2016 OSIsoft TechCon

Advanced Programming with PI AF SDK

OSIsoft, LLC
777 Davis St., Suite 250
San Leandro, CA 94577 USA
Tel: (01) 510-297-5800
Web: http://www.osisoft.com

Published: March 21, 2016

Table of Contents

Introduction

The PI AF SDK is a .NET-based programmatic library that provides access to the variety of data stored in the PI System including:

- Assets and their properties and relationships
- Time series data from the PI Data Archive and other sources
- Event frames that record and contextualize process events

Note: PI AF SDK is not limited to only interacting with the AF Server. It provides rich, powerful methods to interact directly with the PI Data Archive and is almost always more appropriate to use than the older PI SDK (to be deprecated).

This course will cover immediate and advanced usage of the PI AF SDK. A background in .NET application development and a familiarity with the PI System and PI AF SDK are assumed.

The feature set of PI AF SDK is well-described in the PI AF SDK Online Reference. However, less apparent are the workings under the hood. This course aims to elucidate some of the underlying behavior so client applications can be written with higher performance and optimal resource usage.

Getting Started

1. Open the Visual Studio Project

Open the file:

Desktop>Advanced Programming with PI AF SDK> Exercises-Advanced-Programming-with-PI-AF-SDK>
`Advanced-Programming-With-PI-AF-SDK-TechCon2016.sln`

Each exercise is contained in a Console project. An accompanying "solution" project (project name appended with "-Sln") exists for each exercise.

You are not required to do all the exercises. **Rather, choose which exercises are relevant to your interests**.

Each exercise consists of the following sections:

1. **Introduction** – Discuss the features and methods used in the exercise.
2. **Problem** – Describes the set of objectives to achieve.
3. **Hints** – Gives some hints to help complete the problem.
4. **Bonus** (optional) – Gives additional challenges and corner cases.

Look at the answers if you get stuck. This is not an exam. ☺

2. Understand the AF Database structure

All exercises are based on the AF Database called **Feeder Voltage Monitoring**. This database contains a list of feeders and transformers that have been deployed to various sites

Please familiarize yourself with this AF Database using PI System Explorer before starting the exercises.

3. Choose your own adventure

Pick an exercise that you find interesting. A list with approximate difficulty level (4 = easier) is below:

Exercise	Difficulty
Ex1-Finding-And-Loading-Assets	5
Ex2-Measuring-AF-Server-RPCs	5
Ex3-Using-Typed-AF-Value	4
Ex4-Asynchronous-Data-Access	6
Ex5-Real-Time-Analytics	6

Follow the instructions in the workbook for the Exercise(s) you choose.

Exercise 1: Finding and Loading Assets

Introduction

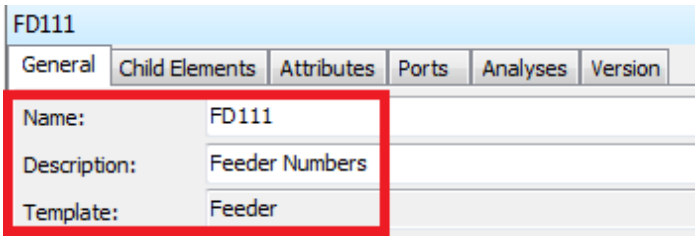
PI AF SDK uses lazy loading when returning objects to the caller. Lazy loading means that PI AF SDK methods will usually retrieve a partial amount of information associated with each object. Additional requests for object information is retrieved on-demand and can require another server round-trip.

This can be advantageous for UI applications which typically respond to ad-hoc queries. Since the next query is unknown, it is serviced on-demand. This pattern can be undesirable, however, in applications in which the queries are predictable. In these cases, lazy loading can cause many unnecessary network roundtrips, incurring high latency cost. It then becomes more desirable to make bulk requests that minimize the number of roundtrips.

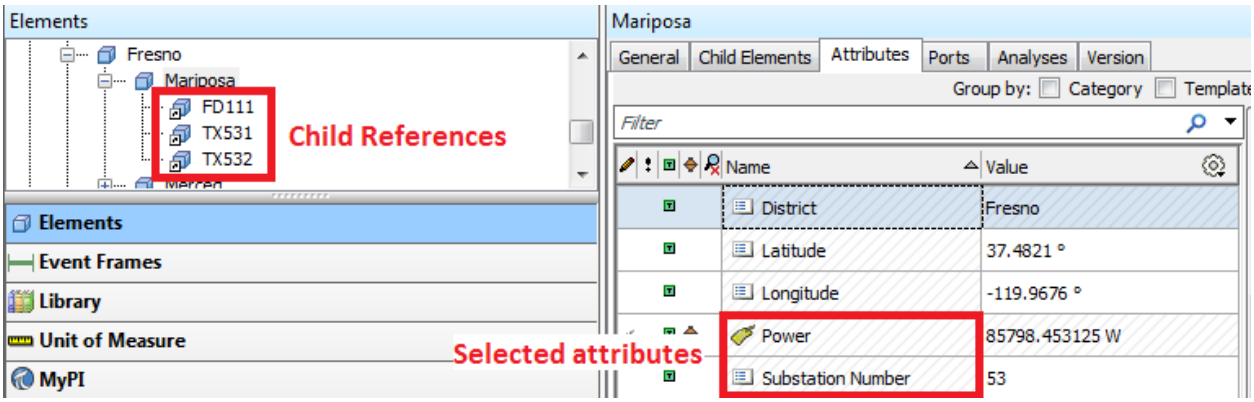
PI AF SDK offers options for specifying the degree of "bulking" when querying for `AFElement` objects. These methods will return `AFElement` objects loaded in different states. Here, "state" refers to the extent to which the `AFElement` object is loaded with information from the server.

An `AFElement` can be in one of three states:

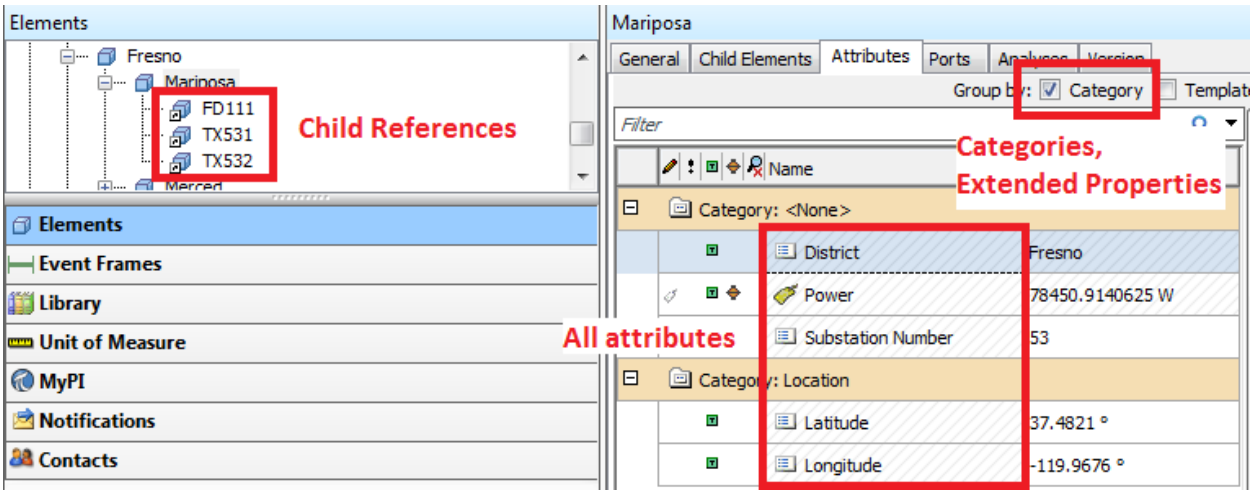
Header only: The element header consists primarily of the Name, Description, and Template.



Partially loaded: The partially loaded state consists of the header, child references, and selected attributes.



Fully loaded: The fully loaded state consists of the header, child references, all attributes, categories, and extended properties.



A few common PI AF SDK calls that generate the corresponding `AFElement` states are outlined below:

Header access:

1) Static "FindElement" methods:

   `AFElement.FindElement()`, `AFElement.FindElements()`, `AFElement.FindElementsBy()*`

   Typically, any methods starting with "Find" will only load the headers. These methods return a collection of headers represented via `AFNamedCollectionList<AFElement>`.

2) Elements collection:

   `AFElement.Elements`, `AFDatabase.Elements`

   Accessing a child elements collection via the `Elements` property on an `AFElement` or `AFDatabase` returns a paged collection of element headers.

3) `AFElementSearch` using the default `fullLoad: false`

   `AFElementSearch.FindElements()`

   This method is new in AF 2.8. It will return an enumerable collection of element headers.

Partial load:

1) `AFElement.LoadAttributes()`

   This method will ensure that each `AFElement` specified in the elements list are at least partially loaded with the specified attributes. This method is recommended if you know in advance the attributes to use because it reduces the number of server roundtrips and memory usage.

Full load:

1) `element.Attributes[attrName]`, `element.Attributes.Count`

   If the element is only header-loaded, accessing an attribute via the `Attributes` collection will fully load the element.

2) `AFElement.LoadElements()`, `AFElement.LoadElementsToDepth()`

   These methods return fully loaded elements within the `AFNamedCollectionList<AFElement>`.

3) `AFElementSearch` using the `fullLoad: true`

   `AFElementSearch.FindElements(fullLoad: true)`

   This method is new in AF 2.8. It will return an enumerable collection of fully loaded elements.

Partial loading is particularly advantageous for optimizing memory usage of an application. Only the AF Attributes used by the application are loaded from the server, instead of all attributes of an element.

Problem

Imaginary Power Company tracks their feeders in an AF Database called **Feeder Voltage Monitoring**.

They would like to decrease memory consumption of `AFElement` objects that represent feeders by taking advantage of **partial loading** of elements (i.e. load only the attributes needed for the application).

Your task is to write a method that returns a list of `AFElement` objects that are partially loaded. The method accepts an `AFElementTemplate` and list of attribute names as input. As output, a list of `AFElement` objects is returned partially loaded with the attributes requested. The method signature is below.

```
IList<AFElement> LoadElements(AFElementTemplate elementTemplate, IEnumerable<string>
attributesToLoad)
```

Please implement this method in `AFElementLoader.cs`.

To test this application, please use the driver application in `Program1.cs`. It will return all elements belonging to the **Feeder** template and loads the **Reactive Power** and **Total Current** attributes.

Your challenge is to keep the reported memory usage **below 260 KB**. `Program1.cs` contains code for measuring the memory usage.

Hints

Use `FindInstantiatedElements()` on the instance of an `AFElementTemplate` to find all base elements instantiated from the template. This method will only load the element headers.

`FindInstantiatedElements()` only returns a fixed number of elements each call. Use a paging pattern to make sure you have found all the elements.

Bonus

How would you also find elements instantiated from an element template that derives from the method's input element template?

The input attribute names may actually belong to a base template of the passed in element template. In this case, the `AFAttributeTemplate` must be instantiated from the base template. How would you do this?

Exercise 2: Measuring AF Client Performance

Introduction

This exercise demonstrates how to diagnose where bottlenecks might be occurring in a PI AF SDK application.

In order to diagnose performance issues, it is important to first understand the steps performed during a call to the AF Server.

1) PI AF SDK converts AF objects into data transfer objects.
2) The data transfer objects are serialized and transported to the AF Server.
3) AF Server receives the request and builds the call(s) to SQL.
4) SQL Server executes the query and returns the result set.
5) AF Server converts the result set into data transfer objects.
6) The data transfer objects are serialized and transported back to the client.
7) PI AF SDK constructs AF objects and returns them to the caller.

RPC Metrics in PI AF SDK

PI AF SDK provides two methods to measure the RPC (remote procedure call) durations:

`PISystem.GetRpcMetrics(), PISystem.GetClientRpcMetrics()`

These methods can help not only to identify client and server-side bottlenecks, but can also reveal unexpected calls being made to the AF Server.

Server-side RPC measurements are returned by `GetRpcMetrics()`. They include steps 3-5 above for each RPC.

Client-side RPC measurements are returned by `GetClientMetrics()`. They include steps 2-6 above for each RPC.

To measure the times for steps 1 and 7, a .NET `Stopwatch` class can be used to measure the total time, and then the client RPC times can then be subtracted from the total.

AFProbe class

In the `External` project, we have created a class called `AFProbe` that you can use to measure durations of PI AF SDK methods. Its usage is very simple.

```
using (new AFProbe("Name of your measurement", piSystem))
{
      // make your PI AF SDK call here
}
```

The class will output duration information to the Console upon leaving the `using` scope. You can also nest calls by instantiating one probe inside another's scope.

Here is a sample output from a `FindElementsByAttribute()` call using an `AFAttributeValueQuery`. We've highlighted interesting aspects below.

```
Operation FindElementsByAttribute took: 147 ms

RPC Metrics

      GetElementTemplate: 1 calls.  21 ms/call on client. 3 ms/call on
      server.  Delta: 18 ms/call

      GetCategory: 5 calls.  3 ms/call on client. 1 ms/call on server.
      Delta: 2 ms/call

      FindElementsByAttributeValue: 1 calls.  52 ms/call on client. 31
      ms/call on server.  Delta: 20 ms/call

Total RPCs: 7
```

Problem

Explore the usage of the `AFProbe` class for measuring durations of PI AF SDK calls to the AF Server. Choose any methods you'd like and examine the output and timings.

For the AF Database called **Feeder Voltage Monitoring**, print to the Console the **name** and **attribute count** for all feeder and transformer elements. Do this with 10 or fewer AF Server RPCs. Use the `AFProbe` class to measure durations and number of RPCs. Note that requesting the attribute count for an element will **fully load** the element if it is not already loaded.

Hints

In order to keep the RPC count below 10, fully load all feeder and transformer elements before requesting the attribute counts.

Bonus

Use the new `AFElementSearch` object in PI AF 2.8 to fully load the feeder and transformer elements.

If you run the application twice in a row, is the second query faster? Why?

Exercise 3: Using Typed AFValue

Introduction

This exercise introduces "typed" `AFValue` features introduced in PI AF SDK 2.7.5 and 2.8 that reduce the overhead of boxing/unboxing when working with `AFValue` objects.

PI AF 2015 R2 introduced a mechanism to retrieve a value from an `AFValue` object without the unboxing overhead of converting a .NET `object` to a primitive type. This enhancement is exposed by the following new properties and methods.

- AFValue.ValueType property
- AFValue.ValueTypeCode property
- AFValue.ValueAsDouble() method
- AFValue.ValueAsSingle() method
- AFValue.ValueAsInt32() method

For numeric types, the `ValueAs*()` methods above will perform the appropriate casting if necessary. `ValueAsInt32()` can throw an overflow error. In most cases, `ValueAsDouble()` is appropriate for numeric conversions.

PI AF 2016 introduced a new factory method `AFValue.Create()` that allows the creation of an `AFValue` without the boxing overhead.

Problem

We would like a sort a list of `AFValue` objects by their underlying primitive values. Although `AFValue` objects have a `CompareTo` method, this method compares the timestamps, rather than values. Therefore, we will write our own class that performs the comparison by value.

Using the methods above for typed `AFValue`, create a class called `AFValueComparer` that implements the `IComparer<T>` interface. This class is used to compare two `AFValue` objects by their underlying primitive values, as returned by the `ValueAs*()` methods above. The `IComparer<T>.Compare` method should check if the `AFValue` objects have an accepted `TypeCode` (`Double`, `Single`, `Int32`) and should call the `CompareTo` methods exposed by the primitive types (e.g. `double.CompareTo()`).

The class should have the following skeleton:

```
public class AFValueComparer : IComparer<AFValue>
{
	public int Compare(AFValue val1, AFValue val2){ // your code here }
}
```

A driver program exists in `Program3.cs` which uses your new class to sort a list of `AFValue` objects by their underling values, using the `List<T>.Sort(IComparer<T>)` overload where `T` is `AFValue`. It can be used to verify that your class sorts the values correctly.

Bonus

Change `Program3.cs` to use `AFValue.Create()` when instantiating new `AFValue` objects.

Exercise 4: Asynchronous Read and Write

Introduction

Asynchronous data methods provide a mechanism for concurrency during a data access call to the PI Data Archive. Concurrency here refers to two types of operations that can execute simultaneously:

1) The handling of the data request by the PI Data Archive, possibly over the network
2) Continued execution of client-side code that does not depend on the result of the request

PI AF SDK 2.8 introduces asynchronous data access methods based on the Task-based Async pattern introduced in .NET Framework 4.5. These methods are implemented in both the `OSIsoft.AF.Data` and `OSIsoft.AF.PI` namespaces. They are supported for AF Attributes configured with PI Point data references.

These asynchronous methods can be advantageous in many cases:

1) In front-end applications, the UI thread can stay responsive during a data access call.
2) In both client and server applications, the number of threads used to service a call can be reduced, as waiting threads can be returned to the thread pool for re-use
3) The effect of latency is mitigated because remote calls can be executed concurrently.

Problem

Imaginary Power Company has a front-end display that shows real-time summaries of power generation data for its feeders. The power generation is stored in the AF Attribute "**Power**". The summaries span the range of one day. The summary data includes minimum, maximum, average, and total power.

Your task is to write a method that retrieves the summary data for all Power attributes of Feeder elements using an asynchronous method. The method has the following signature:

```
async Task<IList<IDictionary<AFSummaryTypes, AFValue>>>
GetSummariesAsync(AFAttributeList attributeList)
```

Each item in the returned list is a dictionary that stores the per-attribute summary values (minimum, maximum, average, total). The dictionary is keyed by the `AFSummaryTypes` enumeration.

To implement this method, you should use the following asynchronous method for each attribute.

```
AFData.SummaryAsync()
```

The list of attributes has already been provided to you as `attributeList` which is passed into the method.

Bonus

Often, we want to limit the number of asynchronous calls that are in progress. The PI Data Archive places a limit on the number of client requests in progress, and one client executing many asynchronous

requests can easily exhaust server resources. How could you throttle the asynchronous method calls? See `GetSummariesAsyncThrottled()` in the answers for one approach.

Can we place a timeout on asynchronous method calls and if so, how? See `GetSummariesAsyncWithTimeout()` in the answers for one approach using a cancellation pattern. See the PI AF SDK 2.8 Help file (%pihome%\HELP\AFSDK.chm) for how to use a cancellation token with the new asynchronous methods.

Exercise 5: Real-Time Analytics

Introduction

PI AF SDK allows the client to sign up for and retrieve events that have arrived on the data source. Subscription allows the client to retrieve all of the arrived events, in contrast to snapshot polling, which may miss events.

This functionality is provided by the `AFDataPipe`. The `AFDataPipe` maintains a list of `AFAttribute` instances that represent the data source for the pipe. Clients can sign up for events produced by the `AFDataPipe`.

To receive events, the client must always poll the data pipe. This triggers a call to the data source to get the latest events. In this model, the server never calls the client.

The `AFDataPipe` is implemented as an Observer pattern. The `AFDataPipe` is an Observable, and the client should implement an Observer and subscribe this to the data pipe. To receive events, the client must explicitly call the `GetObserverEvents()` on the `AFDataPipe` instance. Within `GetObserverEvents()`, the `AFDataPipe` will call `IObserver.OnNext()` for each of the subscribed observers.

The `AFDataPipe` can use unmanaged resources and should be cleaned up after use by calling its `Dispose()` method. For PI Data Archive data sources, this also removes the server-side subscription.

In the `AFDataObserver.cs` (see VS Project called "External"), we present an example code pattern that uses `AFDataPipe` to process real-time events.

Problem

News feeds are implemented in a variety of social networks (Facebook, Linkedin, Quora, etc.). This is made possible by services that can return queries such as "Show me the top N items".

This feature has also been suggested for PI AF*. See [PI Square - Displaying Top N Elements].

Your goal is to write a class called `AssetRankProvider` that provides the top N **Feeder** elements sorted by their current **Reactive Power** values. Write code to do the following for the methods listed below.

1) `AssetRankProvider` constructor

   Initialize an internal data structure to store the current Reactive Power values keyed by Feeder element.

2) `AssetRankProvider.Start()`

   Subscribe the instance to the internal `AFDataPipe`. Signup the attributes to the data pipe. Start polling the data pipe for events.

3) `AssetRankProvider.OnNext(AFDataPipeEvent dpEvent)`

   Process the `AFDataPipeEvent` and add the returned `AFValue` to the internal data structure defined earlier.

4) `AssetRankProvider.GetTopNElements(int N)`

   Return an `IList<AFRankedValue>` that contains the top N Feeder elements sorted by their current Reactive Power values. The `AFRankedValue` is simply a container class that associates an `AFValue` with a ranking.

```
public class AFRankedValue
{
        public AFValue Value { get; set; }
        public int Ranking { get; set; }
}
```

5) `AssetRankProvider.Dispose()`

   Dispose of the `AFDataPipe` and other unmanaged resources.

In `Program5.cs`, a driver program has been prewritten to test the `AssetRankProvider`. Feel free to modify as necessary to suit your design.

* While PI Analysis Service rollups can calculate the maximum or minimum value from a collection of related element attributes, it does not currently return the corresponding element associated with the maximum and does not service “top N” queries.

Hints

An internal data structure is required to store the latest values of each attribute. A dictionary is a good choice.

You can sort a list in-place in C# using `IList.Sort()` and its various overloads.

Appendix 1: PI SDK Deprecation

PI AF SDK data access methods in the OSIsoft.AF.Data namespace and the OSIsoft.AF.PI namespace supersede the functionality provided by PI SDK. Though the PI SDK is still available, it is more appropriate to use PI AF SDK than the PI SDK, except in an architecture that demands only native code. For more information regarding functionality and migration, please consult the Overview section "PI SDK Equivalents" in the AF SDK online documentation.

Appendix 2: Optimization

Optimization Metrics

There is rarely a "free lunch" when optimizing – tuning an application by one metric will almost inevitably have a cost in another. Here we identify the metrics we will consider and briefly discuss their interplay.

Processor Usage

This is the amount of time the computer's CPU spends doing work. If the demand for computational resources exceeds the resources available, applications on the computer need to "take turns" which adversely impacts their performance.

Memory Usage

This is all the data an application is working with. If the memory used exceeds the physical memory available, much slower storage is used to hold the data. This adversely impacts all applications running on the computer.

Latency

This is the delay between the time something is requested and the time it is available.

Optimization Techniques

There are several common techniques that the AF SDK supports to achieve better performance. These techniques can be applied with great benefit where they are appropriate. However, they also make code more complex.

Lazy Loading

It is wasteful of processor cycles and memory to create and store objects that will never be used. By deferring the creation of objects until they are accessed, no work is done that has not been requested. AF SDK's object model makes heavy use of lazy loading.

Bulking

Loading items one-at-a-time can be inefficient when there is overhead per-call (e.g. if a network call is required). In this case there can be significant savings by making a 'bulk' request for several operations at once.

Chunking

When a bulk request becomes too large, the time required to process it might introduce latency of its own because all operations must be completed before further work is done. In this case it is advantageous to break the request into individual 'chunks' that can be processed as they complete. A general rule-of-thumb is that chunks of 1,000-10,000 are a good middle-ground.

Caching

By storing a copy of data locally, repeated network calls or calculations can be avoided. Caching can improve performance tremendously, but cache trimming (knowing when a value is no longer needed) and invalidation (knowing when the value needs to be re-queried or re-calculated) is often difficult.

Thus, caching should be reserved for cases with a clear benefit and a plan for trimming/invalidation should be designed from the start.

So a general optimization strategy is to:

- Determine what your performance goals are (e.g. analysis should run in under 10 min but takes 15 min).
- If your program does not meet your performance goals, identify the primary bottlenecks (e.g. data access calls take 8 min).
- Apply appropriate optimizations (e.g. break data access up and make calls in parallel).

Optimization Scenarios

The PI AF SDK is also used in a variety of applications. These applications each have different concerns that lead to different approaches to optimization.

Interactive Configuration/Exploration (e.g. PI System Explorer)

Interactive applications typically must run on computers with limited resources. Because the application is driven by a human, user input is typically the limiting step. Lazy loading is a good fit in this situation because resources usage is minimized while responsiveness is reasonable.

Scheduled (e.g. report generation)

For a program that is run periodically (perhaps daily), runtime is the most important factor. The correct balance between bulking/chunking leads to low overhead and good processor utilization.

Long Running Service (e.g. PI Analysis Service)

Long running services benefit greatly from caching since they might access the same data repeatedly. Thus it is critical to consume updates (both data and metadata) to keep cached information in sync with changes being made. Long running services typically run on server class machines so higher processor and memory usage is acceptable in exchange for throughput improvement.

Multi-user service (e.g. PI Web API)

Multi-user services are concerned with the volume of requests that can be handled while staying responsive. Caching is productive if it can be shared across users. However, any significant resource usage that scales with the number of users is problematic.

Appendix 3: Set up Visual Studio for PI AF SDK in a new project

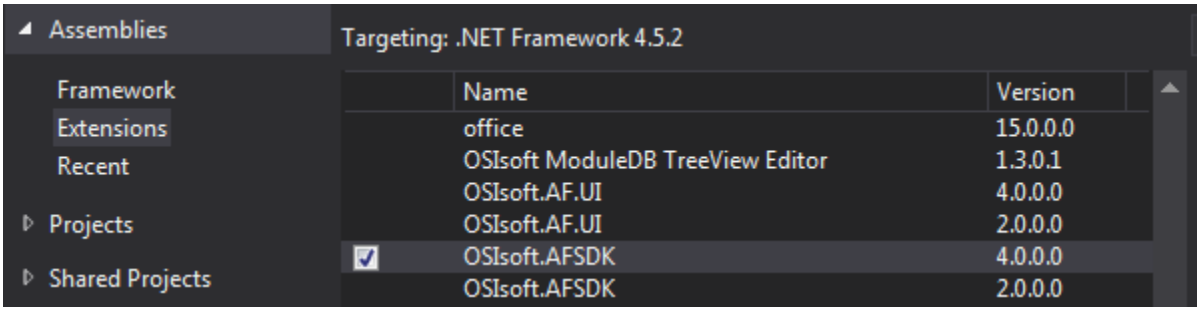
Adding the PI AF SDK Assembly Reference

The Visual Studio Solution for this lab already has a reference to the PI AF SDK added.

However, if you are creating a new Visual Studio Project, you can add a reference to OSIsoft.AF.SDK.dll (4.0.0.0 assembly version) via the following:

1) From the Visual Studio toolbar, click Project>Add Reference.

2) Under Assemblies in the left-pane, choose Extensions, and then select OSIsoft.AFSDK with Version 4.0.0.0.



Note: The 4.0.0.0 assembly version is only available when targeting at least .NET Framework 4.x. For exact version requirements, consult the product release notes.

Importing Namespaces

Namespaces can be brought into scope by adding the appropriate `using` directives at the top of the class file.

```
using OSIsoft.AF;

using OSIsoft.AF.Asset;

using OSIsoft.AF.Data;

...
```

Appendix 4: Further Resources

General References

PI Developers Club Forums

Online PI AF SDK Reference

PI AF SDK Guidelines

OSIsoft GitHub Sample Code

vCampus Live 2012 Presentation – PI AF SDK – Performance and Scalability

vCampus Live 2013 Presentation - Optimizing PI AF Server and PI AF SDK Applications

UC 2015 TechCon Hands-On Labs (Unzip and see "Working with the PI AF SDK")

AFDataPipe References

There is a wealth of information on using `AFDataPipe`, ranging from design, functionality, performance, scalability, and good practices. They are listed below in an approximate learning order.

vCampus Live 2013 Presentation - Optimizing PI AF Server and PI AF SDK Applications (Slides 20-25)

PI AF SDK Reference: AFDataPipe

PI Developers Club Webinar Series: AF SDK AMA 2015

PI DevClub: How to use the PIDataPipe and the AFDataPipe

PI AF SDK Guidelines

UC 2015 TechCon Hands-On Labs (Unzip and see "Working with the PI AF SDK – Exercise 4")

KB01195 – Why AFDataPipe requires polling when it uses IObservable pattern

PI DevClub: Using data pipes with future data in PI AF SDK 2.7

PI DevClub: Implementing the AFDataPipe in a Custom Data Reference

PI DevClub: Reactive Extensions for AF SDK (Part 1): LINQ to AFDataPipe

TechCon Class Exercises

The exercises can also be found online at

GitHub - Advanced-Programming-With-PI-AF-SDK-TechCon2016

Version Information

Version	Date	Description
1.0	March 14, 2016	Initial version