



How to deploy an additional module

Step by step guide

The AgentService Team

Copyright @ 2007 – l.i.d.o. – DIST University of Genoa

Abstract

This tutorial will help you to deploy an additional module in the AgentService Platform. We propose the implementation of a simple message sniffer module with a graphical interface, showing every step necessary to its implementation. A simple graphical interface is included in the downloadable example, in order to use practically the module.

The GUI interface code is separated from the basic implementation of the module, in order to simplify the understanding of the module functioning.

Index of Contents

Abstract	2
Index of Contents	3
1. Introduction	4
2. The AgentService module	4
2.1. The IPlatformModule interface	5
2.2. The IPlatformContext interface	6
2.3. The IEventSinkManager interface.....	7
2.4. The IEventManager interface.....	8
2.5. The PlatformEventArgs class	9
2.6. The ModuleInfo class	10
3. Implementing the message sniffer module	11
3.1. The SnifferInfo class.....	12
3.2. The SnifferModuleClass	12
3.3. Adding the sniffer module to the platform	17
3.4. An overview of the sniffer GUI.....	18

1. Introduction

The AgentService architecture is strongly modular: several functionalities of the platform (both core and additional) are implemented as a module.

A module is a plug-in which has to be installed in the AgentService platform, in order to be loaded when the platform is executed. By using the module plug-ins, the AgentService user is able to modify default modules and to add additional features, fully integrated with the life-cycle of the platform.

To create a new additional module, the developer must implement some standard interfaces, described in the chapter 2.

The message sniffer module, which is reported as an example, is also freely downloadable at <http://www.agentservice.it/download/snifferModule.zip>. The implementation of the sniffer module is fully described in chapter 3.

2. The AgentService module

To create a new module, the developer has to implement the *IPlatformModule* interface. A module can be a *core module* or *additional module*. If the developer wants to re-design a core module, has to implement one of the following interfaces:

- *ILoggingModule*: it represents the module which manages the logging service of the platform.
- *IMessagingModule*: it represents the module which manages the messaging subsystem.
- *IPersistenceModule*: it represents the module which manages the persistence service.
- *IStorageModule*: it represents the module which manages the storage service.

Each of these interfaces derives from the *IPlatformModule* interface.

As shown in the Figure 1, the *IPlatformModule* interface exposes four methods which are called during the whole life-cycle of the module. The *install* method is called during the first execution of the platform, when the latter is in the *binding* state. The module is then loaded and unloaded from the platform using *attach* and *detach*, respectively during the states *running* and *shutting down*.

Uninstall is called when the module is going to be uninstalled (e. g. typing *asdrv -u*, to uninstall the whole platform).

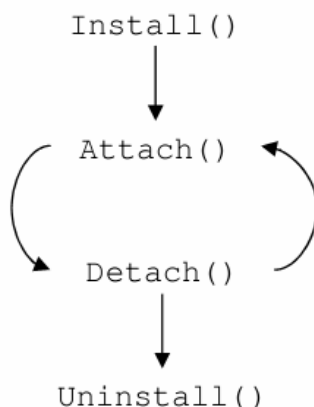


Figure 1: the life-cycle of a module.

2.1. The *IPlatformModule* interface

The *IPlatformModule* interface defines the common functionalities which every module must have.

```

public interface IPlatformModule
{
    string Name { get; }
    string Description { get; }
    ModuleCategory Category { get; }
    bool IsMain { get; }
    bool CanSupport(string capabilityInterface);
    IModuleError GetLastError();
    bool Attach(IPlatformContext hostingPlatform, ModuleInfo info);
    bool Detach();
    bool Install(IPlatformContext context, ModuleInfo info, ref
        InstallationInfo instInfo);
    bool UnInstall(InstallationInfo instInfo);
}
  
```

Properties:

- `string` Name: gets the name of the module.
- `string` Description: gets a brief textual description of the module.
- `ModuleCategory` Category: gets the module category, described by the following *enum*:

```

[Flags]
public enum ModuleCategory
{
    Core = 0x0100,
    Messaging = 0x0101,
    Persistence = 0x0102,
    Storage = 0x0104,
}
  
```

```
    Logging = 0x0108,  
    Other = 0x0200  
}
```

- `bool` `IsMain`: returns true if the module is able to support the basic functionalities of the module category.

Methods:

- `bool` `CanSupport(string capabilityInterface)`: checks if the module supports the features defined by the interface passed as parameter.
- `IModuleError` `GetLastError()`: returns the last error occurred during the module execution. It returns null if no error occurred.
- `bool` `Attach(IPlatformContext hostingPlatform, ModuleInfo info)`: this method is called every time the module is loaded. It receives two parameters: an `IPlatformContext` object and `ModuleInfo` object, both described in paragraphs 2.2 e 2.6.
- `bool` `Detach()`: is invoked when the module is unloaded; it returns true if the detachment if errors don't occur.
- `bool` `Install(IPlatformContext context, ModuleInfo info, ref InstallationInfo instInfo)`: this method is invoked during the installation of the process. `IPlatformContext` and `ModuleInfo` are showed in paragraphs 2.2 e 2.6, while the class `InstallationInfo` contains installation information. This class is an empty abstract class, totally customizable. `InstallationInfo` must be serializable in order to store in a file the contained information, used by the `uninstall` method at the end of the module execution.
- `bool` `UnInstall(InstallationInfo instInfo)`: it receives an `InstallationInfo` object and returns true if the process ends successfully.

2.2. The `IPlatformContext` interface

This interface defines the basic functionalities a module can handle. These functionalities can be used in the `install` and `attach` methods. `IPlatformContext` is composed by:

- Standard events raised by the platform.
- Read only properties returning information about the platform.
- Methods to manage and raise user-defined events.

Moreover, `IPlatformContext` inherits from `IEventSinkManager` and `IEventManager`, described in paragraphs 2.3 e 2.4.

This is the structure of the interface:

```

public interface IPlatformContext : IEventSinkManager, IEventManager
{
    string PlatformBaseDir {get;}
    string PlatformCoreConfPath {get;}
    string PlatformCoreBinPath {get;}
    string PlatformAddConfPath {get;}
    string PlatformAddBinPath {get;}
    IPlatformDescription Description {get;}
    bool IsFirstStart {get;}
    void PostCommand(PlatformCommand cmd);
}

```

There are five properties which get the paths respectively of the platform directory, the directory containing configuration files of core modules, the directory containing assemblies of core modules, the directory containing configuration files of additional modules, and the directory containing assemblies of additional modules.

The property *Description* gets the platform description, which contains the platform name, indicates if the platform is dynamic (that is if the platform accepts the import of mobile agents) and mobile, and returns the transport description (it is a string containing information about communications inter-platforms as: protocol, ip address, port, etc.).

IsFirstStart returns true if the platform is started for the first time.

PostCommand posts a command to the platform command queue.

In the next paragraphs *IEventSinkManager* and *IEventManager* are briefly described, in order to give a complete overview of the platform context that a module has at its disposal.

2.3. The *IEventSinkManager* interface

The *IEventSinkManager* interface manages the list of event sinks registered to the platform and forwards the appropriate events to the modules owning the sinks.

```

public interface IEventSinkManager
{
    EventSinkDescriptor[] GetRegisteredSinks();
    SinkStatusCodes RegisterEventSink(EventSinkDescriptor
        evtSink, PlatformEventSinkHandler handler,
        out int token);
    SinkStatusCodes DeregisterEventSink(EventSinkDescriptor
        evtSink, int token);
    SinkStatusCodes PumpEvent(string evtName, PlatformSinkEvent
        evt);
}

```

Methods:

- `EventSinkDescriptor[] GetRegisteredSinks()`: gets the list of registered events sinks.
- `SinkStatusCodes RegisterEventSink(EventSinkDescriptor evtSink, PlatformEventSinkHandler handler, out int token)` : it registers an event sink, receiving the descriptor of the event sink, the handler of the event, and the owning token. The token is the unique identifier of a module; it is automatically generated during the the module instantiation. The methods returns the sink status that can be one of the following types:
 - o SinkOk, if the registration is done successfully.
 - o SinkNotFound, returned if the sink is not registered or present.
 - o SinkAlreadyExists, in case of an already registered event sink.
 - o SinkNotOwned, if the module tries to deregister an event sink it doesn't own.
 - o SinkInvalidEvent, if the module submits an invalid event sink.
- `SinkStatusCodes DeregisterEventSink(EventSinkDescriptor evtSink, int token)` : deregisters an event sink.
- `SinkStatusCodes PumpEvent(string evtName, PlatformSinkEvent evt)`: raises an event with a given name for a particular event sink.

2.4. The IEventManager interface

The *IEventManager* interface provides the standard platform events and the methods for the management of these events. Below is reported the structure of the *IEventManager* interface.

```
public interface IEventManager
{
    event PlatformEventHandler OnAgentCreated;
    event PlatformEventHandler OnAgentResumed;
    event PlatformEventHandler OnAgentStopped;
    event PlatformEventHandler OnAgentWokenUp;
    event PlatformEventHandler OnMessageSent;
    event PlatformEventHandler OnMessageReceived;
    event PlatformEventHandler OnMessageBatch;
    event PlatformEventHandler OnConversationRequested;
    event PlatformEventHandler OnConversationAccepted;
    event PlatformEventHandler OnConversationRefused;
    event PlatformEventHandler OnConversationClosed;
    event PlatformEventHandler OnAgentPersisted;
    event PlatformEventHandler OnAssemblyUploaded;
    event PlatformEventHandler OnAssemblyRemoved;
}
```

```

void RaiseStandardEvent(PlatformEvent eventType,
    PlatformEventArgs args, int token);
int GetToken(IPlatformModule module);

int AddEvent(string evtName);
bool RemoveEvent(int token);
bool RaiseEvent(int token, PlatformEventArgs args);
bool Register(string evtName, PlatformEventHandler handler);
bool Deregister(string evtName, PlatformEventHandler handler);
}

```

Avoiding the mere listing of the supported events, it is clearly visible that there are available event regarding the agent life-cycle, the message sending and receiving, the life-cycle of a conversation, and the upload (or removal) of an assembly.

Two methods manage standard events:

- `void RaiseStandardEvent(PlatformEvent eventType, PlatformEventArgs args, int token)`: raises a standard event, notifying it to the registered modules, passing the arguments for the platform event and the token.
- `int GetToken(IPlatformModule module)`: returns a valid token for the given module.

Other methods allow the user to manage custom events:

- `int AddEvent(string evtName)`: adds an event with the given name and returns the token to use in order to track the event.
- `bool RemoveEvent(int token)`: removes the event associated to the given token.
- `bool RaiseEvent(int token, PlatformEventArgs args)`: raises the event associated to the given token, passing an object containing the platform event arguments (see paragraph 2.5).
- `bool Register(string evtName, PlatformEventHandler handler)`: registers a handler to the given event.
- `bool Deregister(string evtName, PlatformEventHandler handler)`: deregisters a handler to the given event.

2.5. The PlatformEventArgs class

It defines a class for the arguments of every platform event handler. As shown in the following code, the class associates the type of an event with its name and additional data.

```

public class PlatformEventArgs
{
    private PlatformEvent type;
    public PlatformEvent Type
    {
        ...
    }
}

```

```

private string name;
public string Name
{
    ...
}
protected object data;
public object Data
{
    ...
}
public PlatformEventArgs(PlatformEvent type, string name)
{
    ...
}
public PlatformEventArgs(PlatformEvent type, string name, object
                        data) : this(type, name)
{
    ...
}
}

```

The *Data* property is of type object, so the event argument can be fully customizable. There are two constructors: the former receives only event type and name. The latter receives the additional data too.

2.6. The ModuleInfo class

The *ModuleInfo* class implements the *IModuleInfo* interface. It holds information about the module. Below is reported the *IModuleInfo* interface:

```

public interface IModuleInfo
{
    string Key {get;}
    ModuleInfoState State {get;}
    string FullTypeName {get;}
    string AssemblyName {get;}
    string ConfigFile {get;}
    ModuleCategory Category {get;}
    string Name {get;}
    string Description {get;}
}

```

This is the meaning of the properties:

- `string` Key: the unique identifier of the module.
- `ModuleInfoState` State: the state of the module (if it is new, installed or pending, it is used in case the module is going to be installed in an already deployed platform).
- `string` FullTypeName: the name of the type which implements the module.
- `string` AssemblyName: the name of the assembly containing the module.
- `string` ConfigFile: the name of the configuration file.

- `ModuleCategory` Category: the module category (core, messaging, storage, persistence, logging, other).
- `string` Name: the module name.
- `string` Description: the description of the module.

3. Implementing the message sniffer module

In this chapter we show how to implement a simple additional module: the message sniffer module. This module will be able to sniff the messages sent by the agents, attaching itself to the events raised by the platform, which are, relating the message subsystem:

- Message sent: raised when a new message is sent by an agent.
- Message received: raised when a message is received by an agent.
- Message batch: raised when a bulk of messages is received.

You can download on <http://www.agentservice.it/download/snifferModule.zip> the entire solution containing both the module implementation and the graphical user interface respectively with project name: *MessageSniffer* and *SnifferGUI*. The aim of this guide is to accompany the user in his first module development and then we will concentrate on the *MessageSniffer* project. As you can see opening the solution, there are two `.cs` files:

- *SnifferInfo.cs* which contains the *SnifferInfo* class.
- *SnifferModule.cs* which contains the *SnifferModule* class.

These two classes are the only ones necessary for the implementation of a module.

Please also note the references of this project:

- *AgentService.dll*: it is the core of the AgentService platform.
- *AgentServiceInterface.dll*: it contains several interfaces and classes useful to remotely access the platform (for the message sniffer purpose, it is not necessary to deeply know the meaning of this assembly).

- *AgentServiceDefaultModules.dll*: here are hosted the classes which implement the default core modules of AgentService and others related classes and interfaces.
- *SnifferGUI.dll*: this assembly contains the graphical user interface of the sniffer module.

3.1. The SnifferInfo class

The first point is to implement the *SnifferInfo* class. It manages the installation information of the module. In this case, the class provides only one property called *StartRecord*, which returns *true* if starts automatically to record the messages sent by agent, *false* otherwise (in this case the user must click the *start button* to start sniffing the messages, as you can see in the paragraph 3.4).

The *SnifferInfo* must to implement the *InstallationInfo* interface (see paragraph 2.1). Here is reported the whole *SnifferInfo* class:

```
[Serializable]
public class SnifferInfo: InstallationInfo
{
    private bool startRecording;
    public bool StartRecording
    {
        get
        {
            return this.startRecording;
        }
    }

    public SnifferInfo(bool startRecording)
    {
        this.startRecording = startRecording;
    }
}
```

3.2. The SnifferModuleClass

This is the core of the sniffer module. The class must implement the *IPlatformModule*:

```
public class SnifferModule : IPlatformModule
{
```

The properties *Name*, *Description*, *Category* and *IsMain* implement the properties of *IPlatformModule*.

```
private GUIFactory GUI;
private PlatformEventHandler handler;
private ModuleInfo info;
private bool startRecording = true;
private AgentService.Platform.IPlatformContext hapContext;
private string name = "Sniffer Module";
public string Name
{
    get
    {
        return this.name;
    }
}
private string description = "Snooping of messages and
                             conversations: record on file";
public string Description
{
    get
    {
        return this.description;
    }
}
public AgentService.Platform.Modules.ModuleCategory Category
{
    get
    {
        Return AgentService.Platform.Modules.
                ModuleCategory.Other;
    }
}
public bool IsMain
{
    get
    {
        return true;
    }
}
```

There are also private data members as:

- `private GUIFactory GUI`: the object which represent the GUI of the module.
- `private PlatformEventHandler handler`: the handler of the events raised by the platform.
- `private ModuleInfo info`: it contains information about the module (e.g. the module name, the name of the assembly, the configuration file path...)
- `private bool startRecording`: true if the module must start automatically the recording of messages.
- `private IPlatformContext hapContext`: the platform context (see paragraph 2.2).
- `private ModuleError lastError`: holds the last error occurred in the module.

CanSupport returns *true* if the module can support features defined in the given interface. In this project you should return *false* anyway.

```
public bool CanSupport(string capabilityInterface)
{
    return false;
}
```

GetLastError returns the last occurred error:

```
public IModuleError GetLastError()
{
    return this.lastError;
}
```

Now is the turn of the *Install* method:

```
public bool Install(AgentService.Platform.IPlatformContext
hostingPlatform, ModuleInfo info, ref InstallationInfo instInfo)
{
    this.hapContext = hostingPlatform;
    this.info = info;
}
```

During the installing process the configuration file is readed (if it exists, otherwise is created with a default value). Note that in this method the configuration file name is obtained from the *info* object (type: *ModuleInfo*). The name of this file is mentioned only in the platform configuration file (see paragraph 3.3).

Below you can see the statements to create a default config file:

```
if (System.IO.File.Exists(info.ConfigFile) == false)
{
    System.IO.StreamWriter writer = new
    System.IO.StreamWriter(info.ConfigFile);
    writer.WriteLine(bool.TrueString);
    writer.Close();
    instInfo = new SnifferInfo(true);
    this.startRecording = true;
}
```

And here there are the statements for the reading of the configuration file:

```
else
{
    try
    {
        System.IO.StreamReader reader = new
        System.IO.StreamReader(info.ConfigFile);
        string record = reader.ReadLine();
        bool startRecord = false;
        if (record == bool.TrueString)
        {
```

```

        startRecord = true;
    }
    else
    {
        startRecord = false;
    }
    reader.Close();
    instInfo = new SnifferInfo(startRecord);
    this.startRecording = startRecord;
}

catch (System.Exception ex)
{
    this.lastError = new ModuleError(0, info, "Could not read
        configuration file", ex);

    return false;
}
return true;
}
}

```

Please note that the *startRecording* property is initialized with value readed in the configuration file and the same value is stored in *instInfo* (type: *SnifferInfo*).

For the sniffer module there aren't particular operations during the *uninstall* process:

```

public bool UnInstall(InstallationInfo instInfo)
{
    return true;
}

```

Regarding the *Attach* method, note that receives an *IPlatformContext* object (see the paragraph 2.2):

```

public bool Attach(AgentService.Platform.IPlatformContext
    hostingPlatform, ModuleInfo info)
{
    this.info = info;
    this.hapContext = hostingPlatform;
}

```

Then, it reads the configuration file (remember that during the installation process the config file is automatically created if it doesn't exist), in order to get information about the recording start.

```

try
{
    System.IO.StreamReader reader = new
    System.IO.StreamReader(info.ConfigFile);
    string record = reader.ReadLine();
    if (record == bool.TrueString)
    {
        this.startRecording = true;
    }
}

```

```

        else
        {
            this.startRecording = false;
        }
        reader.Close();
    }
    catch (System.Exception ex)
    {
        this.lastError = new ModuleError(0, info, "Could not read
            configuration file", ex);
        return false;
    }
}

```

Finally, the GUI is loaded, the event handler is instantiated with a the *SnoopMessage* method, and the module is registered to the events raised in case of message sent, received or batch.

```

this.GUI = new GUIFactory(this.startRecording);
this.GUI.Show();
this.handler = new PlatformEventHandler(this.snoopMessage);

this.hapContext.OnMessageSent += handler;
this.hapContext.OnMessageReceived += handler;
this.hapContext.OnMessageBatch += handler;
return true;
}

```

The detachment process consists only in the event deregistration, as performed in the *Detach* method:

```

public bool Detach()
{
    if (this.handler != null)
    {
        this.hapContext.OnMessageSent -= handler;
        this.hapContext.OnMessageReceived -= handler;
        this.hapContext.OnMessageBatch -= handler;
    }
    return true;
}

```

The *SnoopMessage* method manages the incoming message events, for each type of raised event call the *AddMessage* method of the GUI object, to send to the graphical interface the sniffed message.

```

private void SnoopMessage(object sender, PlatformEventArgs args)
{
    MessageEventArgs theArgs = (MessageEventArgs)args;
    switch (args.Type)
    {
        case PlatformEvent.MessageSent:
        {
            this.GUI.AddMessageRecord(PlatformEvent.MessageSent,
                theArgs);
        }
    }
}

```

```

        break;
    }
    case PlatformEvent.MessageReceived:
    {
        this.GUI.AddMessageRecord(PlatformEvent.MessageReceived,
            theArgs);
        break;
    }
    case PlatformEvent.MessageBatch:
    {
        this.GUI.AddMessageRecord(PlatformEvent.MessageBatch,
            theArgs);
        break;
    }
    default: break;
}
}
}

```

3.3. Adding the sniffer module to the platform

Now the implementation of the module is complete. There are two cases:

- The platform has to be installed.
- The platform is already installed.

In the first case you must modify the installation file *install.xml*, otherwise you must modify the *platform.conf* file in the *conf* subdirectory of your AgentService installation.

In *install.xml* you must add these tags into `<installation></installation>`:

```

<additional>
  <module
    full-type-name="AgentService.Platform.Modules.Additional.
                                     SnifferModule "
    assembly-name="C:\MessageSniffer\MessageSniffer.dll"
    config-file="C:\MessageSniffer\snoopy.conf"
  />
</additional>

```

The *full-type-name* attribute contains the namespace of your additional module. *Assembly-name* shows the path of the *dll* containing the module. *Config-file* contains the path of the configuration file of the module.

If the platform is already installed, you can add the module modifying the *platform.conf* file, adding into the `<modules></modules>` tags an xml structure similar to the previous one:

```

<additional>
  <module
    key=""

```

```
    full-type-name="AgentService.Platform.Modules.Additional.  
SnifferModule "  
    assembly-name="C:\MessageSniffer\MessageSniffer.dll"  
    config-file="snoopy.conf"  
  />  
</additional>
```

Please note the new attribute *key*, which hasn't value.

Both in case of a new installation and in case of an addition to an existing platform, you should create a configuration file, in this case called *snoopy.conf*. This file contains only one value: *True* if you want that the sniffer module automatically starts to record the messages, *False* otherwise. Note that the file is not an xml file: you must simply write the word *True* (or *False*) into an empty file called *snoopy.conf*. In case of multiple configuration parameters, each value will fill a new line in the file. Obviously the name of this file can be changed. As you can see in above example, the *config-file* attribute has value *snoopy.conf*. This means that the file *snoopy.conf* is stored in the main directory of your AgentService installation.

Before starting the platform, you must change a setting in the configuration file of the messaging core module. Open the file *modules\core\conf\msmq.conf* and set the parameter *EnableEvents* to *True*. This change allows the messaging module to raise the usual message events.

Now the platform is ready to start also executing the sniffer module. For example you should run the auction demo application, sniffing the messages sent by the bidders and the auctioneer.

3.4. An overview of the sniffer GUI

When you start your AgentService application with the new sniffer module, a simple GUI appears. In Figure 2 is shown the main form of the sniffer module GUI: a list of detected messages. You can read information about the event raised, the ID of the conversation between sender and receiver, the ACL type and additional information related to the *MessageBatch* event. Remember to include *SnifferGUI.dll* in the folder containing the assembly *MessageSniffer.dll*.

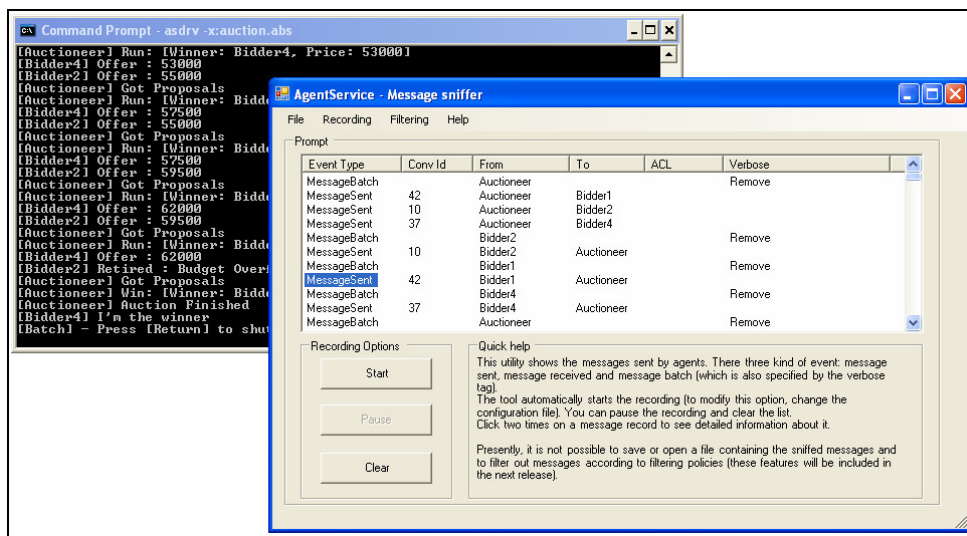


Figure 2: the message sniffer GUI.

If you click on the first element of the select line, a new form is shown (see Figure 3). This form contains additional information of the sniffed message (remember that the object representing the message is passed by the raised event as an event argument additional data).

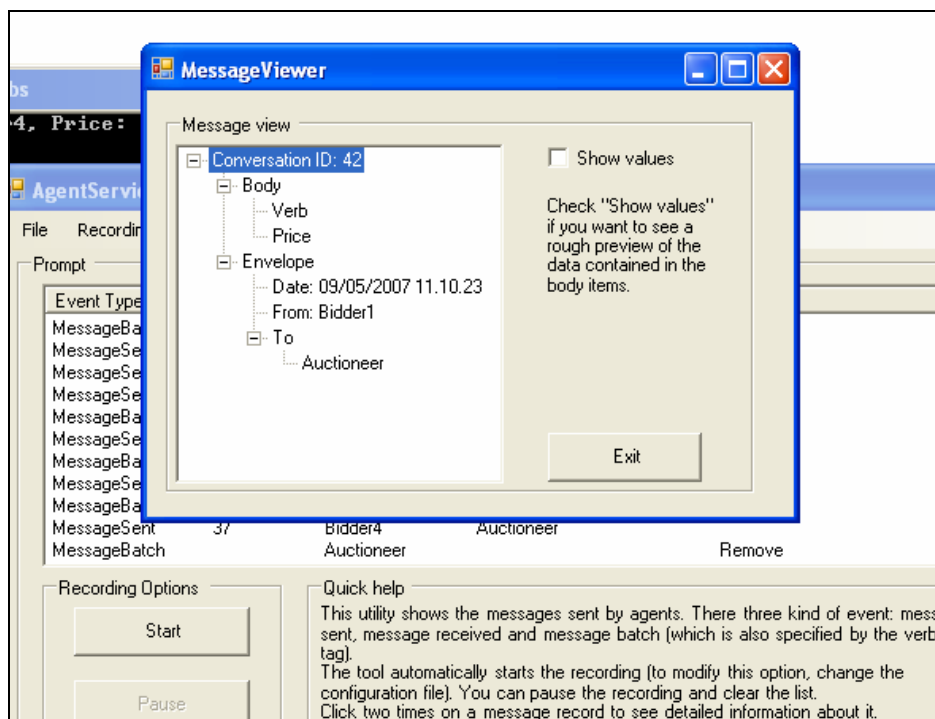


Figure 3: the message viewer.

Check the “show value” checkbox to see a rough representation of the body items content.