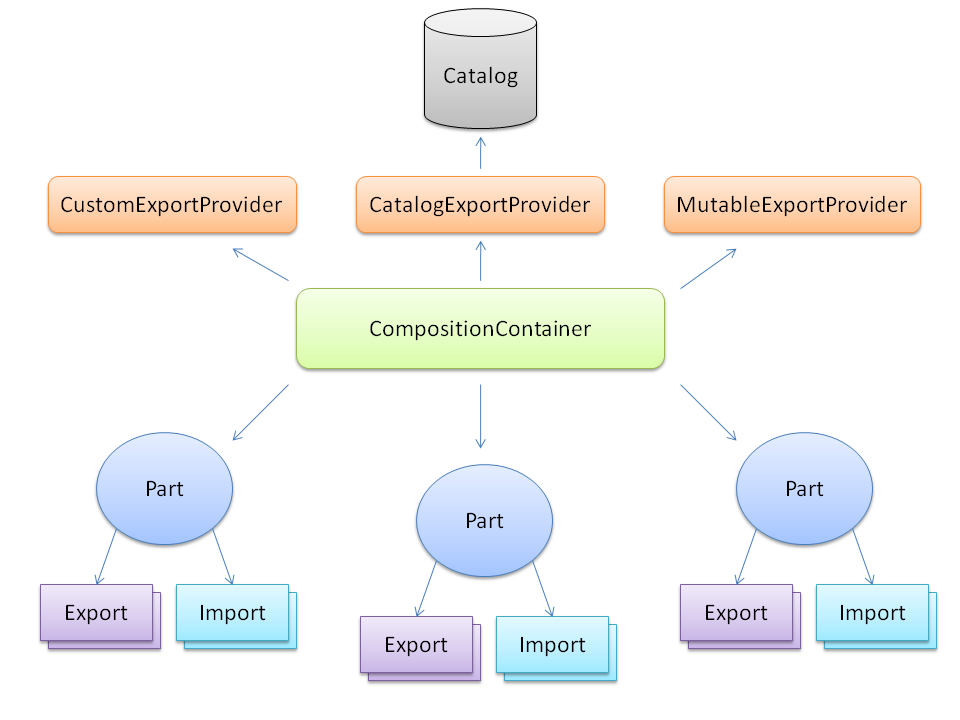
# Managed Extendibility Framework

* MEF provides a standard way for the host application to expose itself and consume external extensions. Extensions, by their nature, can be reused amongst different applications. However, an extension could still be implemented in a way that is application-specific. Extensions themselves can depend on one another and MEF will make sure they are wired together in the correct order (another thing you won't have to worry about).
* MEF offers a set of discovery approaches for your application to locate and load available extensions.
* MEF allows tagging extensions with additonal metadata which facilitates rich querying and filtering



[Export(typeof(IMessageSender))]

public class EmailSender : IMessageSender {

...

}

[Export(typeof(IMessageSender))]

public class TCPSender : IMessageSender {

...

}

public class Notifier {

[ImportMany]

public IEnumerable<IMessageSender> Senders {get; set;}

public void Notify(string message) {

foreach(IMessageSender sender in Senders)

sender.Send(message);

}

}

Or [Import]

A common pattern when building extensible applications with MEF is to deploy a contract assembly. A contract assembly is simply an assembly which contains contract types that extenders can use for extending your app. Commonly these will be interfaces, but they may be abstract classes. Additonally contract assemblies will likely contain metadata view interfaces that importers will use, as well as any custom MEF export attributes.

## Property exports

Parts can also export properties. Property exports are advantageous for several reasons.

* They allow exporting sealed types such as the core CLR types, or other third party types.
* They allow decoupling the export from how the export is created. For example exporting the existing HttpContext which the runtime creates for you.
* They allow having a family of related exports in the same Composable Part, such as a DefaultSendersRegistry Composable Part that exports a default set of senders as properties.

For example you might have a Configuration class that exports an integer with a "Timeout" contract as in the example below.

public class Configuration

{

[Export("Timeout")]

public int Timeout

{

get { return int.Parse(ConfigurationManager.AppSettings["Timeout"]); }

}

}

[Export]

public class UsesTimeout

{

[Import("Timeout")]

public int Timeout { get; set; }

}

## Method exports

A method export is where a Part exports one its methods. Methods are exported as delegates which are specified in the Export contract. Method exports have several benefits including the following.

* They allow finer grained control as to what is exported. For example, a rules engine might import a set of pluggable method exports.
* They shield the caller from any knowledge of the type.
* They can be generated through light code gen, which you cannot do with the other exports.

*Note: Method exports may have no more than 4 arguments due to a framework limitation.*

In the example below, the **MessageSender** class exports its **Send** method as an **Action<string>** delegate. The Processor imports the same delegate.

public class MessageSender

{

[Export(typeof(Action<string>))]

public void Send(string message)

{

Console.WriteLine(message);

}

}

[Export]

public class Processor

{

[Import(typeof(Action<string>))]

public Action<string> MessageSender { get; set; }

public void Send()

{

MessageSender("Processed");

}

}

You can also export and import methods by using a simple string contract.

## Inherited Exports

MEF supports the ability for a base class / interface to define exports which are automatically inherited by implementers. This is ideal for integration with legacy frameworks which want to take advantage of MEF for discovery but do not want to require modifying existing customer code. In order to provide this capability use the System.ComponentModel.Composition.InheritedExportAttribute. For example below ILogger has an InheritedExport. Logger implements ILogger thus it automatically exports ILogger.

[InheritedExport]

public interface ILogger {

void Log(string message);

}

public class Logger : ILogger {

public void Log(string message);

}

## Property Imports

To import a value to a property, decorate the property with the **[System.ComponentModel.Composition.ImportAttribute]**. For example the snippet below imports an IMessageSender

class Program

{

[Import]

public IMessageSender MessageSender { get; set; }

}

[ImportingConstructor]

## Optional imports

MEF allows you to specify that an import is optional. When you enable this, the container will provide an export if one is available otherwise it will set the import to Default(T). To make an import optional, set AllowDefault=true on the import as below.

[Export]

public class OrderController {

private ILogger \_logger;

[ImportingConstructor]

public OrderController([Import(AllowDefault=true)] ILogger logger) {

if(logger == null)

logger = new DefaultLogger();

\_logger = logger;

}

}

## **Importing collections**

In addition to single imports, you can import collections with the ImportMany attribute. This means that all instances of the specific contract will be imported from the container.

MEF parts can also support recomposition. This means that as new exports become available in the container, collections are automatically updated with the new set.

[ImportMany(AllowRecomposition=true)]

public IEnumerable<IMessageSender> Senders {get; set;}

## IPartImportsSatisfiedNotification

In some situations it may be important for your class to be notified when MEF is done with the import process for your class instance. If that's the case implement the **[System.ComponentModel.Composition.IPartImportsSatisfiedNotification]** interface. This interface has only a single method: OnImportsSatisfied, which is called when all imports that could be satisfied have been satisfied.

public class Program : IPartImportsSatisfiedNotification

{

[ImportMany]

public IEnumerable<IMessageSender> Senders {get; set;}

public void OnImportsSatisfied()

{

// when this is called, all imports that could be satisfied have been satisfied.

}

}

# Lazy Exports

For some applications delaying this instantiation – and preventing the recursive composition down the graph – may be an important factor to consider as creation a long and complex graph of objects can be expensive and unnecessary.

[Export]

public class HttpServerHealthMonitor

{

[Import]

public Lazy<IMessageSender> Sender { get; set; }

}

# Exports and Metadata

In some cases it’s necessary to associate information with exports for a variety of reasons. Commonly it’s used to explain about the capabilities of an specific implementation of a common contract. This is useful to allow imports to either constraint the export that can satisfy it, or to import all available implementations at the time and check their capabilities in runtime before using the export.

[Export(typeof(IMessageSender))]

[ExportMetadata("transport", "smtp")]

public class EmailSender : IMessageSender

{

public void Send(string message)

{

Console.WriteLine(message);

}

}

if (sender.Metadata.ContainsKey("Transport") && sender.Metadata["Transport"] == MessageTransport.Smtp &&

sender.Metadata.ContainsKey("Issecure") && Metadata["IsSecure"] == true)

Another way is the strong type way.

public interface IMessageSenderCapabilities

{

MessageTransport Transport { get; }

bool IsSecure { get; }

}

[Export]

public class HttpServerHealthMonitor

{

[ImportMany]

public Lazy<IMessageSender, IMessageSenderCapabilities>[] Senders { get; set; }

}

### Metadata filtering and DefaultValueAttribute

When you specifiy a metadata view, an implicit filtering will occur to match **only** those exports which contain the metadata properties defined in the view. You can specify on the metadata view that a property is not required, by using the System.ComponentModel.DefaultValueAttribute. Below you can see where we have specified a default value of false on IsSecure. This means if a part exports IMessageSender, but does not supply IsSecure metadata, then it will still be matched.

public interface IMessageSenderCapabilities

{

MessageTransport Transport { get; }

[DefaultValue(false)];

bool IsSecure { get; }

}

## Catalogs

One of value propositions of MEF's attributed programming model is the ability to dynamically discover parts via catalogs. Catalogs allow applications to easily consume exports that have self-registered themselves via the Export attribute.

### Assembly Catalog

To discover all the exports in a given assembly one would use the **[System.ComponentModel.Composition.Hosting.AssemblyCatalog]**.

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

### Directory Catalog

To discover all the exports in all the assemblies in a directory one would use the **[System.ComponentModel.Composition.Hosting.DirectoryCatalog]**.

var catalog = new DirectoryCatalog("Extensions");

If a relative directory is used it is relative to the base directory of the current AppDomain.

The DirectoryCatalog will do a one-time scan of the directory and will not automatically refresh when there are changes in the directory. However, you can implement your own scanning mechanism, and call Refresh() on the catalog to have it rescan. Once it rescans, recomposition will occur.

var catalog = new DirectoryCatalog("Extensions");

//some scanning logic

catalog.Refresh();

### Aggregate Catalog

var catalog = new AggregateCatalog(

new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly()),

new DirectoryCatalog("Extensions")

### Type Catalog

To discover all the exports in a specific set of types one would use a **[System.ComponentModel.Composition.Hosting.TypeCatalog]**.

var catalog = new TypeCatalog(typeof(type1), typeof(type2), ...);

### Using catalog with a Container

To use a catalog with the container, simpy pass the catalog to the container's constructor.

var container = new CompositionContainer(catalog);

# Filtering Catalogs

When using child containers it may be important to filter catalogs based on some specific criteria. For example, it is common to filter based on part's creation policy.If CreationPolicy is not enough as a criteria to select parts you may want to use the **[System.ComponentModel.Composition.PartMetadataAttribute]** instead.

var catalog = new AssemblyCatalog(typeof(Program).Assembly);

var parent = new CompositionContainer(catalog);

var filteredCat = new FilteredCatalog(catalog,

def => def.Metadata.ContainsKey("scope") &&

def.Metadata["scope"].ToString() == "webrequest");

var perRequest = new CompositionContainer(filteredCat, parent);

var controller = perRequest.GetExportedObject<HomeController>();

perRequest.Dispose();

# Parts Lifetime

The “shareability” of a part is defined through the CreationPolicy set (class level) using the PartCreationPolicyAttribute. The following values are supported:

* **Shared**: the part author is telling MEF that at most one instance of the part may exist per container.
* **NonShared**: the part author is telling MEF that each request for exports of the part will be served by a new instance of it.
* **Any or not supplied value**: the part author allows the part to be used as either “Shared” or “NonShared”.

[Export]

public class Importer

{

[Import(RequiredCreationPolicy=CreationPolicy.NonShared)]

public Dependency Dep { get; set; }

}

|  |  |  |  |
| --- | --- | --- | --- |
| - | Part.Any | Part.Shared | Part.NonShared |
| **Import.Any** | Shared | Shared | Non Shared |
| **Import.Shared** | Shared | Shared | *No Match* |
| **Import.NonShared** | Non Shared | *No Match* | Non Shared |

## Disposing the container

A container instance is generally the lifetime holder of parts. Part instances created by the container have their lifetime conditioned to the container’s lifetime. The way to signal the end of the container lifetime is by disposing it. The implications of disposing a container are:

* Parts that implement IDisposable will have the Dispose method called
* Reference to parts held on the container will be cleaned up
* Shared parts will be disposed and cleaned up
* Lazy exports won’t work after the container is disposed
* Operations might throw System.ObjectDisposedException

Thus, the container will not hold references to parts it creates unless one of the following is true:

* The part is marked as Shared
* The part implements IDisposable
* One or more imports is configured to allow recomposition

## Scoped operations and early reclaim of resources

In order to early release the object graph you need to call the method ReleaseExport exposed by the CompositionContainer:

var batchProcessorExport = container.GetExport<IBatchProcessor>();

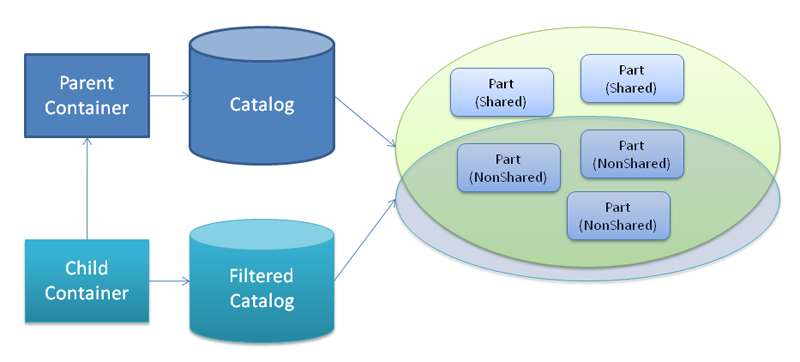
var batchProcessor = batchProcessorExport.Value;

batchProcessor.Process();

container.ReleaseExport(batchProcessorExport);

## Container hierarchies

A common approach is to have Shared parts created on the parent container and Non Shared on the child container(will be released and disposed earlier. ). As Shared parts may depend on exports supplied by Non Shared, then the main catalog will have to contain the whole set of parts whereas the child container should have a filtered view of the main catalog with only the non shared parts.



Disposal ordering

Disposal ordering is not guaranteed in any way. That means that you should not try to use an import in your Dispose method

## AddPart/RemovePart

Not every part is created by the container. You can also add and remove parts from it. This process triggers composition and may start creating parts to satisfy dependencies of the part added recursively. When the part added is removed MEF is smart enough to reclaim the resources and dispose the non shared parts used by the part added.

**Note:** that MEF will never take ownership of an instance supplied by you, but it does have the ownership of part it creates to satisfy your instance’s imports.

# Recomposition

Some applications are designed to dynamically change at runtime. MEF is prepared to handle these kinds of scenarios by relying on what we call recomposition, which is changing values of imports after the initial composition.

[Export]

public class HttpServerHealthMonitor

{

[ImportMany(AllowRecomposition=true)]

public IMessageSender[] Senders { get; set; }

}

# Caveats of Recomposition

* When recomposition occurs, we will replace the instance of the collection / array with a new instance, This is in order to facilitate thread-safety.
* Recomposition is valid for virtually all types of imports supported: fields, properties and collections, but it is not supported for constructor paramters.
* If your type happens to implement the interface **[System.ComponentModel.Composition.IPartImportsSatisifiedNotification]**, be aware that ImportCompleted will also be called whenever recomposition occurs.

# Querying the CompositionContainer

* GetExportedValue
* GetExport
* GetExportedValueOrDefault

# Composition Batch

The batch contains a list of parts to be added and/or removed. After performing the changes, the container automatically triggers a composition which updates recomposable imports affected by the changes.

var batch = new CompositionBatch();

batch.AddPart(partInstance1);

batch.AddPart(partInstance2);

batch.RemovePart(part3);

container.Compose(batch);

## ASP.net MVC Vs ASP.net Web Form

Software Architects have been involving lot of debates about different approaches and architectures. Some of the examples are ORM Vs Store Procedures, REST Vs SOAP, etc. There is a debate happening inside the Microsoft community about ASP.net web form Vs ASP.net MVC. Many people thinking that ASP.net MVC will be replace webforms at least eventually and others are thinking that ASP.net MVC will not be replace webforms. Will ASP.net MVC replace webforms?. ASP.net MVC is an alternative approach to webforms rather than a replacement. It will not replace webforms and webforms will not replace ASP.NET MVC. The fact is that ASP.NET MVC and webforms will co-exist and that ASP.NET MVC is not a replacement for webforms. If you prefer ASP.net MVC use it and you feel webform is more comfortable, you can use it. . Both approaches are just choices and different approaches and choices are good things. Different choices are available for other platforms especially in the Java platform.

**Problems with ASP.net Web Form**

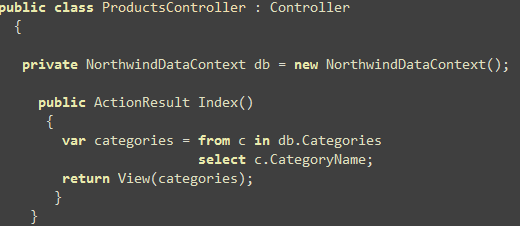
What are the problems with webforms? In webforms, Microsoft has tried to make windows form model development for web application development. That model was attracted lot of windows form developers especially VB 6.0 developers. Many of VB 6.0 developers had moved to ASP.net web development without knowing the basics of HTTP and web. For simulating windows form model development experience, webforms introduced event-driven approach and also introduced Viewstate and Postback. The end result is that web forms breaks the stateless nature of the Web. Both Viewstate and Postbacks have been made lot of problems and increased complexity of the web application development. Many web pages having hundreds of KB size of Viewstate that affected the performance of the applications sometime. Developers do not have the control of the rendering HTML of web forms and Server controls that render html with mixed inline style and deprecated tags that does not follows standards. Another problem with Web Forms is the integration of JavaScript frameworks due to the naming conventions of rendered HTML. The page life cycle of the Web Form is too complex and has the tightly coupling between all things in the ASP.net framework and a single class is used both to display output and handles user input. So unit testing is almost an impossible task. Today unit testing is very important in modern software development especially when we following agile methodologies and practices. Since web is a stateless thing, Events, Postbacks and Viewstate are not a good way. Today many ASP.net web form developers are facing different type pf browser compatibility issues when developing public face internet applications

**The ASP.net MVC way**

The ASP.NET MVC simplifies the complex parts of ASP.net Web Forms without any compromise of the power and flexibility of ASP.NET platform. ASP.net MVC implements Model-View-Controller UI pattern for web application development that lets you allows to develop applications in a loosely couples manner. MVC pattern is separating the application in three parts- Model, View and Controller. A view is responsible for rendering the user interface (UI) of the application and it is nothing more than html templates that filled with application’s data passed by the controller. The Model implements the logic for the application's data and it represents the business objects of the application that using the View for rendering user interface. Controllers are handles and responds to user input and interaction. The web request will be handled by the controller, and the controller will decide which model objects to use and which view objects to render. The MVC model replaces the Web Form events with the controller actions. The main advantages of the MVC models are clear separation of concerns, unit testing facility, and more control over the URLs and HTML. The MVC model does not use Viewstate, Postbacks, Server controls, and server-based forms that enable full control over the application and html rendered by the Views. MVC model is using Representational state transfer (REST) based URLs instead of file-name extensions used by the Web Form model so that we can make search engine optimization (SEO) URLs published by the application.

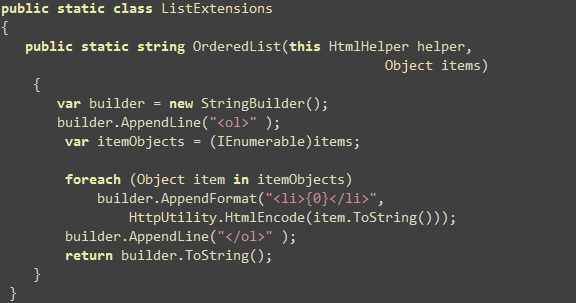
The below code shows the implementation of MVC application.

**ProductsController.cs (Controller)**



In this sample, I have used extension methods to the HtmlHelper class to display ordered list of information.

**OrderListExtensions.cs**



**Category.aspx (View)**



**Advantages of MVC Model**

1. Enable clean separation of concerns (SoC) .
2. Enable full control over the rendered HTML.
3. Enable Test Driven Development (TDD) (built with TDD in mind).
4. SEO and REST friendly URL.
5. Easy integration with JavaScript frameworks.
6. Support third-party view engines such as NVelocity, Brail, NHaml.
7. No ViewState and PostBack events.
8. Follows the stateless nature of web.
9. Extensible and Pluggable framework.
10. Ideal platform for Web 2.0 applications.

**Advantages of Web Form Model**

1. Provides RAD development.
2. Easy development model for heavy data-driven LOB applications.
3. Provides rich controls.
4. Familiar model for windows form developers.

**Which is the best approach?**

The choice would be vary on different people. If you want more control over the HTML or you want Test Driven Development (TDD), or you care about web standards, accessibility, or you want to build SEO based URLs, you can choose MVC model. If you want rich controls and state oriented event-driven web development, you can choose Web Forms model. If you feel more comfortable with MVC, choose that model and you feel Web Form model is more comfortable, choose that model. Both are just choices. If you start your career with ASP.net Web Forms and do not have full knowledge of Web, it will be very difficult moving to MVC model.

I prefer MVC over Web Forms and I feel that Microsoft is going to a right direction through MVC. Its technical features as well as the open source nature are attracted me a lot.The MVC model allows me full control over the HTML and enables Test Driven Development (TDD). We can easily integrate with jQuery and other JavaScript frameworks with MVC. Using extension methods of C# 3.0, we can make powerful and rich HTML helper methods. I believe that tesatbility, refactoring capability and maintainability are the main factors for a successful project and prefer these factors than RAD capability. The MVC model allows to build highly testable, maintainable loosely coupled applications with good practices such as TDD, Seperation of Concerns (SoC) and Dependency Injection (DI). You MUST use ASP.NET MVC for public face internet applications.