From Zero to Proficient with MEF

Tim Corey, 22 May 2012 CPOL

5.00 (166 votes) Rate:

Learn how to go from being an absolute beginner in the Managed Extensibility Framework to being an advanced user.

Download Example Application Source Code - 31.7 KB

Introduction

Why is it that when we look for a new laptop, we look at the types of ports it has? Not only do we put up with these holes in the sides of our new laptops, we complain if there aren't enough of them. The answer, of course, is that these ports allow us to extend our laptop. We can add a second monitor, an external hard drive, or a number of other devices. This doesn't mean that the original laptop is inferior; it just means that different use cases lend themselves to different configurations. So why is it that we insist on making applications without "ports"?

In .NET 4.0, Microsoft provided us with the Managed Extensibility Framework (MEF). This framework allows us to easily create extensibility points (ports) in our applications. The most obvious use for this is for plug-ins. You could allow a customer to create their own menu items just like they can do in Microsoft Word or Visual Studio. However, there are other uses for MEF as well. For example, if you expect business rules might be changed or expanded in the future (that never happens, right?), you could use MEF to make this process simple. I'll show you how below.

Why Should I Care About MEF

MEF has a lot of similarities to an IoC, but there are some differences. MEF concentrates on bringing parts into your application, even if the implementations are not known or local, whereas IoC is more about loosely coupling known parts. That still leaves the question of when to use MEF. The simplest explanation would be to say that it is for when you want to allow plugins for your application. However, I think there are a lot more reasons to use MEF. As I see it, the major reasons to use MEF are:

- To enable users to develop their own plugins for your application.
- To allow others to extend or modify how your application works without getting access or changing the source.
- Easy testability (just change where the DLLs are coming from and you could use your app against an entire set of mocks).
- To loosely couple library projects to your application, especially when they are used by multiple projects.

This last reason is a fairly broad one. Let me give you an example. Say you create a standard way to access your database in a safe manner. Maybe you have some custom data access business logic. In the end, you have an Interface that exposes two methods – `ReadData` and `WriteData`. You want use this DLL in all of your applications that you build internally. You could put the DLL in the GAC of every server or you could attach it to each project. Either solution has its merits but now you have a third option. You could store it centrally and load it via MEF. Then you could easily change it whenever you needed to without a big deployment issue (as long as you didn't change the interface, which you should never do).

The bottom line is that there are a lot of reasons to use MEF. As you get comfortable with MEF, you will start to see more and more places in your applications that could be made even better. This is one tool you really want in your toolbox.

Target Audience

When I first started learning how to use MEF, there were a number of resources that helped me. I found practical articles that showed me how to use the framework in a real application and I found technical articles that explained certain features in great length. What I never seemed to find was that one article that was both practical and in-depth. I ended up going from source to source, picking bits and pieces from each.

This article is intended to be that in-depth look at what MEF is in a practical manner. I will attempt to walk you through MEF from start to finish. Along the way I will build example applications so you can see exactly how each feature works. For those of you who are familiar with MEF, this article is broken up by feature so that you can quickly get to just the area you need help with.

Example Application

I have developed an example application that goes over each of the topics below. I have tried to keep the examples simple and easy to follow, yet practical. In the attached code section, you will find a solution that contains eight projects. At the beginning of every section, I will tell you which project to find the examples for that section. Each project also has text at the beginning that tells you what sections it covers. Finally, the parts of the code that deal with a particular section are also labeled with the section name. For the project Example05 (dealing with external libraries), I created two additional projects (Common and ExternalLibrary).

The Theory

MEF is used to design “ports” in your application. It can create both single-use ports (think VGA port) as well as multi-use ports (think USB ports). It also specifies how to create the “plugs” that go into those ports. It sounds complicated but it is really not much more than properly using interfaces and a little markup. I could go into a long-winded explanation of how it is designed and what it can do, but that can be hard to follow. Instead, let’s dive right into how to use MEF and then we can come back to the theory.

The Basics

There are three basic parts to MEF in your application. If we continue on with our laptop analogy, we can visualize these three parts as the USB port on the laptop, an external hard drive with a USB connector, and the hand that plugs the USB connector into the port. In MEF terms, the port is defined as an **[Import]** statement. This statement is placed above a property to tell the system that something gets plugged in here. The USB cable is defined as an **[Export]** statement. This statement is placed above a class, method, or property to indicate that this is an item to be plugged in somewhere. An application could (and probably will) have a lot of these exports and imports. The job of the third piece is to figure out which ports we have and thus which cables we need to plug into them. This is the job of the **CompositionContainer**. It acts like the hand, plugging in the cables that match up to the appropriate ports. Those cables that don’t match up with a port are ignored.

Simple Example

Example Project Name: Example01

Let’s look at a simple example. If you have done any research into MEF, you have probably seen this type of example before but we need to crawl before we can walk. This application will have a `string` that imports a value at runtime. We will export a message to be put into the `string` and then we will display the string to show it works.

To follow along with this example, simply start a new C# project of type “Console Application” in Visual Studio. When the project is created, you will have a “Program.cs” file. Inside that file, you should see the following code:

Hide Copy Code

```
static void Main(string[] args)
{
}
```

Since this is a static application, we need to create an instance of it instead so that we can play with MEF. Create a new void method called Run and then add code inside the Main method to instantiate the program and call the Run method. Your code should now look like so:

Hide Copy Code

```
static void Main(string[] args)
{
    Program p = new Program();
    p.Run();
}

void Run()
{
}
```

So now we have the plumbing set up. There is nothing MEF-specific in these lines, so I wanted to be sure that this code gets separated from our MEF code for the sake of clarity. Now let's get to the steps needed to get a basic MEF implementation working.

Step 1: You will need to add a reference to `System.ComponentModel.Composition` in your application. This is where MEF lives. This is dependent on .Net 4.0 framework. In theory you can download it from Codeplex and get it to (mostly) run on .Net 3.5 but that isn't supported or recommended.

Step 2: Add a using statement in your Program.cs file that references `System.ComponentModel.Composition`. This will allow us to use the Import and Export statements directly. It will also allow us to call the "ComposeParts" method on our container later (for now just remember this information – it will make sense later on).

Step 3: Add a class-level variable of type string called `message`. On top of this variable, add a `[Import]` statement. Your code should look like this:

Hide Copy Code

```
[Import]
string message;
```

This is our "port". We are asking our hand to find us a plug that fits into a string variable.

Step 4: Add a new class outside the Program class (it can be in a different cs file or in the same one – I put ours in the same file just for simplicity). Call it `MessageBox`. Inside the class, add a string property called `MyMessage` that returns an example message when the get is called. On top of the property, add `[Export]`. Your class should look like this:

Hide Copy Code

```
 public class MessageBox
{
    [Export()]
    public string MyMessage
    {
        get { return "This is my example message."; }
    }
}
```

The `MyMessage` property is our "plug" that will be put into the string "port" we defined above. We have defined the port and the plug, but we aren't done yet. If we stopped now, the application would run but nothing would happen. The reason is that we have not yet defined the method that will put the plugs into the ports. That comes next.

Step 5: This is the step where people new to MEF can get lost because there are actually a couple steps in one. I'm going to walk you through each part of this step so you know exactly what it going on. In the end, we will put all of the code from this step into one helper method called `Compose`. This will make it easy for our application to hook everything up in a clean manner. To make this easier, we should also add a using statement for `System.ComponentModel.Composition.Hosting`. This will make the code to follow more concise.

First, we need to create a catalog of all of the exports and imports. This will simply be an inventory of what is being exported and what needs to be imported. The catalog will need to know where to look. In our case, we will point to the executing assembly. This tells the catalog to inventory our current application. This may seem redundant but don't forget that MEF is designed to load external resources as plug-ins. In a more complex application, you would need to create multiple catalogs. For now, though, we will just create the one catalog like so:

Hide Copy Code

```
AssemblyCatalog catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
```

Next, we need to put this catalog into a `CompositionContainer`. This will give us the methods we need to compose (hook up) our exports and imports from our catalog. The code looks like this:

Hide Copy Code

```
CompositionContainer container = new CompositionContainer(catalog);
```

Finally, we need to actually hook everything up. This can be accomplished in this case by telling our new container to either compose the parts (`ComposeParts`) or satisfy the imports once (`SatisfyImportsOnce`). In either case, we need to pass in the instance that needs to have the imports satisfied on (where the ports are that need to be plugged in). We will do that by passing in the `this` keyword, which tells the method to look at our current instance of Program for the imports statements. Because we are using such a simple example, either of the composition methods listed would work. The difference is that the `SatisfyImportsOnce` statement will only hook up the parts once. That means that any changes to our parts catalog (either adding parts or removing them) won't be put into our application automatically. We will get into this more advanced topic later. For now, we are going to use the simplest implementation like so:

Hide Copy Code

```
container.SatisfyImportsOnce(this);
```

So now we have our helper method. If you have been following along, you should have a method that looks like this:

Hide Copy Code

```
private void Compose()
{
    AssemblyCatalog catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
    CompositionContainer container = new CompositionContainer(catalog);
    container.SatisfyImportsOnce(this);
}
```

Step 6: The final thing we need to do to get all of this to work is to get up our `Run` method to call the `Compose` method and then display the value of our message variable. Because this is a console application, I will also put a `Console.ReadLine` at the end of the method so that the application doesn't close until we hit a key. Here is the completed Run method:

Hide Copy Code

```
void Run()
{
    Compose();
    Console.WriteLine(message);
    Console.ReadKey();
}
```

That is all it takes to create a simple MEF application. If you run this application right now, you will see the message “This is my example message.” on the screen. If you got something different than I did or if you want to just see the end results in a working file, see Example 01 in my attached code.

To summarize, we created a new console application that had one “port” of type `string` called message. We designed a “plug” of type `string` that returned the message “This is my example message.” We then set up a container that took our catalog of parts and hooked them up (once). This allowed us to display the value of our message variable, which was populated with the value pulled from our “plug”.

Generated Questions

There are a lot of simple MEF demos that wrap things up at this point, but for me this demo generated a lot of questions so before we proceed, let’s answer a few of these questions.

1. **When I perform the `SatisfyImportsOnce` or `ComposeParts` method, am I creating an instance of every item I need to import right away?** Yes.
2. **Funny. Care to expound?** In most cases, creating an instance of every import right away will be fine. However, in the event that you might not use all of the instances or you want to load them later on, you can use lazy load on your imports. We will get into that below.
3. **If I export a class and then import it in two locations, do I get two instances of the class?** By default, MEF treats each export as a `Singleton`… well, sort of. If all the defaults are on, you get a singleton. See the “Object Instancing” section for a more detailed description of how the defaults work.
4. **Can I create “regular” instances of classes that are marked for export?** Yes, you can. Just note that any import statements that are inside the class will not be satisfied since the instance was created after MEF satisfied the imports. You could change this by passing in the instance reference instead of the “this” keyword in the line `container.SatisfyImportsOnce(this);`
5. **If I’m using MEF, should I use it everywhere in my application?** If you had a hammer, would you use that for all of your home repair needs? If so, you probably aren’t any better at home repair than I am. To directly answer your question, no, MEF is just one tool you can use in your application. It is designed to handle one set of requirements (plug-ins) very well. It can do other things (IoC/DI) but that isn’t what it was designed to do. MEF, like a hammer, can do a lot but sometimes it is better to use a different tool.
6. **What do I do if I want a bunch of one object?** There are a couple ways to handle this, depending on what you want. We could do different interfaces (if you wanted the same object to be used differently in different parts of your application), we could import it into multiple variables (if we wanted to use the same object in multiple locations in our code) or we could use an ImportMany (if we wanted multiple different objects with the same signature). We will see all of these solutions below.

Allowing null Imports

Example Project Name: Example02

Sometimes you may want to have an import that might or might not have a matching export. In the example of our laptop, we wouldn't want the laptop to violently blow up if we didn't have every USB port filled. Yet, if we don't change any of the defaults, that is exactly what our application will do if we don't fill all of our import statements. We can have as many non-matching exports as we want, but the first import statement that doesn't have a matching export will throw an exception. If this isn't what you want, there is a way to change the default behavior. On an import statement, you can put named parameters at the end to modify how the import works. One of these parameters is called `AllowDefaults`. If you set this to true, MEF will attempt to satisfy the import statement, but failing that, it will put a null value in the variable instead. The import would look like this:

Hide Copy Code

```
[Import(AllowDefault = true)]
string message;
```

In your code, make sure you then test the variable to be sure it isn't null before you use it. Otherwise, you will have just shifted your application-killing exception to a little later in your application's life.

Object Instancing

Example Project Name: Example02

By default, as we said before, MEF creates each export as a `Singleton`. This is technically true and a lot of descriptions stop there. However, the truth is a bit more complicated. In MEF, there are three `PartCreationPolicy` values that can be specified. The three values are `Shared`, `NonShared`, and `Any`. By default, `Any` is what is used on all exports. If the Import statement does not specify the type of `PartCreationPolicy` it is looking for, the `Any` policy will translate to `Shared`. A `Shared` policy is a `Singleton` and, of course, `Non-Shared` indicates that a new instance will be created for the export each time it is imported. Confused yet? How about some code to make this a little easier to understand? I'll put an export and an import statement and then explain what would happen.

Example 1: First, we will look at the simplest example just to get our feet wet:

Hide Copy Code

```
[Export, PartCreationPolicy(CreationPolicy.Any)]
public class MyClass {…}

[Import]
MyClass _port1;

[Import]
MyClass _port2;
```

In this example, the `_port1` and `_port2` variables will get the same `Singleton` instance of the `MyClass` object. The export statement is functionally equivalent to just `[Export]`. I just explicitly set the `PartCreationPolicy` to show you what was happening by default.

Example 2: Next, let's look at how we could change the export statement to create an instance each time we import it:

Hide Copy Code

```
[Export, PartCreationPolicy(CreationPolicy.NonShared)]
public class MyClass {…}

[Import]
MyClass _port1;

[Import]
MyClass _port2;
```

This time, _port1 and _port2 will get separate instances of the MyClass object. The creation of the instances is done by reflection, which can be a bit costly. There are ways to reduce this cost, but they are outside the scope of this already large article.

Example 3: I've already hinted above that you can specify what type of PartCreationPolicy an import will accept. Let's see how to do that:

Hide Copy Code

```
[Export, PartCreationPolicy(CreationPolicy.Shared)]
public class MyClass {…}

[Import(RequiredCreationPolicy = CreationPolicy.Shared)]
MyClass _port1;

[Import]
MyClass _port2;
```

In this case, the _port1 variable is requiring that the export be Shared. Because our export is Shared, _port1 will get the Singleton instance of the MyClass object like normal. The _port2 variable will also get the same Singleton instance. If we changed our export to be NonShared however, we would get an error on the _port1 variable because no exports would match the required import.

Example 4: This brings us to the question we asked originally about the default export being a Singleton but that the import also had to be default for that to be true. Let's look at an interesting example below:

Hide Copy Code

```
[Export, PartCreationPolicy(CreationPolicy.Any)]
public class MyClass {…}

[Import(RequiredCreationPolicy = CreationPolicy.Shared)]
MyClass _port1;

[Import(RequiredCreationPolicy = CreationPolicy.NonShared)]
MyClass _port2;
```

I bet you are wondering what happens here. I specified the Any on the export again just so you could see it. We could have just said [Export]. When the _port1 variable asks for an object with a Shared part creation policy, it gets a Singleton instance of the MyClass object. However, when the _port2 variable asks for an object with a NonShared part creation policy, it gets a new instance of the MyClass variable. By specifying a PartCreationPolicy of Any or by not specifying any policy, the export becomes whatever the import wants it to be. If nothing is specified on the import, you will get a Singleton. Make sure you understand this. This is where you could run into trouble down the road if someone decides to specify the RequiredCreationPolicy of NonShared on an import when you were expecting your export to always be a Singleton, even though you only labeled it as [Export].

Specifying Export Types

Example Project Name: Example02

When you mark an object as an export, by default MEF looks at the type of object it is and matches it up to an import that needs that type. Thus when we export a string, it gets matched up to an import of type string. However, this becomes problematic in anything other than an example application, since you may want to export multiple objects of type string. If there are multiple exports that could satisfy an import, MEF will throw an error because it doesn't know which export to use (see Gotchas below). There is a way to put multiple export instances into one import by using ImportMany (see below for more information on this topic) but most likely we want to match up specific exports to specific imports. The way you do this is by describing this export to MEF. You can do that in a number of ways. The first way is to specify a contract name. This can be any name you choose. Here is an example:

Hide Copy Code

```
[Export("MyConfigInfo")]
public class MyClass  {…}

[Import("MyConfigInfo")]
object _port1;
```

There are a couple things to note here. First, I used a magic string to identify my export. I needed to use the same magic string to identify the import. If I didn't use the "MyConfigInfo" on the import statement, MEF would have thrown an error saying it couldn't find the export I was looking for (see Gotchas). Second, I changed the type for _port1 to show that the types don't need to match anymore. Since we are explicitly saying how the import and export should know each other, the type of the variable doesn't matter. That doesn't mean you can put a string into an int variable. That still throws an error (can I get a collective "duh" here?). I recommend against using a contract name to identify an export. The reason being that you are reliant on typing everything correctly and you cannot use Intellisense. However, that is just a general rule of thumb I follow. There are cases where it is the best solution. As always, do what is best for your environment.

We can also identify objects by type in our Export/Import. This way does make use of Intellisense and it can be a lot cleaner (in my opinion, but feel free to harass me mercilessly below should you disagree). When you are exporting class objects, you can make use of Interfaces. Here is an example:

Hide Copy Code

```
[Export(typeof(IConfigInfo))]
public class MyClass : IConfigInfo {…}

[Import(typeof(IConfigInfo))]
object _port1;
```

That is pretty straightforward. Basically, if you are familiar with programming to an interface, you are already set. In your export statement, just specify the type you want to identify the export by and then do the same for the import.

If you really want to get specific, you can specify both a contract name and a contract type in your Export/Import.

Basically it takes the two scenarios above and combines them. The end result would look something like this:

Hide Copy Code

```
[Export("MyInfo", typeof(IConfigInfo))]
public class MyClass : IConfigInfo {…}

[Import("MyInfo", typeof(IConfigInfo))]
object _port1;
```

One question that might occur to you is what will happen if you export both the contract name and contract type but only import based upon one of the criteria. The answer is that it won't work. If you are specific in the export, you have to have an exactly matching specificity in the import. The one exception to this is when you don't specify the type in either the export or the import. Since MEF actually implies the type, you can be implicit in one and explicit in the other and the connection will still work.

OK, make sense so far? Great, because we are going to make it more complicated. Up until now we have talked about exporting objects like classes and value types. There is one area we haven't touched on yet, and that is the area of methods. MEF allows us to export and import methods in the same way we do with entire classes. Just like with everything else, you can specify a contract name and/or contract type. One thing that is a little different here is that we can use a delegate like `Action` or `Func` to specify the contract type. Here is an example that uses both a contract name and contract type:

Hide Copy Code

```
[Import("ComplexRule", typeof(Func<string, string>))]
public Func<string, string> MyRule{ get; set; }

[Export("ComplexRule", typeof(Func<string, string>))]
public string ARule(string ruleText)
{
    return string.Format("You passed in {0}", ruleText);
}
```

To call the rule, you would do something like this:

Hide Copy Code

```
Console.WriteLine(MyRule("Hello World"));
```

I made sure this example was a bit more advanced so you could see exactly how this would work. If you haven't used delegates and anonymous types before, you might need to brush up on them before moving forward with this part of MEF. Trying to learn two technologies at once and using them together is a recipe for a debugging disaster.

So what we are doing here is exporting a method that has one argument (of type `string`) and an output of type `string`. We are importing it into a property and calling it as if it were a local method. We also applied a custom name to our method export so that we can differentiate it from other methods with a similar signature.

Inheriting an Export

Example Project Name: Example03

One feature that really makes MEF powerful is the idea of `InheritedExport`. With this feature, you can ensure than any class that inherits from the item marked with [InheritedExport] will be exported. This works just like an export statement, so you can have a contract name and/or contract type specified. Before we get into the benefits of this, let's look at an example:

Hide Copy Code

```
[InheritedExport(typeof(IRule))]
public interface IRule


public class RuleOne : IRule {…}
public class RuleTwo : IRule {…}
```

Even though RuleOne and RuleTwo don't have an explicit export statement, they will both be exported as IRule types (by the way, read that interface name over and over until you feel better about yourself – I'm not just a coder, I'm also a self-help coach). Now, look over that scenario for a minute and think about the possibilities. First, if you put your Interface in a different assembly, the assembly where RuleOne and RuleTwo were located would not need to know about MEF. You are also guaranteed that the export name and type are correct. Finally, you (mostly) ensure that part of the contract for implementing the interface is that the concrete class gets exported as a part. I say mostly because there is one keyword that can be used to block the export. You can use the [PartNotDiscoverable] tag on the top of your class to specify that this class should not be picked up by the MEF library. The class can still be added explicitly (we will get to that later) but it won't be picked up by the ComposeParts or SatisfyImportsOnce methods.

Importing Multiple Identical Exports

Example Project Name: Example03

If you were to test out the above example, you would probably run into an error that says "*More than one export was found that matches the constraint*" (see Gotchas). This is because you have one import statement and more than one export statement that matches your import. In terms of our analogy, we have two VGA cables but only one VGA port. Our hand doesn't know which one to plug in. Sometimes this is a mistake on our part. We define two parts to do the same job. Other times, however, we want this to happen. A common example would be a set of rules that needs to be run. It would be highly unlikely that you would have only one rule for any given scenario. Instead, you would probably have many rules. This is where the [ImportMany] statement comes in. It allows us to import zero or more Exported items that match. Did you catch that? **An ImportMany statement won't fail if you don' have any matching export statements**. It will, however, initialize the collection that you have the ImportMany statement on. That prevents errors where a collection wasn't initialized when it was called. Let's see how this works:

Hide Copy Code

```
[ImportMany(typeof(IRule))]
List<IRule> _rules;
```

This ImportMany statement will work with the above InheritedExport example. Now you can use a foreach on the _rules list and call each rule individually.

Delaying Instance Creation

Example Project Name: Example03

Earlier we talked about the fact that MEF creates instances of every needed import when `ComposeParts` or `SatisfyImportsOnce` is run. Normally this should be fine. Newly created instances of classes usually don't take up that much memory. However, there may be times when you want to delay the instantiation of an object or set of objects. To do this, simply use the `Lazy<T>` for delayed instantiation that Microsoft has already provided us. Nothing changes on the export, only the import needs to be changed like so:

Hide Copy Code

```
[Import]
Lazy<MyClass> myImport;
```

Note that this isn't really MEF technology, but rather MEF using `Lazy<T>` effectively. It isn't a MEF-specific implementation. Don't forget that when you lazy-load something, you need to access the actual instance you are looking for using the `Value` property. In our case, that would look like this:

Hide Copy Code

```
myImport.Value.MyMethod()
```

Using lazy-loading to delay instantiation of certain classes definitely has its place, especially if you are uncertain if all of your imported instances will be used or if certain imports are resource-intensive. Again, just because you found a new tool doesn't mean that it is the best tool for all jobs.

Describing the Export

Example Project Name: Example04

Now that we have learned about how we can use `ImportMany` to bring a bunch of matching exports into one array and how we can wait to instantiate our imports until we need them, the issue of export decoration comes up. How do we differentiate one export from another if the signatures match without peaking inside them? The answer is metatdata. We can attach metadata to our exports to describe information about the export to the importing application. For instance, say we were going to import a number of business rules. Each rule has the same signature, but deals with how to process a record in a different state. This might be an excellent opportunity to use lazy-loading because we most-likely will not use the rule for every state. It is also an excellent opportunity to use metadata, because we can describe which state each rule handles. Let's look at an example. Here is our exported class objects:

Hide Copy Code

```
public interface IStateRule
{
    string StateBird();
}

[Export(typeof(IStateRule))]
[ExportMetadata("StateName","Utah")]
[ExportMetadata("ActiveRule", true)]
public class UtahRule : IStateRule
{
    public string StateBird()
    {
        return "Common American Gull";
    }
}

[Export(typeof(IStateRule))]
[ExportMetadata("StateName", "Ohio")]
[ExportMetadata("ActiveRule", true)]
public class OhioRule : IStateRule
{
    public string StateBird()
    {
        return "Cardinal";
    }
}
```

I added two pieces of metadata to each export just to show you that you can stack these up. I will only do my filter based upon one piece of data (StateName) but I could filter these exports based upon which were active as well. That way we could have multiple exports for the same state. It would be even better if we put an activation date that we could filter on so we could always get the latest version yet the older versions could be retained in case we needed them to process old records, etc.

Here is how we would import those class objects:

Hide Copy Code

```
[ImportMany(typeof(IStateRule))]
IEnumerable<Lazy<IStateRule, IDictionary<string, object>>> _stateRules;
```

Notice that we are using lazy-loading, [ImportMany], and export metadata all against on variable. Whew. That is a lot of different items crashing together in two lines of code. If we hadn't already gone over each of these concepts, it would probably be very confusing but I'm sure it is simple to you now. Just in case it isn't, however, let me explain a bit. We are doing an ImportMany so we can bring in multiple items with the same signature. In our case, the signature is that the export will be of type IStateRule. We are putting this set into an IEnumerable so that we can cycle through the set of imported rules. Next, we are specifying that the items will be `Lazy<T>` so that they are not loaded right away but rather only when called. Finally, after the type of item to import is specified (`IStateRule`), we add a parameter to specify an IDictionary set. This is the metadata declaration.

Now that we have set everything up, here is how we would access the metadata. In this example, I am looking to only call the Utah rule:

Hide Copy Code

```
Lazy<IStateRule> _utah = _stateRules.Where(s => (string)s.Metadata["StateName"] ==
"Utah").FirstOrDefault();
if (_utah != null)
{
    Console.WriteLine(_utah.Value.StateBird());
}
```

The first line in the above part is a simple `Linq` query that gets me the first item that matches the criteria specified. In this case, I specified that the "StateName" value should be "Utah". That will put the `Lazy<IStateRule>` copy into my local variable `_utah` for use. I did not try to get the actual instance of `IStateRule` inside the `Linq` query because if the value isn't found, the system will throw an error.

Normally I try to keep examples as simple and focused as possible, so I avoid standard error handling and other plumbing. However, in this case I thought it wisest to include some basic error handling capabilities. Since we are using simple `string` keys in our `Dictionary`, there is a very real possibility that we might not have the correct pair listed in every export. What I did here was I made sure to use the FirstOrDefault method so that it won't fail if the query doesn't return any data. I also then check to be sure the _utah variable isn't null before I try to call it.

<<hand raises>>

I see that hand, but I alread know what you are going to ask: "**Why aren't you using [InheritedExport] in this example?**" I was hoping you wouldn't ask that, because I don't have a very satisfactory answer. Since you did, however, I'll give you what I have. When you use the [InheritedExport] tag on the IStateRule interface in this example, the metadata does not get exported with the exports. From what I can tell, this is because the metadata needs to be exported at the same time as the export itself and the [InheritedExport] tag seems to be processed at a different time. I have tried multiple tricks to get it to work to no avail. If you know how to get this to work (without a workaround), please let me know. I'll be sure to post your solution here and give you the credit.

Advanced Metadata

Example Project Name: Example04

To make metadata even more useful, instead of using the Dictionary<string, object>, you can specify your own class or interface. The easiest way to do this is on the import side of the application. You simply need to replace the Dictionary<string, object> with your interface or class name like so:

Hide Copy Code

```
[ImportMany(typeof(IStateRule))]
IEnumerable<Lazy<IStateRule, IStateRuleMetadata>> _stateRules;
```

My interface looks like this:

Hide Copy Code

```
public interface IStateRuleMetadata
{
    string StateName { get; }
}
```

One thing to note here is that I made my property read-only (only a getter). This is intentional. MEF will throw an error if you try to include a setter. There is no need for a setter, however, so this is fine. To access this metadata, you use the metadata dot property like so:

Hide Copy Code

```
Lazy<IStateRule> _utah = _stateRules.Where(s => (string)s.Metadata.StateName ==
"Utah").FirstOrDefault();
```

As you can see, my code on the import side no longer uses strings for the key names, but property names. This allows me to utilize Intellisense at design time, which makes my development easier and it makes it easier to avoid errors that will be difficult to debug.

I am sure you are wondering what has changed on the export side. The answer is that nothing has changed. You still specify the metadata properties using strings (no Intellisense). MEF handles how to hook these into your metadata class or interface at runtime. This is especially cool if you use an interface for your metadata like

I did, since your interface is never actually implemented in your code. MEF implements it for you.

Loading Other Assemblies

Example Project Name: Example05

So far we have concentrated on how to get the exports from our current assembly to match up with our imports. MEF provides us with a lot more options than this, which is important. Let's look at the different options we can use.

Assembly Catalog

This is the catalog we have been using up until now. You pass in a specific assembly and MEF handles the discovery of the parts inside that assembly. In our case, we have been passing a reference in to the current assembly (`Assembly.GetExecutingAssembly`).

Directory Catalog

This is the fun one. It takes in either a relative or absolute path to a directory. MEF will then scan that directory (but not the sub-directories) for assemblies that are decorated with MEF exports. It will add any parts it finds to the catalog. You can further refine this by adding a parameter that specifies the format for the name of the assembly. For example, you could put "*.dll" to include only dll files or you could put "CompanyName*.dll" to include only dlls that start with your company name. Here is an example of how to set up a directory catalog using a relative path and a filter:

Hide Copy Code

```
DirectoryCatalog catalog = new DirectoryCatalog("Plugins", "*.exe");
```

I decided to search for an exe file just to show you that it does not have to be a dll. Note that the folder name does not need to include any leading characters like slashes or dots.

Type Catalog

This is one I personally do not use. The purpose of this catalog is to only load exports of a specified type in the catalog. To specify the types, you need to add them directly to the constructor line. For sake of completeness, I will include an example of doing so here:

Hide Copy Code

```
TypeCatalog catalog = new TypeCatalog(typeof(IRule), typeof(IStateRule));
```

Notice that I added two types, but more could be added as needed.

Aggregate Catalog

When having one catalog isn't enough, you can create an aggregate catalog. This catalog is simply a catalog of catalogs. You can add as many catalogs as you want to this catalog. Simply pass in your catalogs to the constructor and you are set. Most people create their individual catalogs in the constructor line of the aggregate catalog for simplicity and so that they don't have to create individual variables for each catalog. I will show you this technique:

Hide Copy Code

```
AggregateCatalog catalog = new AggregateCatalog(new AssemblyCatalog(Assembly.GetExecutingAssembly()),
new DirectoryCatalog(@"C:\Plugins"));
```

As you can see, I am creating two different types of catalogs (`DirectoryCatalog` and `AssemblyCatalog`) inside the `AggregateCatalog`. This will allow my system to discover the internal parts as well as the parts from all of the assemblies inside the temp folder. Note that I used the string literal character (@) to prevent me from needing to escape the slashes inside my folder path.

Adding New Assemblies at Runtime

Example Project Name: Example05

Now that we know how to load all of the assemblies inside a folder with MEF, the idea of adding assemblies on the fly shouldn't be far behind. This will allow us to drop new dll files into our folder and have our currently-running application pick them up and use them.

To accomplish this, there are a few things we need to bring together. First, we are going to need to use the `DirectoryCatalog`. This is where the changes will be picked up (since we won't be changing our current assembly code on the fly). Next, we will need to create a variable to hold the directory catalog instead of creating a new instance inside the `AggregateCatalog` constructor. We will also need to be sure to scope this variable in such a way as to have access to it later on. Finally, we are going to need to have some sort of trigger to tell us when we should tell the `DirectoryCatalog` to update (and thus trigger a recomposition).

In a real-world application, you would most likely put a `FileSystemWatcher` on your plugins folder and have the events from that trigger the `DirectoryCatalog`. Doing this is outside the scope of this article. Instead, I will show you the code you need on the MEF side. How you trigger that code is up to you.

The actual implementation of recomposition is fairly simple. First, you need to be sure you use the `ComposeParts` method instead of the `SatisfyImportsOnce` method. Then you need to mark any `Import` or `ImportMany statement` that might be getting a new part with the parameter "`AllowRecomposition=true`". Then you either add something new to the `CompositionContainer` or you trigger a `Refresh()` on your `DirectoryCatalog`. The parts look like this:

Hide Copy Code

```
[Import(AllowDefault = true, AllowRecomposition=true)]
string test;

dirCatalog.Refresh();
```

I added the named parameter AllowDefault to my Import as well, just to demonstrate its use. This could also be useful if you are importing a single export and you are going to allow recomposition on that part. If the system doesn't detect a matching export right away, you don't want the system to blow up. By the time you get around to using that variable, the recomposition might have already happened and the part might be populated.

Note that when the parts are recomposed, if a part gets added to an ImportMany, all of the parts in that list get rebuilt. That is by design but it means that you need to understand how recomposition will affect your application and make development decisions based upon that fact. Also, the AppDomain will continue to hold a dll in memory until it is restarted. That means that you will need to use ShadowCopy to delete dlls currently in use. If you do, recomposition will remove these parts from your assembly (but still hold a copy of the dll in memory). Your application will work correctly but it isn't totally clean.

Other Features of MEF

We have touched on the mainline features of MEF, but there are still a lot of areas that we haven't covered. Some of these are only rarely used, but there are at least a few that I feel are more useful. Here are a few that I think would be most useful to you (in no particular order).

Constructor Injection

Example Project Name: Example06

MEF provides us with an option on how we bring in an export. Normally, we will decorate a variable with an [Import] tag. However, there is another option for populating the variable and that is via Constructor Injection. The simplest way to do this is to simply add the [ImportingConstructor] over your specialized constructor like so:

Hide Copy Code

```
public class MyClass
{
    [ImportingConstructor]
    public MyClass(IRule myRule)
    {
        _myRule = myRule;
    }
}
```

I created a specialized constructor for my class that took an interface as an argument. I then decorated the constructor with the ImportingConstructor tag to indicate that I wanted MEF to populate this constructor upon composition. MEF will infer which parts it needs just like it would on an [Import] statement with no modifiers. However, should you want to add modifiers to tell MEF what parts you want specifically, you can do so like this:

Hide Copy Code

```
public class MyClass
{
    [ImportingConstructor]
    public MyClass([Import(typeof(IRule))] IRule myRule)
    {
        _myRule = myRule;
    }
}
```

This makes the constructor a bit messier but you can do everything you can with an [Import] or [ImportMany] tag in your constructor. So now the question becomes "why?". I believe the best use case for this method is when you want to control what happens when a part is created. For example, you might want to initialize values or otherwise set up your system. For example, if you have a User class that imports a class instance that handles payroll information for the user, you might want to pass in the UserID to the payroll instance right away so it knows who it is working with.

Getting an Export Manually

Example Project Name: Example06

There may be times when you want to get an exported part manually instead of using an import statement. Here are three examples of how to do this:

Hide Copy Code

```
string manual1 = container.GetExportedValue<string>();
string manual2 = container.GetExportedValue<string>("ContainerName");
string manual3 = container.GetExportedValueOrDefault<string>();
```

The first example just finds a part of type string. The second finds a part of type string with the specified container name. The third shows how you can do the same thing using the GetExportedValueOrDefault method instead so that the system will not throw an exception if the part is not found.

Personally, I shy away from this way of getting values. It gets ugly quick and it makes you expose your container (in some method - don't just make it a global variable) to a wider audience than you otherwise would need to do.

Getting an Alert when MEF is Done

Example Project Name: Example06

There may be times when you need to know when the import process is complete. This is fairly simple to do. First, you need to implement the interface IPartImportsSatisfiedNotification. This interface provides you with the method OnImportsSatisfied(). This method will be fired off when all of the imports have been successfully hooked up.

Shutting Down MEF Early

Example Project Name: Example06

There may be times when you want to explicitly close out MEF and release the resources before the rest of your application is closed. To do this, call the Dispose() method on your container like so:

Hide Copy Code

```
container.Dispose();
```

Different parts will close out differently, but basically the parts held in the container will be properly closed out and disposed of when the container is disposed.

Gotchas

In the course of working with MEF, you will come across errors and problems. Here are a few of the more common issues and how to resolve them:

1. **"No valid exports were found that match the constraint"** – This error basically tells you the issue. MEF didn't find an export to hook into your import. Why this happens can be a bit baffling at times. First, check to see that you are exporting an item of the expected type. Then, make sure the contract name is exactly the same on both ends (if you are using one). Do the same with contract type. Finally, if all else fails and you are sure that everything matches, make sure you are looking in the given assembly for the export. If you are only looking in dll files and forgot to add the current assembly, you may be missing your export.
2. **"More than one export was found that matches the constraint"** – When you have an `Import` statement, you can only have one `Export` that matches. If MEF finds two, it doesn't know which one to use so it throws this error. Either use an `ImportMany` or modify one of the `Export` statements to have a different signature. This error is especially common when you are letting MEF assume the contract type. If you export two strings, you will get this error. Instead, try to develop everything towards an `Interface`.
3. **"The export is not assignable to type..."** – This means that you specified an `Import` type but you can't put that type into the variable. Just because you call an `int` a `string` doesn't make it so. The other possibility here is that you are trying to do an `ImportMany` but you accidentally put just `Import`. In that case, you would be trying to put one item into an item of type `IEnumerable`, which won't work (it tries to assign it instead of calling the `Add` method). The reverse is also true - if you put an `ImportMany` on a variable but the variable cannot accept a set of instances, you will get this error.

For more information on MEF errors, MSDN has some good information here: http://msdn.microsoft.com/en-us/library/ff603380.aspx

Ways to Use MEF

MEF can be (and is) used all over the .NET stack. It is common to see it used with WPF and Silverlight (look into Caliburn.Micro for an excellent implementation of MEF in conjunction with MVVM). You can, however, also use MEF with ASP.NET to make MVC more extensible. Basically, the technology will work with just about anything. It is more a matter of ensuring that the use case justifies the tool.

The Future of MEF

MEF has been around for a while. Currently the team is working on building a new version of MEF (currently referred to as MEF2 - clever, right?). You can find out more about what the latest build includes by reading the BCL Team blog.

Conclusion

In this article, we have started at the very beginning of the Managed Extensibility Framework and worked our way steadily through the major parts of MEF. We have covered the simple examples, but also the more realistic cases that we are likely to see in a production application. I hope that this has been helpful to you. Please let me know below if there is something that you found unclear. As always, I appreciate your constructive feedback.

History

- May 2, 2012 - Initial Version
- May 16, 2012 - Added Gotchas section
- May 19, 2012 - Minor bug fixes
- May 21, 2012 - Added the Why Should I Care About MEF section

License

Share

EMAIL

About the Author

Tim Corey

Software Developer (Senior) Epicross
United States

I am currently a Lead Technical Consultant for a consulting company called Epicross. My primary skills are in .NET, SQL, JavaScript, and other web technologies although I have worked with PowerShell, C, and Java as well.

In my previous positions, I have worked as a lead developer and IT Director. As such, I have been able to develop software on a number of different types of systems and I have learned how to correctly oversee the overall direction of technology for an organization. I've developed applications for everything from machine automation to complete ERP systems.

My current position is mainly focused making our clients more efficient and effective. I use custom software (desktop, mobile, and web) to help facilitate this goal. When I'm not working for the company, I'm usually developing applications to fill the needs of the organizations I volunteer for.

Follow on Twitter Google LinkedIn

Comments and Discussions

Add a Comment or Question Search Comments Go

First Prev Next

My vote of 5 new
JaredFox 31-Mar-15 21:23

Excellent!!! new
Member 11550421 30-Mar-15 16:50

Excellent!!! new
Member 11360660 14-Jan-15 14:30

Superb new
anshusudhanshu 26-Dec-14 19:10

Fantastic introduction new
Member 11198561 3-Nov-14 13:22

My vote of 5 [modified] new
JoseHenrique 22-Oct-14 18:52

Importing Complex datatypes like class object or structures new
CiaoWorld 15-Oct-14 20:52

MEF QUERY new
CiaoWorld 15-Oct-14 20:06

Re: MEF QUERY new
lagdaemon 17-Dec-14 21:18

Wonderful Article! new
AlexWang2010 24-Sep-14 0:36

Excellent new
GraemeKMiller 2-Sep-14 16:11

Just excellent!! new
Spyridon Stakkos 19-Jul-14 20:06

Re: Just excellent!! new
Member 10807175 4-Aug-14 20:41

My vote of 5 new
Noctis 12-Jul-14 14:18

Nice job new
Member 10236318 12-Jul-14 1:51

wow nice copy and paste article rifl dumb aho,e new
Member 10933788 10-Jul-14 19:27

Re: wow nice copy and paste article rifl dumb aho,e new
Tim Corey 10-Jul-14 19:38

Re: wow nice copy and paste article rifl dumb aho,e new
spigrig 23-Jan-15 20:53

Refreshin the current dll new
Member 10413463 10-Feb-14 19:46

Re: Refreshin the current dll new
Tim Corey 10-Feb-14 19:51

This article really rocked !! hi five!! new
Member 2207819 17-Jan-14 9:38

Best articel on Beginning MEF new
dleasman 4-Jan-14 1:01

My vote of 5 new
OrionGG 20-Nov-13 16:03

My vote of 5 new
M Rayhan 6-Nov-13 11:55

My vote of 5 new
Member 9526344 28-Sep-13 2:31

Refresh 1 2 3 4 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.