Emmanuel Istace's Blog

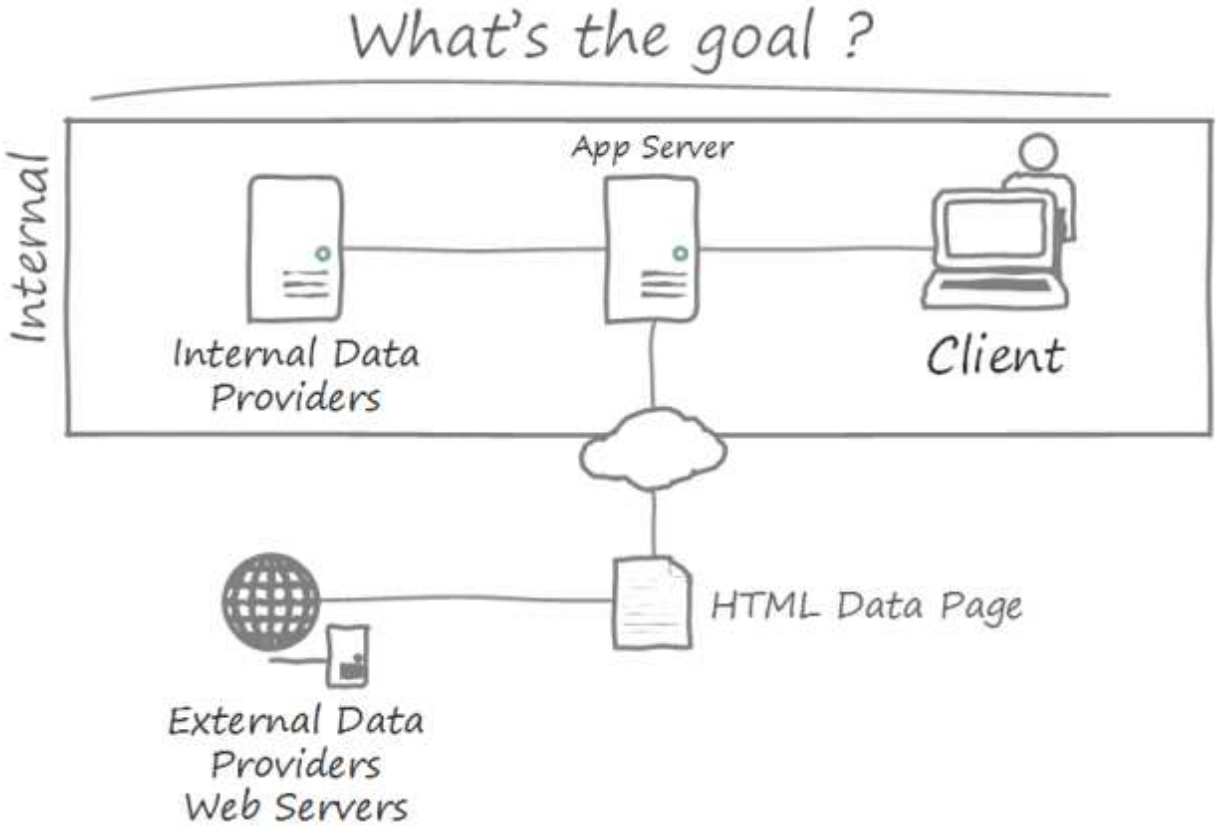
.Net... What else ?

Very simple plugin mechanism in C#

Once again I had to face the famous content aggregator problem. I already have done that on several project so I decided to write an article on that for the ones who wonders how to accomplish things like that.

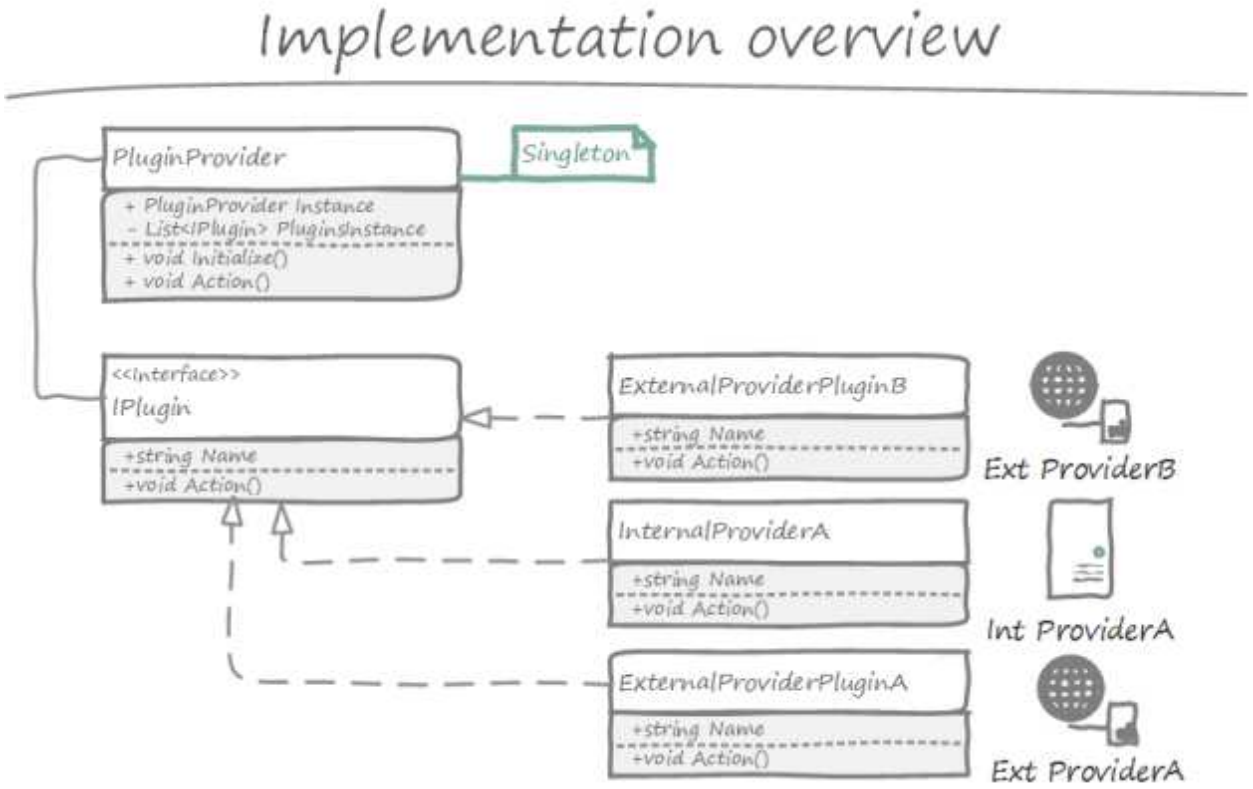
Background

So, the situation is : I have a system, who consume several services, aggregate the results and provide the output to the user. I also want to be able to develop each providers class as separated assemblies (plugins) so the providers can develop their own plugins and distribute them to the software users without having to recompile the application, just by dragging the assembly within a folder. Here's a diagram that resume the situation.

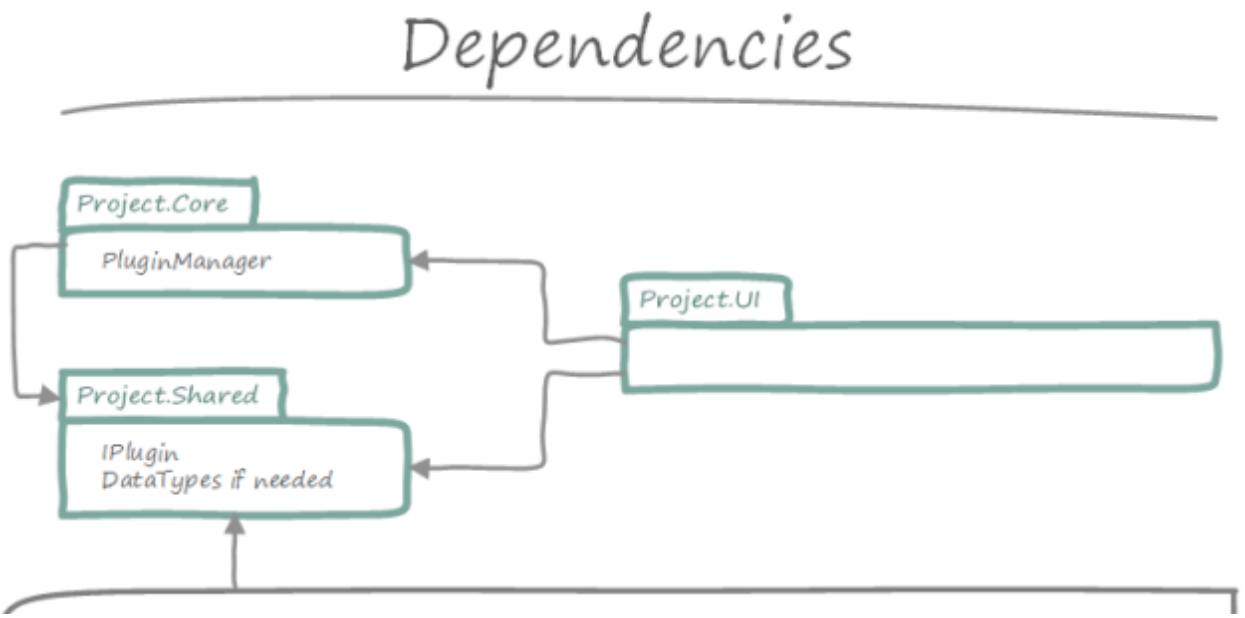


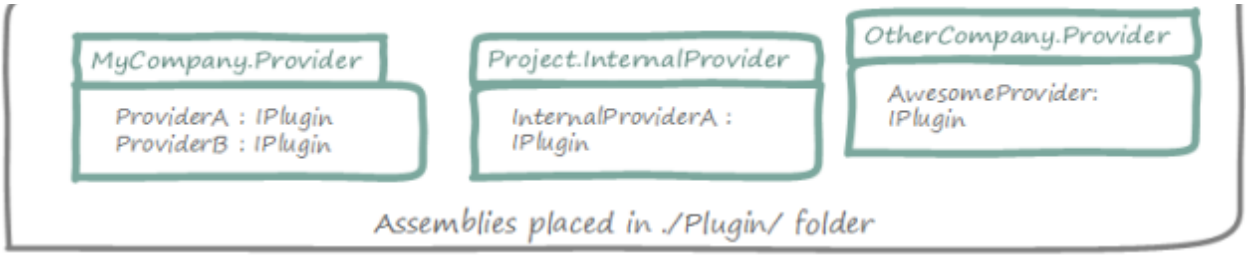
Technical Analysis

Regarding the requirements, some answers can seem obvious. First, we need to provide an interface to define what's a plugin. Then, we need something to control the plugin creation and access. This component will also be responsible in this case for the plugin call. Each providers will have his own implementation of the IPlugin interface. In this case, the manager called PluginProvider, will be a singleton instance and instantiate the plugins on creation. Here's the basic UML diagram of the system with some example providers.



Then, we have a dependency tree like this one





The assemblies that contains plugins will be placed in a folder where the application will look at startup. They just need to know the interface to implement and the data types exchanged if required. The UI layer refer the core, that contain the PluginManager and the shared assembly. No references to the plug-ins are created.

Implementation

The implementation is pretty simple. Here's our plugin interface :

```
1  public interface IPlugin
2  {
3      string Name { get; }
4      void Action();
5  }
```

Then we can implement a basic plugin like this :

```
1  public sealed class ExternalPlugin : IPlugin
2  {
3      public string Name
4      {
5          get { return "Exemple Plugin"; }
6      }
7
8      public void Action()
9      {
10         DoSomething();
11     }
12 }
```

And finally, the most important thing, the PluginManager.

```
1  public sealed class PluginProvider
2  {
3      private static PluginProvider instance;
4      /// Warning ! Not a thread safe implementation, change it if needed
5      public static PluginProvider Instance { get { return instance ?? (instance = n
6
7      private List pluginInstances;
8
9      private PluginProvider()
10     {
11
12     }
13
14     public void Initialize()
15     {
16         pluginInstances = pluginInstances ??
17                (from file in Directory.GetFiles(Path.Combine(AppDomain.CurrentDoma
18                 from type in Assembly.LoadFrom(file).GetTypes()
19                 where type.GetInterfaces().Contains(typeof(IPlugin))
20                 select (IPlugin)Activator.CreateInstance(type)).ToList();
```

```
21      }
22
23      public void Action()
24      {
25          if(pluginInstances != null)
26              pluginInstances.ForEach(plugin => plugin.Action());
27      }
28  }
```

So, first we have a basic not thread safe singleton implementation. When we call the Initialize() method, the Linq query will list all the dll in the Plugin application folder, load each assemblies, find the types that implement IPlugin and then create an instance of them. All the instances are then stored in the pluginInstances private list. When we call PluginProvider.Instance.Action(), we iterate over each instance and call the Action() method. If the method return something, we can easily adapt the code to aggregate the call returns and return it to the PluginProvider caller. And that's pretty much it !

Conclusion

This architecture can then be improved with for example repositories for query caching, using code contract to improve strength, using a thread safe singleton implementation, using a facade to directly call Providers.Action() or make this generic to be able to work with any interface which inherit from IPlugin... With all these improvements you'll start to have a kind of little plugin framework in our personal toolbox. Modularity is usually a key factor in software development and that kind of pattern are really usefull for a lot of scenario.

If you have any questions, feel free to ask in the comments below 🙂

See you next time and keep it bug free !

This entry was posted in DotNet, Programmation and tagged .net, architecture, c#, dotnet, pattern, plugin on 09/09/2013 [https://istacee.wordpress.com/2013/09/09/very-simple-plugin-mechanism-in-c/] .

---

2 thoughts on "Very simple plugin mechanism in C#"

Pingback: Reading Notes 2013-09-16 | Matricis

Emmanuel Istace Post author
16/09/2013 at 22:49

0 0 Rate This
Thank you for the link sharing 🙂