MVC-on-MEF high-level components and notes

*NOTE: This document assumes basic familiarity with MVC, EF Code First, and MEF key concepts and components (as well as core .NET platform knowledge).*

This document describes a solution for creating a “composable MVC application using MEF”. The term “composable” here means an MVC application that has most of its functionality externalized in separate assemblies that are loosely coupled with the root application, which ultimately is supposed to make them interchangeable.

To achieve this, the following key approaches are utilized:

1. Views (i.e. those \*.cshtml files) are statically compiled into assemblies as classes, as opposed to the “out of the box” MVC approach of keeping them on the file system and having the ASP.NET runtime compile them on demand.
2. Controllers are exported by individual components through MEF
3. Some bootstrapping mechanisms (such as filters, routes, etc.) are modularized and exported by individual components through MEF
4. Entity Framework’s DbContext is composed dynamically, based on specifications exported by individual components through MEF

Some key concepts and components involved in the implementation are described below.

# Precompiled MVC Views

## ViewCodeGenerator

A special “single file generator” (in VS terminology) that, when attached to a Razor \*.cshtml file as “Custom Tool”, generates a connected \*.cs file that contains the code of the view as it would be generated by the ASP.NET runtime. This file can then be compiled as usual, thus providing the ability to embed MVC Razor views into DLLs.

A few notes on ViewCodeGenerator:

1. There is nothing magical about this generator. The likes of it have been used forever for similar purposes. For example, it is the specialized generator that gives the \*.edmx files their C# expression.
2. To work, the generator has to be installed. The installation is on per-computer basis, meaning that, once installed on a computer, the generator will work for all projects and solutions on that computer, while on other computers, it will not work unless installed, even if the same solution is opened.  
   The generator installs as part of the Lpp.Mvc.VSExtensions.vsix package.
3. For various configuration options, the generator uses the file named “web.config” located in the project root. This is done in order to minimize the differences between developing an MEFed MVC plugin versus developing a standalone MVC application.  
   NOTE: Unlike a regular Web application, these settings get applied to the generated code at time of generation, and cannot be changed after deploying the application (or even after compiling the particular DLL for that matter).
4. In order for the Visual Studio’s Razor text editor to work properly (i.e. provide syntax highlighting, intellisense, etc.) in a “Class Library” type project, the same web.config file has to exist and contain some special settings. See a section below for details.

## CompiledViewEngine

By default, MVC assumes that views are located on the file system as individual files that have to be compiled on demand. In order to use pre-compiled views, MVC has to be extended.

**CompiledViewEngine** class is such extension. It replaces the default view engine by being added as a sole element in the ViewEngines.Engines collection.

Upon request for a view, CompiledViewEngine inspects the current controller to see if it has a CompiledViews attribute applied to it. If it has, the engine tries to find the view by name in the controller’s assembly.

It is possible to specify either short view name, full name (i.e. qualified with namespace), or the fullest of names – assembly-qualified name. The assembly-qualified name defines the type unambiguously – that is, it defines assembly, namespace, and type. In the absence of assembly (the “full name” case), the engine will assume the assembly of the current controller. In the absence of namespace (the “short name” case), the engine will try assume default namespace as that of the controller type, with the substring “.Controllers.” replaced by “.Views.” (thus imitating the default MVC behavior). It is also possible to override default namespace for views by specifying the DefaultNamespace property on the CompiledViews attribute.

If the requested view cannot be found, the engine will fall back to the MVC’s default RazorViewEngine. It is also possible to override the fallback view engine through the CompiledViewEngine.Fallback property.

## Type-safe view identification

The MVC way of identifying a view is by name: the controller specifies **view name as a string**, and the runtime makes sure to look it up.

While this approach can be applied to regular precompiled views via clever heuristics to determine the view type by name (see above), it breaks down for **partial** precompiled views, because in most cases, they do not belong to the currently executing controller, or even to the current controller’s assembly.

This issue can be addressed by using the assembly-qualified type name (see above), which defines the type uniquely, regardless of context.

To type the assembly-qualified name, however, is a major inconvenience (as well as a source poor maintainability). In order to alleviate that inconvenience, a generic version of the HtmlHelper.Partial method was introduced. It may be called as follows:

**<div id=”childContent”>@Html.Partial<MyComponent.Views.MyView>()</div>**

To bring the same benefit of type safety to the controllers as well, a generic version of the Controller.View method was also introduced:

public class MyController : BaseController

{

public ActionResult Index()

{

return View<MyComponent.Views.Index>();

}

}

# MEFed MVC Routing

## Controller names: using full type names

The “out of the box” MVC controller lookup mechanism relies on a **convention**: the route defines the **controller name** (either explicitly or as a route parameter), and the runtime looks up for **a class** named **[Name]Controller** (where [Name] is the value defined in the route).

In the case of composable MVC application, however, this convention breaks down for two reasons:

1. MVC’s default controller factory would only look for controllers in the “root” assembly
2. More importantly, since the author of the root application does not (theoretically) control plugins, there may be controllers with duplicate names

Therefore, using just name for controller identification is not sufficient.

Instead, we use the full, assembly-qualified name of the controller type. For this to work, a custom controller factory has been created that inherits from DefaultControllerFactory and overrides the GetControllerType method.

To alleviate the inconvenience of typing in the full controller name every time, a generic method for routing has been introduced:

**RouteCollection.MapRouteFor<TController>( string url, [object defaults], [object constraints] )**

The method simply calls the “standard” MVC’s MapRoute method using assembly-qualified name of the given type as controller name.

**Caveat:** because the standard MVC view engine uses controller name to look up views, this mechanism does not work anymore. However, since we use precompiled views anyway, this does not have an effect.

If we must use “regular” views, this problem can still be avoided by specifying the full view path instead of just name – i.e. “~/Views/Home/Default.cshtml”. And in case even that is not satisfactory, we can certainly extend the view engine (in the future) so that it will extract “short” controller name from the full one before looking for the view.

## Automatic routing

Some controllers don’t need to have a particular URL assigned to them. But they do need **some** URL, otherwise there would be no way to route any requests to them. For this case, the concept of “automatic routing” has been introduced.

If a controller is exported through MEF and has a metadata value “AutoRoute” set to “true” (see below), that controller will get an automatic route in the form of:

**/\_\_auto/assembly.dll/Namespace.Controller/{action}**

This form ensures uniqueness of routes across controllers.  
(NOTE: technically, there may be several identically-named assemblies, but that can be mitigated easily by adding public key token to the route, should we ever need it, which is unlikely)

To set the “AutoRoute” metadata attribute, one could use the generic MEF’s way of setting metadata – with the [ExportMetadata(“AutoRoute”, true)] attribute, – or, more conveniently, with the specialized [AutoRoute] attribute (which, essentially, has the same effect).

## Explicit routing in declarative way

Currently, to get an explicit route for your controllers, you must export a component by the IRouteRegistrar contract, and call the MapRouteFor<T> method for every route and every controller inside the IRouteRegistrat.RegisterRoutes method.

In the future, a simpler shortcut for this will be implemented. It will allow a plugin author to request explicit route(s) for a controller by giving it certain export metadata values, which will in turn be simplified with a specialized attribute.

## Action URL Helper Methods

Large part of the power of MVC routing comes from its duplex nature. That is, not only controller, action and arguments may be inferred from the URL, but a reverse mapping is also possible. For this purpose, MVC provides several overloads of the Url.Action method. Developers are strongly encouraged to use these methods instead of hard-coding URLs explicitly, because this allows for changing the URL structure without combing through all the code.

Since we do not rely on controller names anymore, these methods also had to be replaced. Instead, two methods were introduced:

1. Url.Action<TController>( string action, [object routeValues] )
2. Url.Action<TController>( Expression<Func<TController,object>> action, [object routeValues] )

The latter method is recommended, since it provides compile-time checking.

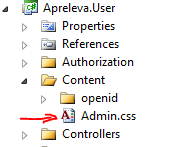
NOTE: The expression tree passed to the second method is NOT examined to extract parameters. It is only used to get the name of the action. The parameters, if any, must be provided through the optional routeValues argument. In the future, the method will be improved to extract parameter values right from the expression tree.

## Mapping resources

A standard MVC app will usually have some plain files in it – i.e. scripts, style sheets, images, etc. These files are usually kept on the file system, and ASP.NET takes care of returning them to the client.

This approach, however, does not work for a composable system: individual modules will compete for the file system, and managing file ownership may become tedious.

Instead, we use .NET resources (aka “Embedded Resources”) to store content files.



Component create route mappings for resources during initialization:

[Export(typeof(IRouteRegistrar))]

class Routes : IRouteRegistrar

{

public void RegisterRoutes( System.Web.Routing.RouteCollection routes )

{

routes.MapResources( typeof( User ).Assembly );

}

And then use specialized extension methods to generate URLs to those resources during rendering:

<link type="text/css" rel="Stylesheet" href="@this.Resource( "Admin.css" )" />

NOTE: for security, only resources located in the “Content” subfolder will be mapped.

## WCF integration

WCF does integrate with MVC natively, through ServiceRoute:

routes.Add( new ServiceRoute( "myservice",

new ServiceHostFactory(),

typeof( MyService ) ) );

The problem is, the service instance does not become a part of MEF composition.  
To cure that, there is a custom WCF activation stack – implementations of ServiceHost, ServiceHostFactory and IInstanceProvider, - which has a neat façade of yet another extensions method RouteCollection.MapService:

routes.MapService<Controllers.RequestService>(

"api/request", Composition );

Optionally, one may provide a continuation for initializing the ServiceHost instance:

routes.MapService<Controllers.RequestService>( "api/request", Composition,

host => host.Description.Behaviors.Add( new ServiceMetadataBehavior { HttpGetEnabled = true } ) );

# Theming

Theming works for compiled views and embedded resources (see sections 1 and 2.5 above). Theming support is built into CompiledViewEngine and ResourceRouteHandler correspondingly.

Themed views and resources may reside either in the same assembly or in a separate assembly. In the latter case, the separate assembly should have the same name as the root assembly postfixed with theme name – i.e. MyApp.Controls.dll and MyApp.Controls.Blue.dll (where “Blue” is the name of a theme).

In the former case, the name of the resource or view itself must be postfixed in a similar manner:

**/Views/Home/Default.cshtml** 🡪 **/View/Home/Default.Blue.cshtml** – for views

**/Content/style.css** 🡪 **/Content/style.Blue.css** – for resources

The name of the “current” theme is defined via the IThemeService interface which has only one member – “string CurrentTheme”. The component implementing the interface should be exported via MEF – then it will get picked up by both CompiledViewEngine and ResourceRouteHandler.

# Persistence composition

## Composable model definition

EF Code First, in its simplest form, is represented by a class inherited from DbContext that has a bunch of properties of type DbSet<T> where T is the type of a business entity. This model does not allow for compositional approach, because it requires knowledge of all entity types at the point of persistence root definition.

However, this is not the only way the model can be defined. DbContext provides a way to hook into the model building process via overriding the OnModelCreating method. Inside that method, one may call modelBuilder.Entity<T>(), and that very act will add the type T to the model, even if there is no corresponding property of type DbSet<T>.

From there, one does not have to think hard to see that the calls to modelBuilder.Entity<T>() do not have to be all defined in one place, but may be aggregated in a pluggable way.

The **ComposableDbContext** class leverages that ability:

public class ComposableDbContext : DbContext, IUnitOfWork

{

private readonly IEnumerable<IPersistenceDefinition> \_definitions;

[ImportingConstructor]

public ComposableDbContext( [ImportMany] IEnumerable<IPersistenceDefinition> defs )

{

\_definitions = defs;

}

protected override void OnModelCreating( DbModelBuilder modelBuilder )

{

foreach ( var d in \_definitions ) d.BuildModel( modelBuilder );

}

Where the IPersistenceDefinition interface consists of a single member, “void BuildModel( DbModelBuilder )”.

The one inconvenience of this approach is that every component of the system has to explicitly declare its entities that require persistence by exporting a corresponding IPersistenceDefinition. Like so:

public static class Persistence

{

[Export] public static IPersistenceDefinition Albums {

get { return new PersistenceDefinition<Album>(); } }

[Export] public static IPersistenceDefinition AlbumTracks {

get { return new PersistenceDefinition<AlbumTrack>(); } }

}

The ideal way would be to build the model on demand, just based on what is required. That, however, is impossible with EF, because EF requires that the model be completely known in advance. I’m working on simplifying this part.

## Consuming persistence

The overall persistence approach is built using the UnitOfWork/Repository pattern, where the base library exports IUnitOfWork and generic IRepository<T> interfaces, and those components interested in persistence may import them.

# Root application concerns

## Composition scoping

In a web application, some parts (in MEF terminology) must have different scopes. For example, controllers must be created anew for each request and disposed of when request is finished. Same stands for data persistence components. Other components, such as authentication service or MVC filters, have to be created at application start and never die.

In MEF, the component lifetime is defined by CompositionContainer. Therefore, in order to achieve scoping, a new CompositionContainer must be created at the start of each scope and disposed of at the end. The scope container references the container of enclosing scope, so that its part can reference higher-scoped parts. Part catalogs for each container are filtered so that each container only sees parts that are intended for corresponding scope.

The definition of scope is left to the part author. That is, the author of a part may decorate it (via MEF metadata and specialized [ExportScope] attribute) with a scope ID. Scope ID is simply a string of free format. That string is later used when filtering catalogs for scoped components.

The persistence subsystem defines a constant named UnitOfWorkScope.Id that is used to designate components that are supposed to be scoped to unit of work (such as DbContext or Repository).

The MVC subsystem defines a constant named WebScope.Request that is used to designate components that are supposed to be scoped to web request.

It is possible to define more than one scope. In that case, the component will be considered scoped to every one of those scopes.

## Application bootstrapping abstraction

For all the above to work, the root application should do some bootstrapping:

Creates composition container

In the beginning of each request, create a nested container for request scope

In the end of each request, dispose of it

Import MVC-related parts and register them with MVC (controllers, model binders, filters, modules, etc.)

Set MVC’s IDependencyResolver implementation to wrap MEF

Import IRouteRegistrar’s and call them to register routes

Set FullTypeNameControllerFactory as current controller factory

Set CompiledViewEngine as the only view engine

Turns out, all this functionality does not depend on any actual application-specific business concerns. That is, all the items above are same for all applications (as long as they use this platform).

Therefore, this bootstrapping functionality has been abstracted away in the form of base class LppMvcComposableApplication. This class inherits from System.Web.HttpApplication, and is intended to be used as base class for Global.asax.

## Summary for creating a new plugin project

Considering all the above descriptions and gotchas, here is what needs to be done in order to create a new “plugin” project:

1. Create a “Class Library” project
2. Copy over the web.config file
3. In section /configuration/system.web.webPages.razor/pages/namespaces, add the namespaces where you will have your Controllers and Models, plus any additional you see necessary.
4. Add references to:
   1. System.Web
   2. System.Web.Mvc
   3. System.Web.Routing
   4. System.Web.WebPages
   5. EntityFramework (optional)
   6. System.ComponentModel.Composition.CodePlex (MEF2, not yet released)
   7. System.ComponentModel.Composition.Registration.CodePlex (MEF2, not yet released)
   8. System.Reflection.Context.CodePlex (.NET 4.5, not yet released)
   9. Lpp.Utilities
   10. Lpp.Mvc
   11. Lpp.Composition
   12. Lpp.Mvc.Composition
   13. Lpp.Data.Composition
   14. Lpp.Mvc.Application
5. Set its output path to the “bin” directory of the root MVC app
6. Every time a new view needs to be created, set its Custom Tool to “ViewCodeGenerator”

To simplify this tedious process, there is a project template called “LPP MVC Composable Part”. This template installs from the same \*.vsix package that the ViewCodeGenerator does.

The project template does all the above steps, except 4.f-4.n, 5, and 6 (though the initially created sample view does have the custom tool defined).

Steps 4.f-4.h will be automated once .NET 4.5 is released.  
Steps 4.i-4.n will be automated once the libraries are stable.   
Step 5 will be automated once I have enough time to figure it out.  
Step 6 – I’m not sure it needs to be automated at all.