


Formpipe.
Lasernet

Lasernet 9
SDK



Lasernet 9 – SDK [Revision 4 – August 2020]

© 2020 Formpipe Software

Lasernet is a trademark of Formpipe Software

Company website www.formpipe.com

Product website www.lasernetbyformpipe.com

1	INTRODUCTION	1
1.1	WHO SHOULD USE THIS GUIDE?	2
2	TERMS OF USE	3
3	MODULE SDK	4
3.1	INTRODUCTION	5
3.1.1	INPUT MODULE	5
3.1.2	ENGINE MODULE	5
3.1.3	OUTPUT MODULE	5
3.1.4	MODIFIER	5
3.1.5	EFSTECH.LASER.NET.DLL	5
3.1.6	YOUR MODULE	5
3.2	LEARNING BY EXAMPLE	6
3.2.1	FILE INPUT MODULE	6
3.2.2	FILE OUTPUT MODULE	10
3.2.3	PASS-THROUGH ENGINE	10
3.2.4	BASE64 MODIFIER	10
3.3	LOGGING	11
3.4	LICENSING	12
3.5	THIRD PARTY MODULE	13
3.5.1	INFORMATION NEEDED IN MODULE DLL FOR THIRD PARTY MODULE	13

1 INTRODUCTION

1.1 Who Should Use This Guide?

This manual is written for external developers who wish to create 3rd party modules for Lasernet in .NET. It is intended primarily to show how to use the different classes and methods which are currently implemented and supported. It also explains the purpose and operation of each module and modifier and how to use them to fit your requirements.

Throughout the document the reader is addressed as you, where “you” means all the above-mentioned persons.

2 TERMS OF USE

No part of this publication may be reproduced, transmitted, transcribed, or translated into any language in any form by any means without the prior written permission of Formpipe Software. The information in this manual is subject to change without notice. Any company names or data is fictive unless otherwise stated.

Formpipe Software shall not be liable for any loss or damage whatsoever arising from the use of this manual and the information contained therein (including errors or omissions).

Trademarks of other companies mentioned in this document appear for identification purposes only and are the property of their respective companies.

© 2020 Formpipe Software

3 MODULE SDK

3.1 Introduction

This manual shows how to extend Lasernet's connectivity and processing abilities to suit your specific needs. Sometimes the standard modules do not fit your exact requirements when trying to connect to systems not widely used. It is also possible that the standard engines or scripting functions do not give you the options you need to fetch, transform or deliver your data. In Lasernet it is possible for any .NET developer to write custom modules for Lascript. Thus, the ability to extend Lascript is almost limitless.

3.1.1 Input Module

Retrieve data from any system according to any schedule or options you like. You can even make the module into a server which waits for other systems to connect and deliver data.

3.1.2 Engine Module

Process the data any way you like. An engine can create new Jobs.

3.1.3 Output Module

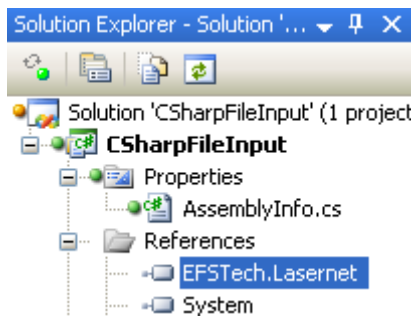
Deliver the data to any system - the way you want.

3.1.4 Modifier

Modify data in any way you wish. A modifier works on just one Job.

3.1.5 EFSTech.Lascript.DLL

The EFSTech.Lascript.DLL is the bridge between the world of Lascript and your world. It contains all the classes you need to make your own modules for Lascript. A reference to EFSTech.Lascript.DLL must be added in your project:



3.1.6 Your Module

A Lascript module is basically a DLL/Class Library that implements a number of classes and methods. Depending on the functionality of your module you can reference the relevant modules and classes from the EFSTech.Lascript.DLL. The .NET Framework 4.8 is installed per standard by Lascript and can be used for your project if it is sufficient for your needs. To compile the DLL on a stand-alone computer without Lascript at least .NET Framework 4.8 must be installed.

3.2 Learning by example

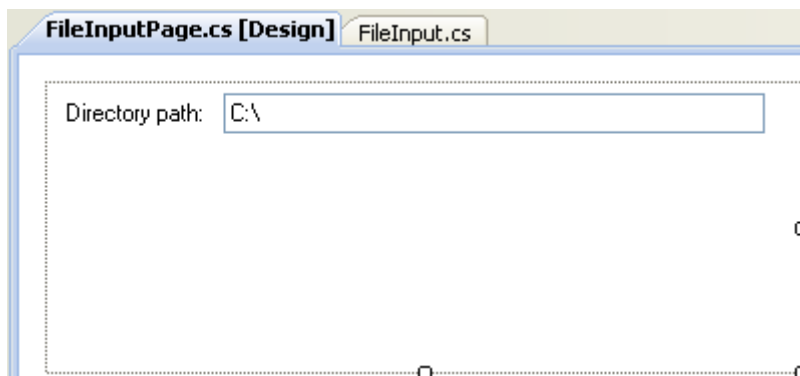
A good starting point is to run through a few simple examples. A File Input module, a Pass-Through module and a File Output module are easy to make and simple to understand.

3.2.1 File Input Module

A simple File Input module needs a configuration page with a folder to look for files in and a scan mask to specify the name(s) of the files. In our example we will just look for *.*.

A page inherits from a normal UserControl. It has a constructor and an ApplyChanges method where a ConfigurationObject is passed. The ConfigurationObject makes it possible to read and write settings in the Lasernet configuration.

A simple page could look like this:



And the code behind reading and writing the path:

```
FileInputPage.cs | FileInputPage.cs [Design] | FileInput.cs
FileInputPage.cs
CSharpFileInput.FileInputPage
1 using System.Windows.Forms;
2 using EFSTech.LaserNet;
3
4 namespace CSharpFileInput
5 {
6     public partial class FileInputPage : UserControl
7     {
8         public FileInputPage(ConfigurationObject moduleConfiguration)
9         {
10             InitializeComponent();
11
12             textBoxDirectoryPath.Text = moduleConfiguration.GetString("DirectoryPath", "C:\\");
13         }
14
15         public void ApplyChanges(ConfigurationObject moduleConfiguration)
16         {
17             moduleConfiguration.SetString("DirectoryPath", textBoxDirectoryPath.Text);
18         }
19     }
20 }
```

The ConfigurationObject (inherited from Settings class) has functions for reading and writing different types of data. In our example we write and read a String with a Key named "DirectoryPath" containing our path. Look at the Settings class in the reference for more info on the different types.

JobModuleUI

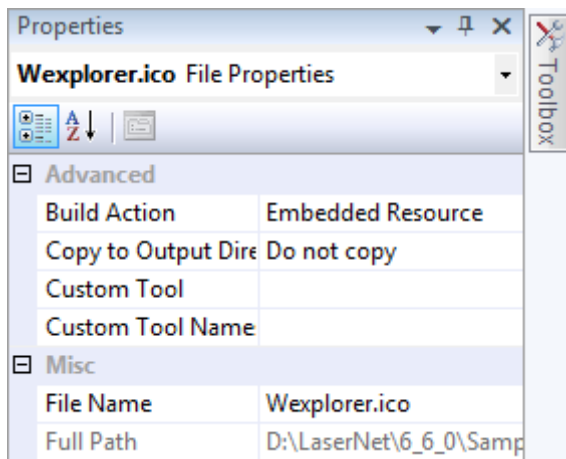
Now that we have our page, we need to implement a JobModuleUI. The JobModuleUI tells Lasetnet what the module can do (input, engine, output), constructs the page(s), sets up the properties for the Property Editor and returns custom icons. NOTE: An icon is not required and an implementation of GetIcon can be omitted.

```

87 [Module("CSharp File Input", JobModuleType.InputPort, "47FEC57F-0AB9-4fd0-8914-15129047B85E")]
88 public class FileInputUI : JobModuleUI
89 {
90     public FileInputUI(JobModuleUIInitData initData)
91         : base(initData)
92     {
93     }
94
95     public override bool CanActAsInput()
96     {
97         return true;
98     }
99
100    public override System.Windows.Forms.UserControl[] CreatePages(ConfigurationObject moduleConfiguration)
101    {
102        pages[0] = new FileInputPage(moduleConfiguration);
103        pages[0].Text = "File Input";
104        return pages;
105    }
106
107    public override System.Drawing.Icon GetIcon()
108    {
109        return new System.Drawing.Icon(this.GetType().Assembly.GetManifestResourceStream("CSharpFileInput.Resources.Wexplorer.ico"));
110    }
111
112    public override void ApplyChanges(ConfigurationObject moduleConfiguration)
113    {
114        ((FileInputPage)pages[0]).ApplyChanges(moduleConfiguration);
115    }
116
117    System.Windows.Forms.UserControl[] pages = new System.Windows.Forms.UserControl[1];
118 }

```

NB! When you add the icon file to the project, you must mark the Build Action as an “Embedded Resource” in the File Properties of the file – otherwise you will keep getting the default icon. Remember you need both 16x16px and 32x32px versions of the icon.



Every module in Lasernet needs a unique GUID. Create your own and use it throughout your code.

The first line in the code above is the construction of a ModuleAttribute for our UI class. It describes the type name of the module, the module type and the GUID of the module.

In CreatePages we can see our settings page being created and given a name. In GetIcon a FileInput.ico is returned which has been added as an Embedded Resource (build type) in our Class Library.

CanActAsInput might seem redundant but must be implemented and return true.

Options:

- Scheduling. Implement the function 'IsSchedulable' and return bits for the type of module where scheduling is enabled. typeInput = 2 and typeOutput = 4. If you want no scheduling enabled, return 0. If you want only scheduling when module is working as an input module, then return 2. If both, return 6.
- Combining. Implement the function 'IsCombinable' in the same fashion as with scheduling above.
- Pausing. Implement 'IsPausable' and return false if you wish to turn this feature off.
- Deliverer. Implement 'deliveryControlEnabled' and return true to enable.
- Prevent spaces in module name by implementing 'allowSpaceInName' and return false.

Property	Value
[-] General	
Active	<input checked="" type="checkbox"/> True
Name	CSharp File Output 1
Descript...	
Created ...	29-11-2010 09:01:16
Modifie...	29-11-2010 12:41:31
Created ...	JACOB\jacobp
Modifie...	JACOB\jacobp
Group	
Language	
Company	
[-] File Output	
Path	C:\LaserNet\Output

The Lasernet Property Editor is supported and recommended but not required. Adding Properties to the module is easy;

```

355
356 public override void ApplyChanges(ConfigurationObject moduleConfiguration)
357 {
358     ((FileInputPage) pages[0]).ApplyChanges(moduleConfiguration);
359 }
360
361 private System.Collections.Generic.List<PropertyBase> properties = null;
362 private System.Windows.Forms.UserControl[] pages = new System.Windows.Forms.UserControl[1];
363 }
364 }
365

```

Implementing GetProperties is all which is needed. Here properties are added to the list;

```

41
42 public override System.Collections.Generic.List<PropertyBase> GetProperties(ConfigurationObject moduleConfiguration)
43 {
44     if (properties == null)
45     {
46         properties = new System.Collections.Generic.List<PropertyBase>();
47
48         PropertyBase propGroup = new PropertyBase();
49         propGroup.Id = (int)PropertyId.pidGrpDirSetup;
50         propGroup.Name = "File Output";
51         properties.Add(propGroup);
52
53         PropertyString propstr = new PropertyString();
54         propstr.Parent = propGroup;
55         propstr.Id = (int)PropertyId.pidDirPath;
56         propstr.Name = "Path";
57         propstr.Description = "";
58         propstr.SettingName = "DirectoryPath";
59         propstr.DefaultValue = "C:\\";
60         properties.Add(propstr);
61
62         EnableDisableProperties(moduleConfiguration);
63     }
64     return properties;
65 }
66

```

The following kinds of properties are supported; String, StringEncrypted, Integer, Boolean, DateTime, Double, Enum, Color, Rectangle and File.

If properties need to be disabled based on some logic, EnableDisableProperties should be implemented. Such an example exists in the CSharp File Input module where poll interval is disabled if polling is disabled.

Settings are automatically saved using the SettingName supplied with the property, however if special handling is required then SetPropertyValues can be overridden for the loading of settings and the different SetProperty handlers for each type can be overridden for saving. Remarkable example exists in CSharp File Input module.

Some of the pages in the Developer are able to show Details and Default Destination. To add this support to a module the following two functions must be overridden;

```

319:         public override String GetDetails(ConfigurationObject moduleConfiguration)
320:         {
321:             return moduleConfiguration.GetString("DirectoryPath", "");
322:         }
323:
324:         public override String GetDefaultDestination(ConfigurationObject moduleConfiguration)
325:         {
326:             return moduleConfiguration.GetString("Default", "");
327:         }

```

JobModule

The code actually reading the files must be implemented in a JobModule. OnStartInputProcessing is called when the module is booted and OnStopInputProcessing is called when it is shutdown. These methods are only called once per configuration and not per instance of a module. This means that threads must be started per instance of the module passing info about the configuration of the instance. It is a requirement that OnStartInputProcessing returns, so spawning a thread is necessary.

```

17:         public override void OnStartInputProcessing()
18:         {
19:             String[] modules = GetModuleNames();
20:             foreach (String module in modules)
21:                 ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadJob), GetModuleConfiguration(module));
22:         }

```

In the code above all instances of our File Input module are returned in a string array. For each of them we spawn a thread where we tell it to perform the code in ThreadJob method, passing the configuration object for the module as a parameter.

In the thread code (ThreadJob method), files are read from disk in the path specified in the configuration. Data is put in a new Job. JobInfo's are added to the Job. A JobContext is created which is basically a group of Jobs (in case more than one file was read). The Job is added to the JobContext. The Job is then passed on to all destinations.

Files must be renamed, moved or deleted to prevent the module from picking up the file over and over. A hash of the filenames could be stored in a database, if none of the above options are possible.

OnStopInputProcessing must wait for all threads to stop before returning.

An example of the whole JobModule can be found in the sample folder.

3.2.2 File Output Module

An output module receives data in a `JobContext` in the `OnJobsReceived` handler. Based on the successful delivery of the data to the output destination, the jobs are committed or failed. Jobs are not passed on since output modules have no destinations.

Example of how all the jobs in the `JobContext` are delivered and committed:

```
16 public override void OnJobsReceived(JobModuleConfiguration configuration, JobContext jobContext)
17 {
18     for (int i = 0; i < jobContext.GetCount(); i++)
19     {
20         Job job = jobContext.GetJob(i);
21
22         FileStream fs = new FileStream(configuration.GetString("DirectoryPath") + "\\\" + job.GetJobInfo("FileName"), FileMode.Create, FileAccess.Write);
23         fs.Write(job.GetJobData(), 0, job.GetJobData().Length);
24         fs.Close();
25
26         jobContext.CommitJob(configuration, job);
27     }
28 }
```

In our example we use a `JobInfo` for the actual file name of the outputted files. In our File Input module example, we set this same `JobInfo` with the actual filename of the input file. Therefore, if you use these examples make sure you do not use the same input and output folder.

`CanActAsOutput` must return true;

3.2.3 Pass-Through Engine

An engine receives jobs just like an output module. It can both modify the data and pass it on to its destinations or it can create any number of new jobs and commit the original job.

`CanActAsEngine` must return true;

3.2.4 Base64 Modifier

A simple modifier which either Base64 encodes `JobData` or Base64 decodes `JobData`. A modifier cannot create other jobs like a module can.

3.3 Logging

Logs can be written via the method `LogEvent`. Different `EventTypes` can be specified. When spawning threads, it's necessary to specify a `ThreadModuleID`. This is done by calling `SetThreadModuleID` before any `LogEvent` calls, passing the Name of the instance of the module (`ConfigurationObject.GetName`).

Example of the thread code in the File Input module using `SetThreadModuleID` and logging the input directory path:

```
29| void ThreadJob(object configuration)
30| {
31|     ConfigurationObject config = (ConfigurationObject)configuration;
32|
33|     SetThreadModuleID(config.GetName());
34|
35|     LogEvent(EventType.Debug, "Looking for data every 500 msec in: " + config.GetString("DirectoryPath"));
36| }
```

3.4 Licensing

If your module needs specific licensing you can override `IsLicensed` in `JobModule` and implement your own logic. However, this does not overrule the fact that you need a Module Developer license to be able to use the module beyond the trial period.

3.5 Third Party Module

3.5.1 Information needed in module DLL for third party module

In the properties of the DLL fill out information for:

- | | |
|------------------|--|
| Product name: | Name of product. The first part of the module name must include the name of your company, like Microsoft Fax module. |
| Product version: | Version number for product. |