# Audience and Prerequisites

This document is targeted to developers who will be required to instrument business applications to measure business performance. This document illustrates how to use the Event Processing Framework to capture and send business data from a line of business application. A basic of WCF is recommended.

# Implications

Use of this framework ensures:

* common plumbing for all applications to reduce coding effort
* capture and staging of events in an efficient manner
* consistency in collecting data important for event correlations
* consistency in how data is formatted
* proper infrastructure abstractions to enable evolution of big data handling

If developers choose not to use this framework:

* there are no assurances the metrics collected for a business application will be coherent and usable as a part of the greater GEICO whole
* the infrastructure that supports high volumes of event traffic will become non-standardized and harder to maintain
* The infrastructure used to support data aggregation will have to deal with an additional “one off” event source

# Instructions

## Overview

The Event Processing Framework is a small footprint framework designed to provide arbitrary publication of application data based on application specific requirements. In addition, this framework captures “system corollaries” for every event, and enables sequencing to represent start, continuing and completion activities in an application.

### Framework Packages

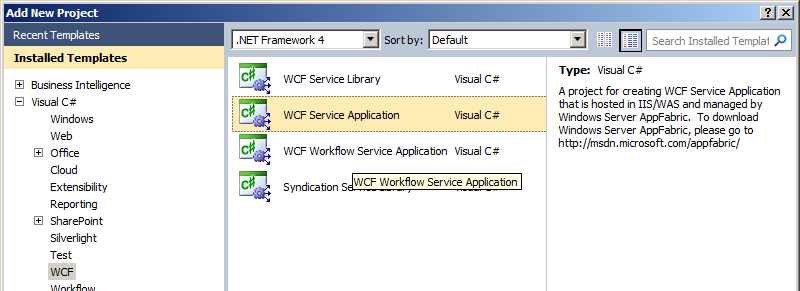
|  |  |
| --- | --- |
| Package | Description |
| Applications.Foundation.EventProcessing | This is the core framework package and is required for all applications that will publish events. |
| Applications.Foundation.EventProcessing.Testability | This package contains supporting implementation to aid in test construction and verification of business events that are raised in application code. |
| Applications.Foundation.EventProcessing.LoggingService | Developer service to allow for local workstations to capture and log events outside of a test context (running a UI locally for example). |
| Applications.Foundation.EventProcessing.StagingService | Infrastructure service to enable business events to be pushed to the Event Framework SQL Server staging DB (note: this requires SQL Server 2012 Developer or Enterprise (or greater) editions).  Developers may opt to use this service if desired, with some additional local configuration to establish the staging database. |

## Developer Configuration

As a general best practice, build your solution in release mode and run all unit tests to ensure there are no pre-existing errors that may be confusing when consuming a new package. If the CI or your local build is broken, fix these issues first before proceeding.

### Add LoggingService Package to Solution

For scenarios where you will need to run an application outside of MS Test (web UI or executable), and you have code in the application that will publish events, you will need to establish the event logging service in your solution. To do so, you must add a WCF Service Application project:



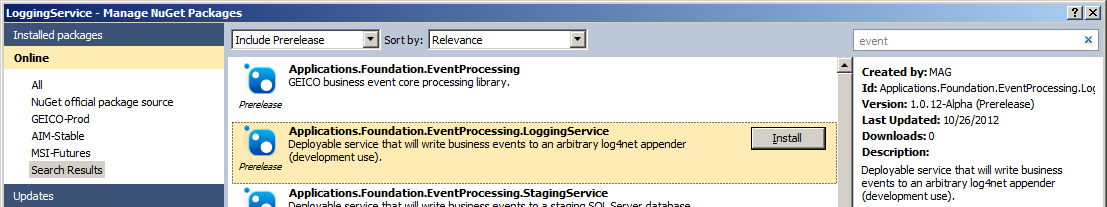
Name this project according to your solution conventions with a name that infers “Business Event Logging”. Once created, clean up template generated files:



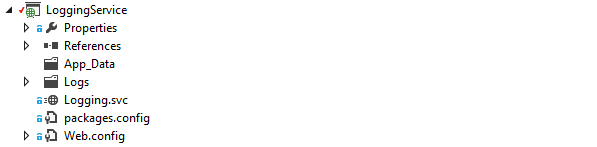
Delete the default template files.

While this project is expected to have the standard Code Analysis and Style Cop configurations, it will not contain any code assets.

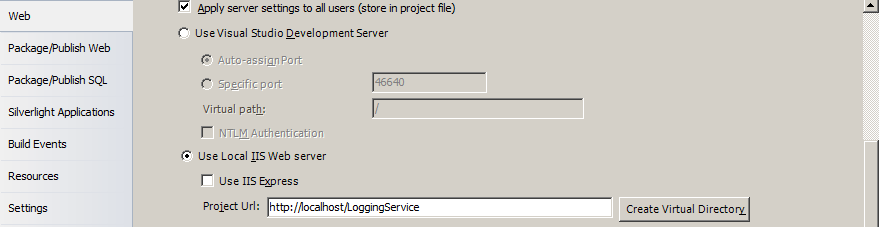
Next, add the Applications.Foundation.EventProcessing.LoggingService package to this project:



This will add the Logging.svc file used by IIS to launch the logging service. This package also adds a “Logs” subdirectory which will contain daily logs of raised events, and this package installation will merge service specific WCF configuration to web.config. Your project should now look similar to as follows[[1]](#footnote-1):

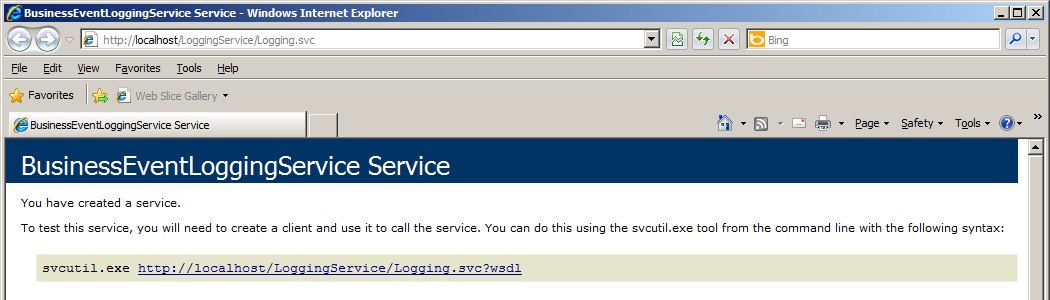


The recommended configuration is to use IIS7[[2]](#footnote-2) or later to host this service. On the project properties Web tab, specify “Use Local IIS Web Server” and accept the default URI:



Once the desired URI is set, click “Create Virtual Directory” and accept any subsequent dialogs. You will need this URI later to set up the application or web.config file for your application code.

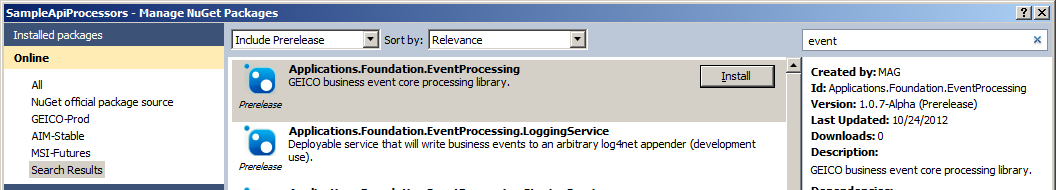
From Solution Explorer, right click on the *Logging.svc* file and select “Browse”. You will see a screen similar to as follows:



You now have a local service that can receive published events. You will need the URL of this service for the following step.

### Add Core Framework Package

To start with this framework, add the **Applications.Foundation.EventProcessing** package to your projects from which you will raise events (you may search using the “event” keyword):



This package has the following dependencies[[3]](#footnote-3):

<dependency id="Newtonsoft.Json" version="[4.5,4.6)" />

<dependency id="Applications.Foundation.ServiceModel.ChannelBuilder" version="[1.0,2.0)" />

<dependency id="Applications.Foundation.ExceptionHandling.Aspects" version="[3.0,4.0)" />

This package will merge the WCF client configuration (system.ServiceModel) into a local application configuration file. You must take this configuration and merge it into to any top level hosting application’s configuration file (typically web.config):

**Binding configuration used for HTTP**

The highlighted text above is where the URL for the logging service should be used. The whole “endpoint” XML node should be uncommented once this configuration is properly placed.

<system.serviceModel>

<bindings>

<wsHttpBinding>

<binding name="localWsHttp16KBTxBinding" maxReceivedMessageSize="16384" transactionFlow="true">

<security mode="None" />

</binding>

</wsHttpBinding>

<!-- Enable this binding if you intend on using transactional MSMQ queues. -->

<!--<netMsmqBinding>…*omitted*…</netMsmqBinding>-->

</bindings>

<client>

<!-- Example client configuration for calling code (requires the wsHttpBinding fragment specified for the

service)-->

<!-- The address below will be the local address of the target web service and will vary by solution and IIS or

IIS Express -->

<!--<endpoint name="BusinessEventService"

address="http://<your target endpoint>"

binding="wsHttpBinding"

bindingConfiguration="localWsHttp16KBTxBinding"

contract="Geico.Applications.Foundation.EventProcessing.IBusinessEventService"/>-->

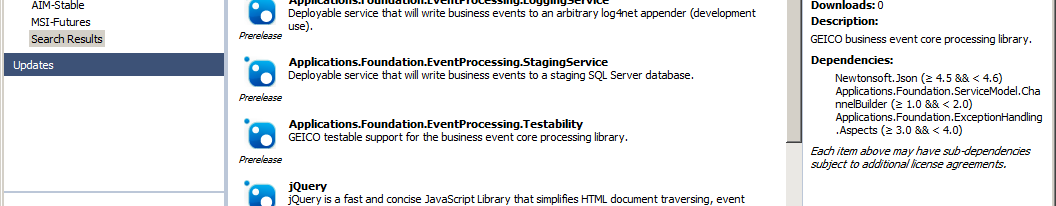
</client>

</system.serviceModel>

### Add Testability Package to Test Projects

To support unit testing to verify if your code is raising the expected events, you will find the Applications.Foundation.EventProcessing.Testability package indispensable.

Add the ***Applications.Foundation.EventProcessing.Testability*** package to unit testing projects that will test code that will raise business events, to ensure event behavior is verifiable:



This package contains supporting implementation that helps with hosting the WCF infrastructure required by this framework in the MS Test runtime. Additionally, there is a helper class to capture the raised events in memory for test verification. This is discussed in greater detail in the Developer Usage section.

## Developer Usage

### Raising an Isolated Business Event

The framework is shaped for calling code to pass the data elements described in the following sections. Below shows the simplest code possible for raising a business event:

// Raise an event to track the fact this activity happened, with no details needed:

new MyAppBusinessEvent<MyApplicationBusinessKeys>("MyActivity").WithKeys(businessKeys).Raise();

The above code snippet shows usage with an application specific Business Event raising a specific event with an event classification of “*MyActivity*” and application specific business keys called “*MyApplicationBusinessKeys*”. The “businessKeys” argument is an instance of this class containing at least one valid (non-empty) value for an application specific business key. Consuming applications are required to create: 1) an application-specific business keys class and, 2) an application-specific business event class (using the business keys) which inherits from the BusinessEvent abstract class. An example is shown below:

// Required applilcation-specific classes

public class MyAppBusinessEvent : BusinessEvent<MyApplicationBusinessKeys>

{

    public MyAppBusinessEvent(string eventClassification) : base(eventClassification)

    {

    }

    public override string ApplicationId

    {

        get

        {

            return "My App Name";

        }

    }

}

public class MyApplicationBusinessKeys

{

    public string RetentionKey { get; set; }

}

#### Event Classification

All events will require that a name for the broad event category be defined. This value is a string that represents this category, or “classification”[[4]](#footnote-4). These values are expected to be widely known and consistent, and driven by application and enterprise requirements.

#### Business Keys

These are the important keys that are application / session wide, such as policy number, quote retention key, email address, client identifier, etc.

Each application is expected to define one or more classes that express what business keys are passed when raising business events. This is driven by application and enterprise requirements, for example:

public class MyApplicationBusinessKeys

    {

        public string PolicyNumber { get; set; }

        public string ExceedClientId { get; set; }

    }

### Raising an Isolated Event with Details

Event details introduce two additional data elements, detail classification and the event details. At least one valid (non-empty) detail must be provided.

This usage is expected to look similar to the following example:

new BusinessEvent("B2BAggregators")

.WithKeys(

new B2BAggregatorKeys

{

LeadProvider = "AllWebLeads",

RecipientEmail = "pavado@yahoo.com",

ApplicationId = "29309158"

})

.Raise(

"BouncedEmail",

new

{

Template = "B2BEstimatedPre...",

DateSent = "10/4/2012 1:28PM",

DateBounced = "10/4/2012 1:28PM"

});

This code example demonstrates raising an isolated event with detailed information (values are string constants for illustrative purposes only). The highlighted type is another illustration of an application defined class that contains the keys used to cross-reference and aggregate business events.

#### Detail Classification

This must be a valid (non-empty) value that classifies the event details, similar to how the event classification classifies the whole event. These values are expected to be widely known and consistent, and driven by application and enterprise requirements.

#### Event Details

This is a dynamic type that does not require an application create a specific class for every possible combination of available detail; instead, the developer may simply “new up” an anonymous type and define the detail properties and values inline.

At least one valid (non-empty) value must be provided. For example, the following code will throw an exception:

new BusinessEvent("B2BAggregators")

.WithKeys(

new B2BAggregatorKeys

{

LeadProvider = "AllWebLeads",

RecipientEmail = "pavado@yahoo.com",

ApplicationId = "29309158"

})

.Raise(

"BouncedEmail",

// Empty details -> Raise an event with no arguments instead if details are not needed:

new

{

});

### Raising an Event Sequence

For business events that have an easily understood start and end (and, if applicable, middle), you can use this framework to raise an event sequence. An event sequence is allows for a single event to be raised multiple times but be distinguished as a “start”, “subsequent” or “completion” event record. A common scenario for this usage is logging web page activity to understand how an end user navigates a single page in an application.

The following example demonstrates using an event sequence:

var keys = new MyBusinessKeys

{

PolicyNumber = "9999999999",

UserEmail = "clickhappy@looneytunes.com",

};

var addVehicleSequence = new BusinessEvent("Add-Vehicle").WithKeys(keys).RaiseStart();

// End web request, go back to cache and save the “open” sequence.

// ..and we are back on a different server in the farm.

// Subequent events require details and use the established business keys for the event instance:

addVehicleSequence.RaiseSubsequent("Add-Vehicle-Select-Year", new { SelectedYear = 2009 });

// End web request, go back to cache and save the “open” sequence.

// ..and we are back on a different server in the farm, posting back the whole add vehicle page.

// Completion events may optionally have details, and use the established business keys for the event instance:

addVehicleSequence.RaiseCompletion();

Event sequences are ordered as they happen, providing end user navigation insights for asynchronous callbacks, for one example. Event sequences use the same underlying unique identifier but are distinguished by a sequence number. All events in an event sequence relate to the same business keys.

Event sequences follow the same rules as isolated events regarding required values, with one exception: *Subsequent* events in a sequence cannot be raised without details [[5]](#footnote-5). A start or completion event allows for optional details in the same manner as for an isolated event.

Once a sequence raises a “Completion” event, it is considered finished and no additional events in the sequence may be raised. If an event is raised for a sequence after completion an exception is thrown.

The Business Event instance for the sequence can be serialized into a session cache in between server round trips (see the section regarding serialization considerations for more details). While implementation of session management is outside the scope of this document, this is expected to be a common, supported scenario essential to raising an event sequence.

### Unit Testing with the Framework

Unit testing this framework is more complex than unit testing code directly because events are published via WCF asynchronously by default. These complexities are managed using the testability library provided by the Applications.Foundation.EventProcessing.Testability package. Adding this package to a unit test project modifies the app.config file by merging the following WCF configuration:

<system.serviceModel>

<services>

<service name="Geico.Applications.Foundation.EventProcessing.Testability.TestEventListener">

<endpoint address="net.pipe://local/businessEventListener"

binding="netNamedPipeBinding"

bindingConfiguration="local16KBTestBinding"

contract="Geico.Applications.Foundation.EventProcessing.IBusinessEventService" />

</service>

</services>

<bindings>

<netNamedPipeBinding>

<binding name="local16KBTestBinding" maxReceivedMessageSize="16384">

<security mode="None"/>

</binding>

</netNamedPipeBinding>

</bindings>

<client>

<endpoint name="BusinessEventService"

address="net.pipe://local/businessEventListener"

binding="netNamedPipeBinding"

bindingConfiguration="local16KBTestBinding"

contract="Geico.Applications.Foundation.EventProcessing.IBusinessEventService"/>

</client>

</system.serviceModel>

This configuration establishes both the client and server for the test execution context. To initialize and tear down this service, each unit test class should specify the following methods with the highlighted lines of code:

[ClassInitialize]

public static void ClassInitialize(TestContext context)

{

TestEventListener.HostHelper.Initialize();

}

[ClassCleanup]

public static void ClassCleanup()

{

TestEventListener.HostHelper.Teardown();

}

The initialize method opens the WCF service host, and the tear down method closes this same service host. The tests that execute in the test class will raise events that are received by the “TestEventListener” service.

The following code example demonstrates a test setup that will allow for verification of events raised by the subject under test:

[TestMethod]

public void Invoke\_XXX\_should\_raise\_one\_event\_with\_expected\_keys\_and\_details()

{

var results = new List<EventRecord>();

using (new AsyncEventScenario(results, 1))

{

CallMyCodeThatRaisesAnEvent();

}

Assert.AreEqual(results.Count, 1);

Assert.AreEqual(results[0].BusinessKeys.Count, 2);

Assert.AreEqual(results[0].BusinessKeys["PolicyNumber"], "XXXXXXXXX");

Assert.AreEqual(results[0].BusinessKeys["ExceedClientId"], "12345");

// Verify implementation

Assert.AreEqual(results[0].Details["ActivityData"], "ThisIsATest");

Assert.AreEqual(results[0].Details.DetailClassification, "LobActivityXXX");

}

The highlighted code is a hypothetical call to the code under test. The assertions are interrogating a collection of EventRecord instances captured from the listener WCF service. An EventRecord instance is the output produced when raising any event. Of note is that the business key and details raised with the event are flattened into name / value strings. The values shown are arbitrary values that are expected to be raised by the test subject.

Also of note is the *AsyncEventScenario* class. This is a disposable class intended to be used within a “using scope”. The AsyncEventScenario class provides the following services:

* A means to capture the event records raised by the test subject and sent to the WCF service
* A default behavior to wait for a configurable number of expected events raised asynchronously

The scope of the AsyncEventScenario must surround the code that will raise the expected business events.

While the AsyncEventScenario class will manage thread synchronization, when running unit tests on build servers you may see that unit tests fail unexpectedly. This can happen when the WCF listener takes longer than expected to initialize and the default timeout of 750 milliseconds is exceeded for a test scenario. It is recommended for test configurations the following application configuration value be added:

<appSettings>

<add key="BusinessEventPublisher.DisallowAsync" value="true"/>

</appSettings>

Despite the fact you can configure this timeout, it is more robust to simply disable asynchrony for unit testing.

**NOTE: Do not disable asynchronous publication for production code.**

# Additional Information

## Captured System Corollaries

In addition to the data provided by the application, every event raised will make a “best effort” to capture the following information:

|  |  |
| --- | --- |
| Property | Description |
| EventTime | The time the event was raised. |
| EventId | The globally unique technical event identifier. |
| EventType | An enumerated value representing the type of event: Isolated, Start, Subsequent, or Completion. |
| EventSequenceNumber | Value indicating the position of the event in a sequence. Isolated and Start event types will always have a value of 0. |
| HttpUserAgent | Gets the user-agent header value from the HTTP context if it is available when the event is raised. |
| HttpSessionId | Gets the session ID value from the HTTP context if it is available when the event is raised. |
| HttpHostUserName | Gets the end user host name value from the HTTP context if it is available when the event is raised. |
| SoapOperationSessionId | Gets the session ID from the WCF operation context if it is available when the event is raised. |
| SoapOperationEndpointAddress | Gets the endpoint address of the hosting service if it is available when the event is raised. |
| LocalTransactionId | Gets the transaction ID of the current transaction, if it exists. |
| DistributedTransactionId | Gets the ID of the current distributed transaction, if it exists. |
| MachineName | Gets the name of the machine raising the event. |
| WindowsIdentityName | Gets the name of the windows identity of the process that raises the event. |
| SourceAssemblyName | Gets the name of the .NET Assembly that raises the event. |
| SourceAssemblyVersion | Gets the runtime version of the .NET Assembly that raises the event. |
| SourceAssemblyProductName | Gets the name of the greater product that is composed of the Assembly raising the event. |
| SourceAssemblyInformationalVersion | Gets the version of the greater product that is composed of the Assembly raising the event. |
| SourceAssemblyFileVersion | Gets the build and revision numbers of the .NET Assembly that raises the event, providing traceability to source code and build. |

## External Correlations

The framework allows for passing an optional unique identifier when setting the event business keys as follows:

var keys = new MyBusinessKeys

{

PolicyNumber = "9999999999",

UserEmail = "clickhappy@looneytunes.com",

};

var addVehicleSequence = new BusinessEvent("Add-Vehicle").WithKeys(keys, myCorrelationId).Raise();

This “correlation ID” provides the ability to relate events across systems and subsystems by a technical key. This key can be used to bridge gaps in analysis when tying together events that happen across the enterprise and may not have overlapping business keys. This key may be provided in the same manner for event sequences.

## Serialization Considerations

Types in this framework are designed to be serialized to JSON format using the Json.Net serializer. Consumers can use the ***EventSerializer*** class to perform serialization to / from a string. String data is implicitly serializable using all .NET serialization technologies.

The ***BusinessEvent*** class may also be directly serialized using the following serializers:

* DataContractSerializer
* NetDataContractSerializer
* BinaryFormatter

It is highly recommended the provided framework serialization methods are used because of the efficiency and small size of the resulting data.

## Infrastructure and Deployment Considerations

This framework addresses how events will be raised and published from application code, and provides a partitioned staging database designed for large data.

This framework does not address the management of large data aggregates, analysis of these aggregates, and corresponding reporting infrastructure. Data in the staging database is expected to be re-cast according to application specific requirements .

Additionally, the events generated by this framework are designed to be compatible with document oriented NoSQL data stores.

This framework is designed to rely on established .NET configuration management practices to deal with environmental differences.

1. The screenshot shows the Visual Studio 2012 visual style; the artifacts in your project will have matching names. [↑](#footnote-ref-1)
2. IIS Express may be used but will require being started when debugging or running your application locally. [↑](#footnote-ref-2)
3. The Applications.Foundation.ExceptionHandling.Aspects package will pull in additional dependencies on the foundation logging framework. This is used to support logging of exceptions that happen when events are raised asynchronously (“fire and forget”). [↑](#footnote-ref-3)
4. The term “category” was avoided because “Event Category” has specific technical meaning when dealing with the Windows Event Log system. [↑](#footnote-ref-4)
5. A subsequent event with no details is meaningless; without details, it has no identifying attributes. “Start”, “Completion” and isolated events offer intrinsic meaning without details. [↑](#footnote-ref-5)