRuntime version="4.0.30319" />
</startup>
```

After performing this step, all breakpoints in the sample code will be honored when executing the test runner from the IDE.

5.3 SmartStruxure™ Solution

Many of the samples communicate with a EWS endpoint. This can be any EWS v1.1 (or later) compliant endpoint. However, the assumption is that EWS endpoint is a SmartStruxure solution Enterprise Server (ES).

The NuGet package provides an XML export file which should be imported into SmartStruxure solution in order to properly execute the included sample code.

5.4 Running the Samples

The best way to exercise the sample code is by running the included unit test fixtures in NUnit. Fixture specific instructions follow.

5.4.1.1 EwsClientFixture (Mongoose.Process.Sample)

This sample illustrates how to make EWS consume calls with the Ews.Client library that is included in SmartConnector. While the creation of unique proxy references to EWS is possible, the use of the references premade in the aforementioned library is encouraged.

In order to connect from the test fixture to the EWS server, first the `UserName`, `Password`, and `EwsEndpoint` constants in this fixture must be modified to match the new system.

```
private const string EwsUrlEndpoint = "http://localhost:8081/EcoStruxure/DataExchange";
private const string EwsUsername = "TODO";
private const string EwsPassword = "TODO";
```

Once the connection information has been updated, the tests can be debugged to exercise the various methods supported.

5.4.1.2 EwsConsumeFixture (Mongoose.Process.Sample)

SmartConnector includes many helpful abstractions to aid in developing processor classes. For EWS, there are `ValueItemReader`, `ValueItemWriter`, `HistoryItemReader`, `AlarmItemReader`, and `SubscriptionReader` to make calling EWS simpler from the code writer's perspective. As described above, once the processor is running SmartConnector will first confirm that all declarative and semantic validation requirements are met. The programmer's job is to author SmartConnector Processors such that the declarations included meet validation requirements. The `ValidationTest` in this fixture reviews how the processor code's validation can be tested prior to being deployed.

Much like the SmartConnector Runtime, the tests in this fixture use a Processor Configuration to instantiate a specific processor and load it with the data needed to perform the desired task. Prior to executing the tests in this fixture, you must modify the connection information within the provided configuration must so that SmartConnector can connect to the EWS endpoint.

Find the `EwsConsumeConfiguration` static method in the `SampleConfigurations` class and review the Parameter definitions particular to the EWS server as shown below.

```
private const string EwsUrlEndpoint = "http://localhost:8081/EcoStruxure/DataExchange";
private const string EwsUsername = "TODO";
private const string EwsPassword = "TODO";
```

Once the connection information has been updated, the tests can be debugged to exercise the various methods supported.

NOTE: these steps are required only when testing outside of the SmartConnector Runtime. When a Processor is running in the service, this is all managed for you.

5.4.1.3 EwsServerAdapterFixture (Mongoose.Process.Sample)

When SmartConnector hosts an EWS Serve endpoint, the data served is stored in SmartConnector's database. Interacting with the database layer is best achieved through the `EwsServerDataAdapter` class. This sample provides a quick jumpstart into it's use.

NOTE: Before executing the tests in this fixture, please review and address all issues enumerated in the class header comments. Special steps are required in order to run this Processor.

5.4.1.4 ProcessorValuesAccessProcessor (Mongoose.Process.Sample)

A Processor sub-class has access to the ProcessorValues data store. This example illustrates how to search and save values.

NOTE: Before executing the tests in this fixture, please review and address all issues enumerated in the class header comments. Special steps are required in order to run this Processor.

5.4.1.5 CustomEwsServeTestFixture (Mongoose.Ews.Server.Sample)

This sample illustrates how to introduce custom logic and handling into a standard SmartConnector EWS serve implementation. In this example both the interface (`INewDataExchange`) and implementation of one of the base methods (`MyCustomGetValuesProcessor`) are overridden.

While this example is not actually hosted by SmartConnector (so it can be debugged easily), it does demonstrate that even though the logic has been modified, the standard SmartConnector `EwsClient` library has no issues consuming the known interface. If you wish to call interface extensions, you need to create a new proxy that includes additional methods and other constructs.

6 Writing Your Code

6.1 Solution Structure

While using the examples described is one way to begin authoring extensions, combining Processors and Unit Test Fixtures in the same class assembly project is not recommended for actual Extension development. It is recommended that the class assembly project you will deploy contains only the necessary classes and references. Listed below are the steps you need to perform when starting a new project.

6.2 Processors

1. Create a Visual Studio Class Library project using your preferred development language.
2. Create a reference to the Mongoose.Process NuGet package. Other SmartConnector NuGet packages can be added as needed.
3. Add a new class. Consistent naming convention is encouraged; Processor name should end with the word "Processor" eg "`MyCustomProcessor`".
4. Have the Processor sub-class the `Mongoose.Process.Processor` class.
5. Override the `Execute_Subclass` method. In this method the logic intended to be executed should be added when the processor runs.
6. If you wish to not require a license in order to execute this Processor, override the `IsLicensed` property and return false. See Licensing for more information about the specific features of Extension licenses.
7. Add properties needed to execute the custom logic. These properties should be public and have public getters and setters. Standard .NET validation attributes can be used to decorate these properties. See Microsoft Developer Network (http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) for details. By default all public read/write properties are available for configuration. If a property is not needed for configuration purposes, you can add the `ConfigurationIgnore` attribute to indicate this.

6.2.1 Execute_Subclass

The main entry point for the custom logic resides in this method. The return type is a list of `Prompt` instances that can be used to indicate any messages to be conveyed to the Worker as part of the results. Presently, these are only used for logging purposes. If no messaging is to be conveyed, then an empty list should be returned.

Loops in your Processor code are expected. Well written Processor code must still be responsive to external input from users, the operating system, or the Worker Manager in the way of a "stop" request. To achieve this, the base class has a `CancellationToken` property that is managed by the SmartConnector Runtime. Processor authors must monitor this if loop constructs of any type are present – even if they are "short" loops. This can be done in two different ways:

6.2.1.1 CheckCancellationToken()

Calling `CheckCancellationToken()` at either the top or bottom of any loop is one way to honor a stop request. This approach should only be used if an uncontrolled and immediate termination of the method is acceptable as it will cause a `TaskCancelledException` to be thrown if a stop request is pending. SmartConnector will call the virtual `CleanupBeforeCancellation` method prior to throwing the exception if cleanup of any type is required.

6.2.1.2 IsCancellationRequested

If a controlled exit is preferred, the `IsCancellationRequested` property can be used to determine if a stop is pending. The method should then terminate as soon as possible after performing any required actions.

6.2.2 Validation

Because SmartConnector is highly configurable, validation of the state of the Processor – loaded from a Processor Configuration – is critical. Processor validation is based on the .NET Validator class and does both schematic (attribute based) and semantic (interface based) validation. SmartConnector enhances the .NET experience by performing a deep graph traversal and validating the entire Processor instance. Any ITraversable or IEnumerable will be traversed during validation. Since Processor implements `ITraverseable`, the validation of all of the Processor's child properties is guaranteed.

6.2.2.1 Schematic Validation

Schematic validation is based on attributes which will be used to indicate what properties are required, ranges of acceptable values, and other declarative methods. Any ValidationAttribute subclass can be used whether it is native to .NET or authored by outside sources to add custom attribute-based validation.

The SmartConnector framework provides the following ValidationAttribute sub-classes to aid in validation.

6.2.2.1.1 CollectionLength

This attribute specifies the minimum and/or maximum number of items that can be in any IEnumerable construct.

6.2.2.2 Semantic Validation

Semantic validation is based on the IValidatableObject interface which `Processor` implements. "Semantic" refers to the type of validation which is beyond the obvious schematic-based validation where context is required. For example, consider a Processor that has two properties but only one or the other is required; never both. Clearly, adding the Required attribute to both (or neither) would fail to accomplish the desired validation result. However, overriding the `Validate` method on `IValidatable` allows one to perform that type of contextual validation to ensure that one and only one value was supplied.

6.2.3 Other Interfaces

Other interfaces are available for use in the SmartConnector libraries. Some of these interfaces also provide functionality in the form of extension methods which are contained in the SmartConnector libraries while others are simply used to identify behavior.

6.2.3.1 ITraversable

Any class decorated as `ITraverseable` will allow graph traversal by the `ObjectExaminer` class for deep analysis. This is specifically used for Validation and to extract a Processor Configuration. If custom class constructs are added to a Processor which will require exposure via the configuration engine, you need to decorate those classes with this interface so that child properties will be exposed.

6.2.3.2 ILongRunningProcess

A Processer should perform its work as efficiently as possible and terminate. This will allow the finite number of Workers available to be reused as other Processors need to be executed. It is understood that sometimes a Processor can't be written in such a manner. For example, if the processor is communicating with a third party system by opening a socket and "listening" for traffic it may need to do so for an indefinite amount of time before exiting.

For cases like this, the interface `ILongRunningProcess` should be used to assure the SmartConnector Runtime that the Processor has not become unresponsive or stuck in an infinite loop. Failure to do so may cause the Worker Manager to terminate the Process because it has become unresponsive.

6.2.3.3 IEwsEndpoint

The `IEwsEndpoint` defines the properties needed to connect to a EWS Server. Extension methods exist to instantiate a `EwsClient` instance based on the credentials of an implementing class. SmartConnector's native EWS readers and writers use this interface.

6.2.3.4 IStaThreadedProcessor

If a Processor contains references to COM assemblies the Processor may be required to execute in an STA thread. To instruct the Worker Manager to handle threading using STA threads, the author should include the sub-class directive for this interface.

6.2.4 Other Attributes

Other attributes are available for use in the SmartConnector libraries.

6.2.4.1 ConfigurationIgnore

Processor properties with this attribute will be ignored by SmartConnector for the purposes of configuration and will not be displayed to the user in the Portal.

6.2.4.2 DefaultValue

The DefaultValueAttribute is part of the .NET framework (System.ComponentModel). It is enumerated here since SmartConnector supports its use during Processor Configuration creation if a public property of the Processor class is decorated with it.

6.2.4.3 RandomStringDefaultValue

SmartConnector includes a sub-class of the System.ComponentModel.DefaultValueAttribute which will generate a random string value rather than one the extension author defines.

6.2.4.4 ProcessConfigurationDefaults

When attributed to a Mongoose.Process.Processor sub-class, newly created ProcessConfigurations will automatically populate the Name and Description of the ProcessConfiguration with the supplied values.

6.2.4.5 Tooltip

A Processor author can decorate any configurable property with the Tooltip attribute. The contents of the "tip" will be rendered in the SmartConnector Portal when the user clicks on the icon. This can be useful to provide context based instructions and guidance to Portal users.

6.3 EWS Servers

One way to begin authoring a unique EWS Serve implementation is to use the Mongoose.Ews.Server.Sample project as a starting point. Listed below are the steps you need to perform when starting a new project.

1. Create a Visual Studio Class Library project using any language preferred.
2. Create a reference to the *Ews.Server.Contract* and *Mongoose.Ews.Server* NuGet packages. Other SmartConnector NuGet packages can be added as needed.
3. Add a service class which sub-classes `MongooseDataExchange`.
4. If you want to modify the default behavior for one of the implementation classes:
    a. Add a class for each method you want to override. For example, in the sample NuGet, `MyCustomGetValuesProcessor` sub-classing `MongooseGetValuesProcessor` was added.
    b. For each class added in step a, override the create processor method in the service class added in step 2 and return your new class.
5. If you wish to extend the interface to add new methods and/or data structures:
    a. Add a new interface which sub-classes `IDataExchange`.
    b. Add your methods to this interface.
    c. Modify the service class created in step 3 to also sub-class this interface.
    d. Implement any methods in the service class.

6.4 Licensing

SmartConnector supports licensing of Processor and EWS Server Extension Assemblies. License files are created and managed in your Tenant on www.smartconnectorserver.com. Once generated, a license is immutable. Additionally, once a license is added to the SmartConnector runtime, the data cannot be altered even though it is stored in the internal database. Any edits to either the physical file or the database license fields will render the license unusable.

Licenses can be created with the following restrictions. Multiple restrictions will behave in a logical "OR" fashion:

- Time based licensing. License will expire at an absolute date in the future.
- Machine based licensing. Licenses can be generated that allow code to run on only a specific machine.
- Version based licensing. Licenses can be generated that will only allow specific versions of the extension assembly to execute.
- Custom defined features. Licenses can be generated with custom features that are enforced at run time from the Extension itself. See Custom Licensing for more details.

6.4.1 Licensing your Extension

To license your extension, you only need to annotate your `AssemblyInfo` file with the `PublicKey` attribute with the corresponding "Public Key" value from the www.smartconnectorserver.com extension you created for your assembly as shown in Figure 6. See https://www.smartconnectorserver.com/Faq for instructions on how to create an "Extension" to represent your Extension Assembly.

```
[assembly: AssemblyTitle("Mongoose.Process")]
[assembly: AssemblyDescription("Core Mongoose framework library.")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Schneider Electric")]
[assembly: AssemblyProduct("Mongoose.Process")]
[assembly: AssemblyCopyright("Copyright © Schneider Electric 2013-2015")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: PublicKey("YOUR PUBLIC KEY GOES HERE")]
```

Figure 6: Extension Public Key Annotation

6.4.2 Opting Out

By default, all Processor sub-classes in any Extension assembly will require a license in order to execute. Additionally, all EwsServiceHost sub-classes in any Extension assembly will require a license. In order to "opt out" of license enforcement for any Processor or EwsServiceHost, you must override `IsLicensed` in your and return false. Processor Extension assemblies can contain a mix of licensed and unlicensed Processors. If no Processor classes in your assembly are `IsLicensed` then the `PublicKey` attribute described above is not required. The same is true if the EwsServiceHost sub-class is not to be licensed.

6.4.3 Custom Licensing

Custom license features can be used to provide highly granular control of specific aspects of the Extension itself. Custom license features are represented as a Dictionary of string values. For the majority of scenarios, this should suffice. However, if more complicated design scenarios are required, you can store a serialized object in the value and deserialize it at runtime.

Enforcement of the custom licensing features is the responsibility of the Extension author. This enforcement is accomplished by overriding the `Processor`.

`ValidateCustomLicenseFeatures` method and returning the appropriate `Prompt` instance in the response. An example for the `LicensedNullProcessor` is shown below. The custom license feature in this example is called "MaxSleep" and represents the maximum allowed sleep value which the `Processor` can be configured for. By deferring this validation from design time (using attribute validation) to runtime, different customers can operate the same `Processor` under different constraints.

```
protected override IEnumerable<Prompt> ValidateCustomLicenseFeatures (ExtensionLicense license)
{
  var baseResults = new List<Prompt>();
  baseResults.AddRange(base.ValidateLicense(license));

  var maxSleep = license.Features.FirstOrDefault(x => x.Key == "MaxSleep");
  if (!string.IsNullOrEmpty(maxSleep.Value))
  {
    int asSleep;
    if (int.TryParse(maxSleep.Value, out asSleep))
    {
      if (SleepDuration > asSleep)
      {
        baseResults.Add(new Prompt
        {
          Message = string.Format("You are not licensed to use a SleepDuration value greater than {0}",
                      maxSleep.Value),
          Severity = PromptSeverity.MayNotContinue
        });
      }
    }
  }
  return baseResults;
}
```

7 Annotating Your Assemblies

The Portal displays assembly information for a Configuration or EWS Server to the user on both the Configuration and EWS Server pages. Additionally, this information is displayed during step one of the Add Configuration and Add EWS Server workflows. The following fields are displayed:

- **Class Name** – The class name of the Processor or ServiceHost sub-class.
- **Assembly File Name** – The file name of the extension assembly.
- **Assembly Description** – Value of the AssemblyInfo.AssemblyDescription for the assembly.
- **Assembly Company** – Value of the AssemblyInfo.AssemblyCompany for the assembly.
- **Assembly Copyright** – Value of the AssemblyInfo. AssemblyCopyright for the assembly.
- **Assembly Version** – Value of the AssemblyInfo.AssemblyCompany for the assembly.

As an extension author, you are encouraged to enter information specific to your business unit. The information displayed here is invaluable for users which may be troubleshooting the extension.

8 Testing Your Code

A test-driven development approach is strongly encouraged when developing extension libraries for SmartConnector. The samples illustrate a minimal approach to doing this. The developer should decide the exact amount of testing that needs to be performed prior to solution development. A NuGet package is provided to you to help facilitate test creation. Follow the instructions outlined in the Sample NuGet Packages above and chose the *Mongoose.Process.Test* package to download useful extension methods for your tests.

9 Deploying Your Code

As you are developing in .NET, deploying the class library is very straightforward. Simply copy the binary output preferably generated from a Release configuration, into the same folder location that the SmartConnector Service was installed to. If the class library has external references that are not part of the SmartConnector core, then those dependencies should also be copied.

SmartConnector native libraries that were installed into the class library via NuGet should NOT be copied.

9.1 Strong-Named Assemblies

Beginning with the 2.0 release, all SmartConnector binaries are Strong-Named Assemblies. As a result, all Extension assemblies written prior to 2.0 must be re-compiled against SmartConnector 2.0 or later. Otherwise an exception will be thrown when the SmartConnector framework attempts to load an assembly.

This requirement is true regardless of whether or not you strongly name your Extension assembly or whether or not you incorporate licensing requirements into your Extension assembly.

However, once an extension has been re-compiled against SmartConnector 2.0 or later, it is likely that it will not have to be re-compiled against future SmartConnector versions so long as they are backward compatible.

See the Troubleshooting section in the SmartConnector Installation and Configuration Guide for more details.

9.2 Updating Extension Assemblies

Updating Extension assemblies after initial deployment is performed in the same manner as initial deployment. If the Extension being updated was being used, the .NET framework will put a file lock on the DLL which will prevent it from being overwritten. While workarounds are being considered, presently you will need to stop the SmartConnector Server prior to updating your Extension assemblies

10 Appendix

10.1 OData API

Open Data Protocol (OData) is a data access protocol initially defined by Microsoft (through version 3.0) but later standardized by OASIS (version 4.0 and later). SmartConnector currently conforms to the OData v3.0 specification.

You will need to be familiar with the specification in order to consume the API. Please consult the OData web site for information regarding the specification. http://www.odata.org/

10.1.1 Authentication

SmartConnector requires authentication for the all interactions with the API. SmartConnector authentication is done via OAuth2 against a local user store. Future versions will include Active Directory and third party identity support. Refer to the "SmartConnector Installation and Configuration Guide" for information about the default credentials installed and user management.

Before calling an authenticated endpoint, API clients will need to obtain a Bearer Token by issuing an HTTP POST to the Token endpoint and supplying the credentials as shown in Figure 7.

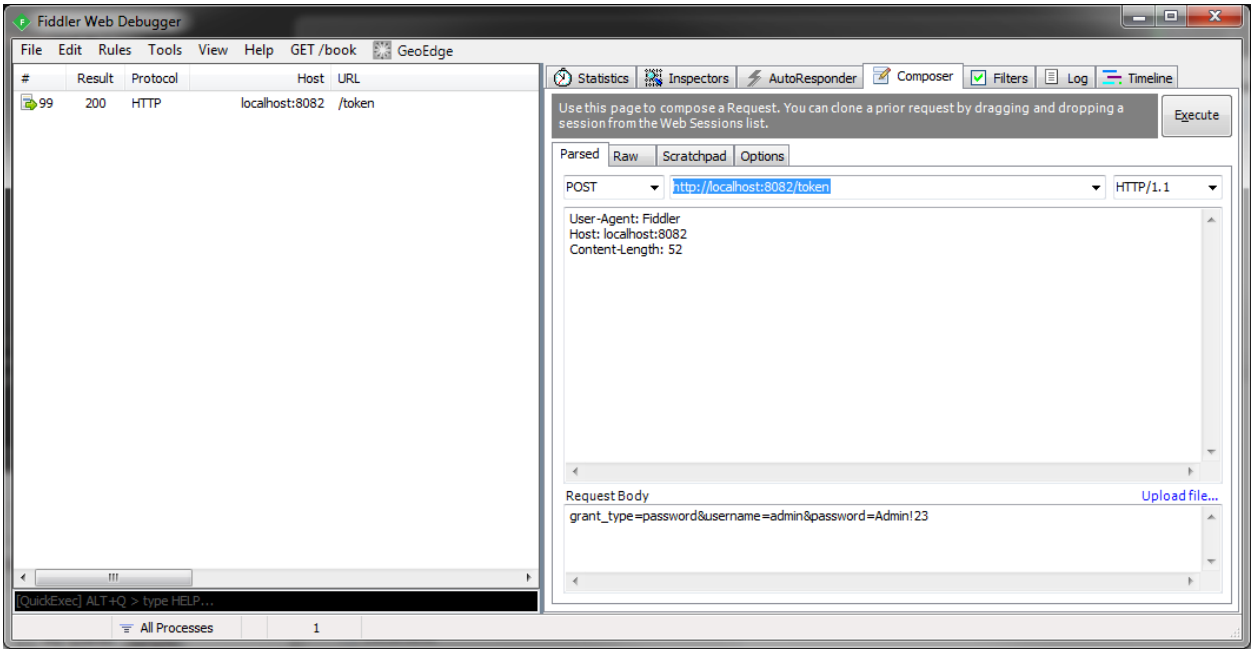

Figure 7: Authentication Bearer Token Request

If you do not wish to construct your own HTTP request, SmartConnector provides the `BearerToken` static helper class in the Mongoose.Common.Security namespace to facilitate retrieval of a Bearer Token.

NOTE: The Bearer Token request occurs in plain text if the SmartConnector Service is not configured for HTTPS. Consult the Security Considerations section of the SmartConnector Installation and Configuration Guide.

Once a Bearer Token has been obtained, subsequent requests need to include an "Authorization" header with the value of "Bearer BEARER_TOKEN_VALUE".

Bearer tokens are useable for 15 minutes from issuance. After that time a new bearer token is required.

10.1.2 What you can do with the API

Typical use for the API is for CRUD operations. Not all controllers support full CRUD operations. The table below enumerates the available controllers and what CRUD actions are supported by each controller. When working with the API, please be advised of the security considerations section listed in the SmartConnector Installation and Configuration Guide.

10.1.2.1 CRUD Operations

Controller Name	Description	C	R	U	D
EwsServerRequests	Requests to start or stop an EWS Server		X		
EwsServers	Logical EWS Servers		X		X
LogFilters	Entries to limit the amount of logging perfomed	X	X	X	X
Parameters	Configuration Parameters of either a ParameterSet or ProcessConfiguration.	X	X	X	X
ParameterSets	Configuration ParameterSets of either a ParameterSet or ProcessConfiguration.	X	X	X	X
ProcessConfigurations	Configuration to run a Processor with an optional Schedule with all of the defining properties to instantiate and hydrate the class that will be executed.	X	X	X	X
ProcessorValues	Persistent data store for Processors to store state data.	X	X	X	X
ProcessRequests	Requests to start or stop a Process Configuration.		X		X
Schedules	Schedule definitions for how often a Process Configuration will run.	X	X	X	X
Settings	Service settings.		X	X	
ThreadStates	Status information on what the current SmartConnector Service is working on.		X		

10.1.2.2 Custom Actions

In addition to the standard CRUD actions defined above some controllers have custom actions. The following enumerates those custom actions and the parameters (if any) required.

Only those controllers listed have custom actions.

Controller Name	Action	HTTP Verb	Parameters
EwsServerRequests	Purge	POST	n/a

EwsServers	Start	POST	id: Id of the EWS Server to start.
	Stop	POST	id: Id of the EWS Server to stop.
	CreateNew	POST	name: name of the server address: complete url of the endpoint. realm: realm for digest authentication username: username to authenticate with password: password to authenticate with isAutoStart: Server will auto start at service start assemblyFile: Name of host class assembly className: Name of the host class in assemblyFile
ParametersController	Decrypt	POST	id: Id of the Parameter to decrypt.
ParameterSetController	CreateCollectionItem	POST	id: Id of the ParameterSet to create a new child for.
ProcessRequests	Purge	POST	Deletes all non-pending requests.
ProcessConfigurations	Start	POST	id: Id of the ProcessConfiguration to start
	Stop	POST	id: Id of the ProcessConfiguration to stop
	Clone	POST	id: Id of the ProcessConfiguration to clone
	Schedule	POST	id: Id of the ProcessConfiguration to schedule. scheduleId: ID of the Schedule to assign to the ProcessConfiguration.
	Unschedule	POST	id: Id of the ProcessConfiguration to unscheduled
	CreateNew	POST	name: Name of the Processor. description: Description of the Processor. assemblyFile: Name of the Processor class assembly className: Nave of the Processor class.
ProcessRequests	Purge	POST	Deletes all non-pending requests.

10.1.3 Other Controllers

In addition to the OData compliant controllers mentioned above, the following conventional Web API controllers are also available.

Controller Name	Route	HTTP Verb	Purpose
Ping	Get	GET	An unauthenticated endpoint to confirm that the service is responding.
	Error	GET	Returns an error to test client handling of the types of runtime exceptions which SmartConnector may return.
Info	Get	GET	Returns runtime information about the version of the SmartConnector Service. Requires authentication.

10.2 Third Party Tools

In general, any programming language that can issue HTTP requests can be used to consume the API. However, for simple testing purposes there are some free applications available that make the job much easier.

10.2.1 Fiddler

Probably the best freeware tool available to debug and manipulate HTTP requests is Fiddler (http://www.telerik.com/fiddler). The web is full of tutorials, tips, and tricks on how to use Fiddler so we will not duplicate that content here.

10.2.2 LINQPad

LINQPad (http://www.linqpad.net) is a freeware application (premium version also available) that makes writing Language Integrated Queries (LINQ) easy. One of the features of LINQPad is the ability to consume WCF Data Services which makes consuming the API quick and easy for querying purposes.

To connect LINQPad to SmartConnector perform the following.

1. Open LINQPad.
2. Click Add connection.
3. In the dialog choose WCF Data Services 5.5 (OData3)
4. Click Next
5. Enter the endpoint of the SmartConnector OData API.

At this point, the connection dialog will look like Figure 8.

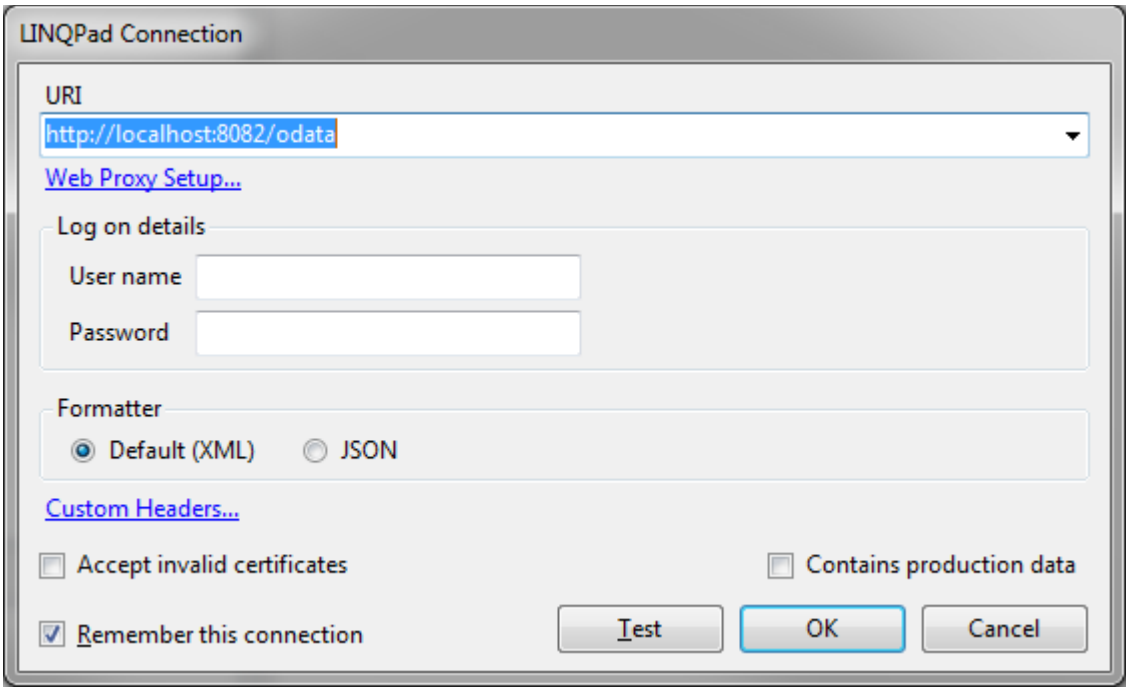

Figure 8: LINQPad Connection Properties

LINQPad doesn't support OAuth handshaking so we are not able to use the "User name" and "Password" inputs. In order to use LINQPad with SmartConnector, we will need to manually add a custom header with a valid bearer token (see Authentication).

6. Click Custom Headers.
7. In the Name column enter "Authorization".
8. In the Value column enter "Bearer VALUE_OF_THE_BEARER_TOKEN".
9. Click OK.
10. Click OK.

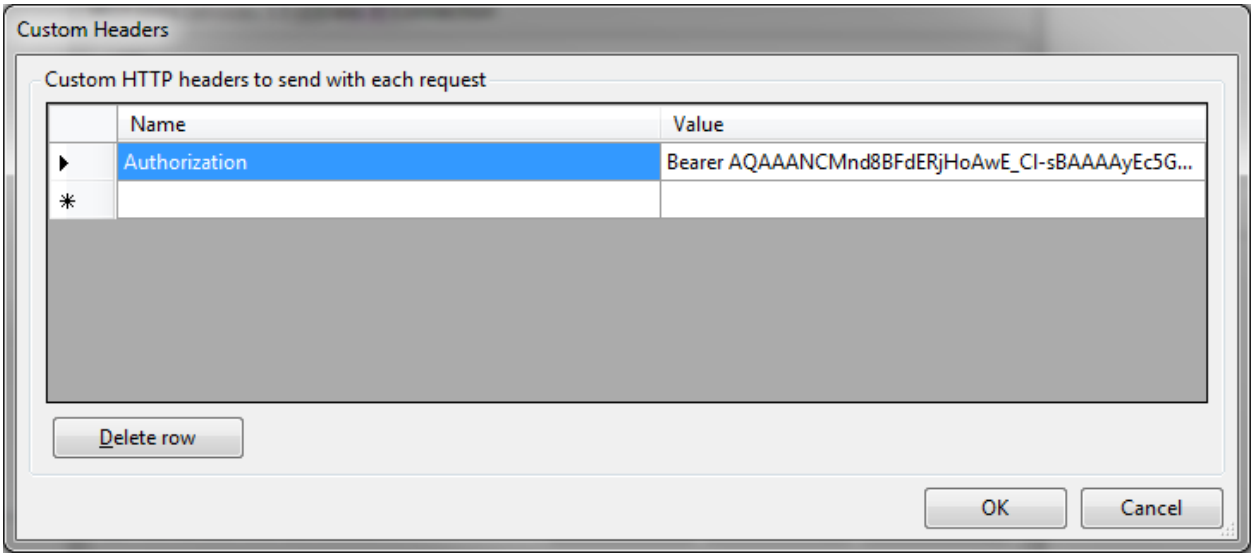

Figure 9: LINQPad Custom Headers

You are now ready to queries against SmartConnector's OData API.

10.3 Supported Operating Systems

The following is a list of Operating Systems which SmartConnector has been tested against. Non-listed operating systems capable of running .NET 4.5 should also work but their compatibility has not been verified.

- Windows 7 64 bit.
- Windows 10 64 bit.
- Windows Server 2008 64 bit.
- Windows Server 2012 64 bit.

10.4 Supported Database Servers

The following is a list of Microsoft SQL Servers which SmartConnector has been tested against. Non-listed servers compliant with Microsoft SQL Server may also work but their compatibility is not guaranteed.

- Local DB.
- Microsoft SQL Server 2012 Express.
- Microsoft SQL Server 2012.
- Microsoft SQL Server 2014 Express.
- Microsoft SQL Server 2014.