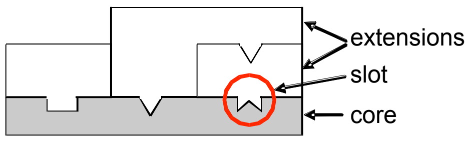
Plux.NET
A Platform for Building Plug-in Systems Under .NET
Quick Tutorial

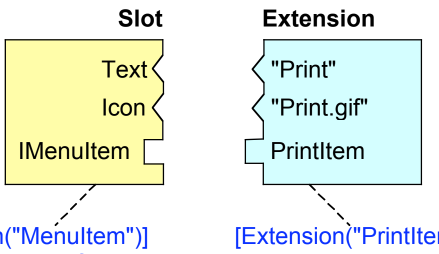
Plux.NET is a plug-in platform for .NET which allows you to build extensible applications consisting of an ultra-thin core and a set of extensions that can be plugged into designated slots of the core or other extensions at run time.



Plux.NET uses the metaphor of slots and extensions. A *slot* specifies a contract for extending a piece of software (called the *host*), and an *extension* is a plug-in component (i.e. a .NET assembly) that fills a slot. In essence, a slot declares the kind of information a host expects and extensions provide this information.

In its simplest form, a slot declares an interface as well as a list of parameters with their names and types. An extension provides a class implementing this interface as well as a list of values for the parameters. The host will rely on these parameters to load and integrate the extension. Slots and extensions are specified using .NET attributes.

Let's assume that our host is an application with a graphical user interface that allows menu commands to be installed as extensions. It opens a slot specifying an interface *IMenuItem* as well as two parameters: *Text* (for the name of the menu item) and *Icon* (for the icon to be used in the menu item). An extension is a class implementing *IMenuItem*. It also provides values for the parameters, e.g. "Print" for the *Text* parameter and "Print.gif" for the *Icon* parameter.



```
[SlotDefinition("MenuItem")]
[Param("Text", typeof(string))]
[Param("Icon", typeof(string))]
interface IMenuItem {
   ...
}
```

```
[Extension("PrintItem")]
[Plug("MenuItem")]
[ParamValue("Text", "Print")]
[ParamValue("Icon", "Print.gif")]
class PrintItem : IMenuItem {
   ...
}
```

For every open slot the platform detects available extensions, loads them, assigns the parameter values to the parameters and notifies the host that owns this slot. The host can then take actions for integrating the extension, e.g., by inserting a new menu item into its menus and linking it to the extension class.

A Simple Example Step by Step

Let's look at a simple example. Assume that we want to write an application that performs some actions and logs them. Since the logging should be kept flexible we do not implement it as part of the application but rather as an extension. For every action to be logged the application will pass to the extension a log message with a time stamp.

Step 1: Define a slot *Logger*

First we define a slot into which extensions can be plugged:

```
using Plux;

[SlotDefinition("Logger")]
[Param("TimeFormat", typeof(string))]
public interface ILogger {
   void Print(string msg);
}
```

A slot is an interface tagged with a *[SlotDefinition]* attribute specifying the name of the slot ("Logger"). A slot can have parameters defined by *[Param]* attributes. In our case we have just a single parameter *TimeFormat* of type *string*, which is to be filled by the extension. We compile this interface to an assembly *ILogger.dll*.

```
csc /t:library /out:ILogger.dll /r:Plux.dll ILogger.cs
```

Step 2: Write an extension for the *Logger* slot

Now we write an extension that fits into the *Logger* slot:

```
using System;
using Plux;

[Extension("ConsoleLogger")]
[Plug("Logger")]
[ParamValue("TimeFormat", "hh:mm:ss")]
public class ConsoleLogger: ILogger {
   public void Print(string msg) {
      Console.WriteLine(msg);
   }
}
```

An extension is a class tagged with an *[Extension]* attribute and implementing the interface of the corresponding slot. In our example the *[Extension]* attribute defines an extension *ConsoleLogger*. The *[Plug]* attribute defines a plug that fits into the *Logger* slot. The *[ParamValue]* attribute assigns the value "hh:mm:ss" to the parameter *TimeFormat*. We compile this class to an assembly *ConsoleLogger.dll*.

```
csc /t:library /out:ConsoleLogger.dll /r:Plux.dll,ILogger.dll ConsoleLogger.cs
```

Step 3: Open the *Logger* slot in the application

Our application runs under Plux.NET, so it has to be implemented as a plug-in itself extending the core. The core has a slot *Startup* into which our application should plug. So we need something like this:

```
using System;
using Plux;

[Extension("MyApp")]

[Plug("Startup")]
public class MyApp: IStartup {...}
```

The application has to open a *Logger* slot. This is done with an *[Slot]* attribute as shown below:

```
using System;
using System.Threading;
using Plux;

[Extension("MyApp")]
[Plug("Startup")]
[Slot("Logger", OnPlugged="AddLogger")]
public class MyApp: IStartup {

   ILogger logger = null; // the logger extension
   string  timeFormat;    // parameter of the logger extension

   public void Run() {
      string msg;
      while (true) {
         DoSomeAction(out msg);
         if (logger != null) {
            string time = DateTime.Now.ToString(timeFormat);
            logger.Print(time + ": " + msg);
         }
         Thread.Sleep(1500);
      }
   }

   public void AddLogger(object s, PlugEventArgs args) {
      logger = (ILogger) args.Extension;
      timeFormat = (string) args.GetParamValue("TimeFormat");
   }

   void DoSomeAction(out string msg) {
      msg = "Hello";
   }
}
```

The *[Slot]* attribute specifies that *MyApp* opens a *Logger* slot. Whenever the runtime finds an extension that fits into this slot it loads it and throws an *Plug* event, which causes *AddLogger* to be invoked.

AddLogger stores a reference to the attached extension in the field *logger*. It also retrieves the value of the *TimeFormat* parameter and stores it in the field *timeFormat*. In that way the extension is integrated with the host.

The *Run* method is called by the runtime core, because it is part of the *Startup* slot contract. It repeatedly performs some action and calls *logger.Print* (if a logger extension has been plugged in).

We compile this class to an assembly *MyApp.dll*.

```
csc /t:library /out:MyApp.dll /r:Plux.dll,ILogger.dll MyApp.cs
```

Step 4: Run the plug-in application

We have two plug-ins now (*MyApp.dll* and *ConsoleLogger.dll*) and one contract (*ILogger.dll*). All three assemblies are moved into a common directory together with the runtime core *Plux.dll* and the runtime starter *Plux.exe*. By default plug-ins and contracts that reside in the same directory as *Plux.exe* are discovered at startup. If your plug-ins reside in directories separate from *Plux.exe* use the command line arguments `/discovery` and `/base` (enter `plux /?` for help).

If we start *Plux* (enter `plux.exe`) it searches the application directory for an extension that fits into the *Startup* slot. It finds *MyApp.dll* and installs it. Since *MyApp* opens a *Logger* slot the runtime again searches for a matching extension. It finds *ConsoleLogger.dll* and plugs it in.

Hot Plugging

Adding plug-ins. Plux.NET supports *hot plugging*, i.e. extensions can not only be added to an application at startup but also at any later time without disrupting its execution. Hot plugging requires a plug-in such as *Cerberus.dll*, which is a discovery component that monitors the application directory to detect any additions or removals of plug-ins. Although the discovery can replaced by a custom discovery component later, *Cerberus.dll* has to be there in the beginning.

In order to demonstrate hot plugging, let us start *Plux* from a directory containing only *Cerberus.dll*. The discovery plug-in *Cerberus.dll* is plugged into a dedicated *Discovery* slot of the runtime. We don't see any effects yet, but the discovery plug-in now monitors the directory for additions. If we now move *MyApp.dll* into the directory it will be detected as a valid extension and plugged into the *Startup* slot of the core. *MyApp* is running now performing its actions but without logging them. If we move also *ConsoleLogger.dll* into the directory it will be plugged into the *Logger* slot of *MyApp*, causing logging to take effect.

Removing plug-ins. Plux.NET allows you not only to add extensions but also to remove them at run time. In order to do so we have to provide a handler for the *Unplug* event:

```
[Extension("MyApp")]
[Plug("Startup")]
[Slot("Logger", OnPlugged="AddLogger", OnUnplugged="RemoveLogger")]

public class MyApp: IStartup {
   ...
   public void RemoveLogger(object source, PlugEventArgs args) {
      logger = null;
      timeFormat = null;
   }
}
```

If we now remove *ConsoleLogger.dll* from the directory this will throw an *Unplug* event and *RemoveLogger* will be invoked. *MyApp* continues to run but it will not log its actions any more.