Hands-On Lab

Introduction to the Managed Extensibility Framework

Lab version: 1.0.0

Last updated: 12/10/2010
CONTENTS

OVERVIEW ..................................................................................................................................................3

EXERCISE 1: USING MEF TO EXTEND AN APPLICATION ..............................................................................5
  Task 1 – Updating Your Application to Load Composable Parts .................................................................6
  Task 2 – Updating Your Composable Part to Export Contracts .................................................................11
  Task 3 – Creating a New Extension ............................................................................................................16
  Exercise 1: Verification ...............................................................................................................................23

EXERCISE 2: USING METADATA AND LAZY-LOADING MODULES ..............................................................23
  Task 1 – Attaching Metadata to an Export ...................................................................................................24
  Task 2 – Using a Custom Export Attribute ..................................................................................................25
  Task 3 – Importing Metadata .......................................................................................................................32
  Exercise 2: Verification ...............................................................................................................................38

EXERCISE 3: USING MEF AND SILVERLIGHT 4 ..........................................................................................40
  Task 1 – Convert Extensions to Silverlight ..................................................................................................40
  Task 2 – Create Silverlight Cash Maker UI ................................................................................................48
  Task 3 – Update Your Silverlight Application to Load Composable Parts ..................................................55
  Task 4 – Load Silverlight Extensions Dynamically ...................................................................................57
  Exercise 3: Verification ...............................................................................................................................61

SUMMARY .....................................................................................................................................................64
Overview

The Managed Extensibility Framework (MEF) allows developers to provide hooks into their .NET applications for extensions by first and third parties. MEF can be thought of as a general application extension facility.

MEF enables developers to create extensions dynamically, without the extending application or the extension requiring any specific knowledge of the other. This ensures no coupling exists between the two at compile time, enabling applications to be extended at runtime, without the need to recompile. MEF also provides the ability to examine the metadata of an extension assembly before loading the extension into the application -- a much faster approach.

Several key concepts relating to extensibility are covered in this lab:

- **Composition** is the practice of combining several objects with discrete functionality into one or more complex objects. Composition is not inheritance of functionality from a parent class, but is instead the assembling of several different objects into one. For example, Wing, Propeller, Fuselage, and VerticalStabilizer objects could be composed as part of an Aircraft object.

- **ComposableParts** are the key building blocks of MEF. ComposableParts enable applications to expose and consume component extensions via Exports and Imports.

- **Contracts** are the avenue for communication between Export and Import components. A Contract is normally implemented via an Interface class. Contracts enable MEF ComposableParts to avoid dependencies or tight coupling with other components.

- **Conditional Binding** allows loading of components, which meet specific metadata criteria. Following the example above, you might choose to load VerticalStabilizer components, which were made only of composite graphite.

Extensibility is achieved, primarily, by adding Import attributes at appropriate points in the application and adding corresponding Export attributes to the extensions. Import and Export attributes can be thought of in the same aspect as a supplier and consumer respectively. Export components supply some value; Import components consume that value. Other options for extensibility are open to developers, including completely custom extensibility approaches; however, this lab focuses solely on the primary approach discussed above. Moreover, the components can be imported using the ImportMany attribute, which loads all the components that match the given contract as a collection.

With the release of Silverlight 4, MEF has been included in it, allowing you to extend rich internet applications. This does not imply that MEF can only be used with Silverlight 4; using the version available in codeplex (http://mef.codeplex.com), MEF can be used to compose applications created in Silverlight 3.
Objectives
In this Hands-On Lab, you will learn how to:

- Create extensibility modules and components
- Import Extended assemblies while an application is running
- Extend a Silverlight 4 application (optional)

System Requirements
You must have the following items to complete this lab:

- Microsoft Visual Studio 2010
- .Net Framework 4
- Silverlight 4 (optional)
- Silverlight 4 SDK (optional)
- Silverlight 4 Tools for Visual Studio 2010 (optional)

Setup
All the requisites for this lab are verified using the Configuration Wizard. To make sure that everything is correctly configured, follow these steps.

**Note:** To perform the setup steps you need to run the scripts in a command window with administrator privileges.

1. Run the Configuration Wizard for the Lab if you have not done it previously. To do this, run the CheckDependencies.cmd script located under the Source\Setup folder of this lab. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.

   **Note:** For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The CheckDependencies.cmd file launches the Visual Studio installer file that installs the code snippets.

Exercises
This Hands-On Lab comprises the following exercises:
1. Using MEF to Dynamically Add Modules to an Application
2. Using Metadata and Lazy-Loading Modules
3. Using MEF and Silverlight 4

Starting Materials

This Hands-On Lab includes the following starting materials.

- Visual Studio solutions. Depending on the exercise you will find Visual Studio solutions that you can use as starting point for the exercises.

**Note:** Each exercise is accompanied by an End folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

**Note:** Each exercise contains a Visual Basic and a C# version; Inside the End/Begin solution folder, you will find two folders: VB, containing the Visual Basic version of the exercise, and C#, containing the C# version of it.

Estimated time to complete this lab: **30 minutes**.

Next Step

Exercise 1: Using MEF to Dynamically Add Modules to an Application

Exercise 1: Using MEF to Extend an Application

One practical use of the Managed Extensibility Framework is adding modules to an application at runtime. This is useful in a scenario in which users chooses specific modules to purchase or install originally and may add more modules at a later time. Using MEF, you can configure your application to monitor a well-known directory and add any module assemblies found in that directory. Dropping module assemblies into a directory allows your application to load those assemblies without explicitly setting references to them.
Task 1 – Updating Your Application to Load Composable Parts

In this task, you will modify an existing WPF Window class to create extension hooks that will allow dynamically importing queries. You will update this query classes later in this exercise.


2. Open the ContosoAutomotive.sln solution file. By default, this file is located in the folder Source\Ex1\begin (choosing the folder that matches the language of your preference.)

3. Add a reference to the MEF library on the ContosoAutomotive project. To do this:
   a. Select the ContosoAutomotive project in the Solution Explorer and select Project | Add Reference... The Add References dialog appears.
   b. Select the .NET tab and then select the System.ComponentModel.Composition component. Click the OK button to add a reference to this library.

![Add Reference](image)

*Figure 1*
*Add a Reference to the MEF library.*

4. Open the App class in code view. To do this, right-click on the App.xaml file in the Solution Explorer and select View Code.
5. Update the App class to use the MEF library. To do this, add the following statement on top of the using clause list above the App class definition.

C#

```csharp
using System.ComponentModel.Composition.Hosting;
```

Visual Basic

```vbnet
Imports System.ComponentModel.Composition.Hosting
```

6. Create an aggregate catalog to load parts from the current application assembly and from assemblies located in the current folder in the file system. Also, create a composition container instance based on it and create an exported CashMaker instance. To do this, modify the AppStartup method with the following code.

(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step6 - StartupCatalogAndContainer CSharp)

C#

```csharp
void AppStartup(object sender, StartupEventArgs args)
{
    var catalog = new AggregateCatalog(new DirectoryCatalog("."),
                                         new AssemblyCatalog(Assembly.GetExecutingAssembly()));
    var container = new CompositionContainer(catalog);

    var window = container.GetExportedValue<CashMaker>();
    window.Show();
}
```

(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step6 - StartupCatalogAndContainer VB)

Visual Basic

```vbnet
Sub AppStartup(ByVal sender As Object, ByVal args As StartupEventArgs)
{
    Dim catalog = New AggregateCatalog(New DirectoryCatalog("."),
                                         New AssemblyCatalog(Assembly.GetExecutingAssembly()))
    Dim container = New CompositionContainer(catalog)

    Dim window = container.GetExportedValue(Of CashMaker)()
    window.Show()
}
```

Note: The starting point for enabling MEF composability is creating a composition container based on one or several catalogs. MEF's container interacts with Catalogs to have access to composable parts. The container itself resolves a part's dependencies and exposes Exports to
the outside world. In addition, you are free to add composable part instances directly to the container if you wish.

**Note:** There are different kinds of Catalogs, which are able to discover parts on different ways. This exercise uses an Aggregate Catalog combining a Directory Catalog searching on the Application directory (represented by the “.”) and Assembly Catalog for look at the types inside the application assembly (retrieved using the Assembly.GetExecutingAssembly method).

The full list of Catalogs available on MEF is:

- **Assembly Catalog:** Discovers the different parts on a specific assembly.
- **Directory Catalog:** Discovers parts inside the assemblies on a specific directory.
- **Aggregate Catalog:** Allows use more than one catalog combining them.
- **Type Catalog:** Looks inside the assemblies for specific types.

For more information on Catalogs, see [Using catalogs](#).

**Note:** Application’s main window is also located and loaded using MEF through the container’s `GetExportedValue` method. In the following steps, you will decorate the `CashMaker` class (which is the main window) with the `Export` attribute to be discoverable by MEF.

The `GetExportedValue` method will throw an exception if no parts or more than one implement the solicited contract. This is not a problem in the exercise implementation because we are not using an interface to discover the type, we are using the class itself so it have to be defined and cannot be duplicated ensuring that MEF will find only one instance.

7. Open the CashMaker Window in code view. To do this, right-click on the `CashMaker.xaml` file in the Solution explorer and select **View Code**.

8. Update the CashMaker class to use the MEF library. To do this, add the following statement at the top of the `CashMaker` class definition.

```csharp
using System.ComponentModel.Composition;
```

```vbnet
Imports System.ComponentModel.Composition
```
9. Since the CashMaker class is now a composable part you will export the class making it available to the composition container. To do this, decorate the CashMaker class with the Export attribute as shown in the code below.

**C#**

```csharp
[Export]
public partial class CashMaker : Window
{
...
}
```

**Visual Basic**

```vbnet``
<Export()>
Class CashMaker
    Inherits Window
...
End Class
```

**NOTE:** Import and Export values are interconnected with unique Contracts. Contracts serve as uni-directional bridges. An export contract can consist of further metadata used to filter on its discovery. For example, it might indicate a specific capability that the export offers.

As you decorate the CashMaker class with the Export attribute, it will discover composable parts exporting the ICarQuery contract, defined in the ContosoAutomotive.Common project. You will declare a collection to hold the Imports and expose them through a property. MEF's composition container will resolve applicable Exports and Imports loading the collection for you.

10. Add the following code to the CashMaker.xaml.cs (C#) or CashMaker.xaml.vb (Visual Basic) file, just above its constructor.

(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step10 - ImportedCarQueries CSharp)

**C#**

```csharp```
[ImportMany(AllowRecomposition = true)]
public ObservableCollection<ICarQuery> CarQueries { get; set; }
```

(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step10 - ImportedCarQueries VB)

**Visual Basic**

```vbnet```
<ImportMany(AllowRecomposition:=True)>
Public Property CarQueries() As ObservableCollection(Of ICarQuery)
```
11. Once MEF completes the CarQueries collection import process, you want your application to be notified and so it can take a particular action. To do this, make the CashMaker class to implement the IPartImportsSatisfiedNotification interface as shown in the following code.

**C#**

```csharp
[Export]
public partial class CashMaker : Window, IPartImportsSatisfiedNotification
{
    ...
}
```

**Visual Basic**

```vbnet
<Export>
Public Partial Class CashMaker
    Inherits Window
    Implements IPartImportsSatisfiedNotification
    ...
End Class
```

12. Implement the IPartImportsSatisfiedNotification interface and bind the CarQueries collection to the UI by setting the DataContext property of the commandGrid control. To do this, paste the following `OnImportsSatisfied` method code inside the CashMaker class.

*(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step12 - OnImportsSatisfied CSharp)*

**C#**

```csharp
public void OnImportsSatisfied()
{
    this.commandGrid.DataContext = this.CarQueries;
}
```

*(Code Snippet – Intro to MEF Lab - Ex1 Task1 Step 12 - OnImportsSatisfied VB)*

**Visual Basic**

```vbnet
Public Sub OnImportsSatisfied() Implements IPartImportsSatisfiedNotification.OnImportsSatisfied
    Me.commandGrid.DataContext = Me.CarQueries
```

**Note:** 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.
End Sub

Note: After MEF’s Container fills the different imports inside the class, it calls the OnImportsSatisfied method if the class implements the IPartImportsSatisfiedNotification interface. This method is used as a notification when the container has filled all the imports. It is also called after a recomposition occurs.

13. Now you can remove the old code used to explicitly create and bind the query instances. To do this, in the CashMaker constructor remove the code used to populate the commandGrid.DataContext property as shown below.

C#

```csharp
public CashMaker()
{
    this.InitializeComponent();

    new Thread(() => this.GenerateCars()).Start();

    // Removed code
}
```

Visual Basic

```vbnet
Public Sub New()
    Me.InitializeComponent()

    Dim thread = New Thread(() => Me.GenerateCars())
    thread.Start()

    ' Removed code
End Sub
```

Task 2 – Updating Your Composable Part to Export Contracts

In this task, you will modify an existing Windows Class Library project with services classes to query a collection of cars. You will use MEF to mark the queries to export a contract to another application.

1. Add a reference to the MEF library on the ContosoAutomotive.Extensions project. To do this:
   a. Select the ContosoAutomotive.Extensions project in the Solution Explorer and select Project | Add Reference... The Add References dialog appears.
   b. Select the .NET tab and then select the System.ComponentModel.Composition component. Click the OK button to add a reference to this library.
2. Open the CohoQuery.cs file (for Visual C# projects) or CohoQuery.vb file (for Visual Basic projects) in the ContosoAutomotive.Extensions project. Add the following namespace declaration to import the types contained in the MEF library at the top of the CohoQuery class definition.

```csharp
using System.ComponentModel.Composition;
```

```vbnet
Imports System.ComponentModel.Composition
```

3. The CohoQuery class is one of several parts available in your application. Take into account that you used the ImportMany attribute in the CashMaker class to pull in only components that implement the ICarQuery contract. You will decorate the CohoQuery class indicating that it matches that contract. To do this, decorate the CohoQuery class with the Export attribute as shown in the code below:
4. Now, you will apply the same changes to the FabrikamQuery class. To do this, open the `FabrikamQuery.cs` file (for Visual C# projects) or `FabrikamQuery.vb` file (for Visual Basic projects) in the `ContosoAutomotive.Extensions` project. Add the following namespace declaration to import the types contained in the MEF library at the top of the FabrikamQuery class definition.

**C#**

```csharp
using System.ComponentModel.Composition;
```

**Visual Basic**

```vbnet
Imports System.ComponentModel.Composition
```

5. You will decorate the FabrikamQuery class indicating that it matches the ICarQuery contract. To do this, decorate the `FabrikamQuery` class with the `Export` attribute as shown in the code below:

**C#**

```csharp
[Export(typeof(ICarQuery))]
public class FabrikamQuery : CarQueryBase
{
    ...
}
```
Visual Basic

```vbnet
<Export(GetType(ICarQuery))>
Public Class FabrikamQuery
    Inherits CarQueryBase
    ...
End Class
```

6. At this point, you can start a new instance of the ContosoAutomotive project to check that the two queries are shown in the UI. To do this, right click the **ContosoAutomotive** project and select **Debug | Start new instance**.

![Contoso Automotive Resellers](image)

*Figure 1*
*MEF loaded queries in ContosoAutomotive application.*

7. To verify the MEF extensibility, you will add several pre-built query classes. To do this, right-click **ContosoAutomotive.Extensions** in **Solution Explorer**, point to **Add** and select **Existing Item**. In the **Add Existing Item** dialog, browse to the **ContosoAutomotive.Extensions** in **Source\Assets** for the language of your project, hold the **CTRL** key down while you select every file in this folder and click **Add**.
Figure 2

*Added new queries in ContosoAutomotive.Extensions project (C#)*

Figure 3

*Added new queries in ContosoAutomotive.Extensions project (Visual Basic)*
8. At this point, you can start a new instance of the ContosoAutomotive project to check that the two queries are shown in the UI. To do this, right click the ContosoAutomotive project and select Debug | Start new instance.

![Contoso Automotive Resellers](image)

**Figure 4**

*MEF loaded queries in ContosoAutomotive application.*

**Task 3 – Creating a New Extension**

In this task, you will create a new application extension with a query services class. You will use MEF to Export contracts and a build action to avoid references between projects. Additionally, you can mix CLR languages and create the new project in different CLR language than your main application.

1. Create a new application extension project named Woodgrove. To do this, right click the solution node in the Solution Explorer and select **Add | New Project**. In the Add New Project dialog select **Windows Class Library** project type and name it ContosoAutomotive.Woodgrove.
Figure 5

New Woodgrove extension project (C#)
2. Delete the default class file Class1.cs (C#) or Class1.vb (Visual Basic).

3. Add a reference to the MEF library on the ContosoAutomotive.Woodgrove project. To do this:
   a. Select the ContosoAutomotive.Woodgrove project in the Solution Explorer and select Project | Add Reference... The Add References dialog appears.
   b. Select the .NET tab and then select the System.ComponentModel.Composition component. Click the OK button to add a reference to this library.
4. Add a reference to the ContosoAutomotive.Common library on the **ContosoAutomotive.Woodgrove** project. To do this:
   a. Select the **ContosoAutomotive.Woodgrove** project in the **Solution Explorer** and select **Project | Add Reference**... The **Add References** dialog appears.
   b. Select the **Projects** tab and then select the **ContosoAutomotive.Common** component. Click the **OK** button to add a reference to this library.
5. Create a new class named WoodgroveQuery. To do this, right click the ContosoAutomotive.Woodgrove project and select Add | Class and name it WoodgroveQuery.

6. Open the WoodgroveQuery.cs file (for Visual C# projects) or WoodgroveQuery.vb file (for Visual Basic projects) in the ContosoAutomotive.Woodgrove project. Add the following namespace declaration to import the types contained in the MEF library at the top of the WoodgroveQuery class definition.

(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step6 – WoodgroveQueryNamespaces CSharp)

C# using System.ComponentModel.Composition;
using ContosoAutomotive.Common;

(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step6 - WoodgroveQueryNamespaces VB)

Visual Basic Imports System.Collections.Generic
Imports System.ComponentModel.Composition
7. The *WoodgroveQuery* class matches the same contract and implements the same interface as the previous modified classes. You will implement a new query class similar to the previous one. To do this, replace the class implementation with the following code.

*(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step7 - WoodgroveQueryClass CSharp)*

```csharp
[Export(typeof(ICarQuery))]
public class WoodgroveQuery : CarQueryBase
{
    public WoodgroveQuery()
    {
        this.Name = "Woodgrove Cycles";
        this.Description = "Who doesn't love a Woodgrove? The fastest thing on two wheels!";
        this.ImagePath = "Images/woodgrove.jpg";
    }

    protected override IEnumerable<Car> RunQuery(IEnumerable<Car> cars)
    {
        var results = from c in cars
                       where c.Make == "Woodgrove"
                       && c.Price >= 50000
                       && c.Year >= 2000
                       && c.SatelliteRadio == true
                       && c.Mpg >= 20
                       && c.Interior == InteriorType.Suede
                       && c.MoonRoof == false
                       && c.Mileage <= 40000
                       select c;

        return results;
    }
}
```

*(Code Snippet – Intro to MEF Lab - Ex1 Task3 Step7 - WoodgroveQueryClass VB)*

```vbnet
<Export(GetType(ICarQuery))>
Public Class WoodgroveQuery
    Inherits CarQueryBase

    Public Sub New()
        Me.Name = "Woodgrove Cycles"
    End Sub
End Class
```
**Description**

Who doesn't love a Woodgrove? The fastest thing on two wheels!

**ImagePath**

Images/woodgrove.jpg

---

```vbnet
Protected Overrides Function RunQuery(ByVal cars As IEnumerable(Of Car)) As IEnumerable(Of Car)

    Dim results = From c In cars 
    Where c.Make = "Woodgrove" 
    AndAlso c.Price >= 50000 
    AndAlso c.Year >= 2000 
    AndAlso c.Transmission = Transmission.Manual 
    AndAlso c.SatelliteRadio = True 
    AndAlso c.Mpg >= 20 
    AndAlso c.Interior = InteriorType.Suede 
    AndAlso c.MoonRoof = False 
    AndAlso c.Mileage <= 40000

    Return results
End Function
End Class
```

---

8. For the new extension project to be discoverable by the catalog, you should place the assembly in the same folder as the main application. You can achieve this in the development environment by configuring the build events. To do this, right click on the ContosoAutomotive.Woodgrove project and select Properties. In the Properties page, select the Build Events tab (for C# projects) or select the Compile tab and press the Build Events... button (for Visual Basic projects) and enter the following value in the Post-build event command line:

**Post-Build Command**

```text
Post-Build Command

copy "$(TargetPath)"
"$(SolutionDir)ContosoAutomotive\bin\$(ConfigurationName)"
```

**Note:** MEF's Container will be able to discover the application parts inside the new assembly since you are copying it to the application directory, and a Directory Catalog is looking at it.

---

**Next Step**

**Exercise 1: Verification**
Exercise 1: Verification

In this verification, you will run the application to verify the new queries are shown in the main application.

1. Compile the solution (CTRL+SHIFT+B).
2. Set ContosoAutomotive as the startup project. In the Solution Explorer, right-click ContosoAutomotive and select Set as startup project.
3. Press F5 to run the application. The parent CashMaker window should appear with a list of manufacturers on it.

![CashMaker window showing manufacturers](image)

Figure 7
The CashMaker window shows manufacturers.

4. Click the close button in the top right of the CashMaker window to close it.

Next Step
Exercise 2: Dynamically Extend a Form

Exercise 2: Using Metadata and Lazy-Loading Modules
During composition of a part, an import will trigger the instantiation of a part (or parts) that expose the necessary exports required for the original requested part. For some applications, delaying this instantiation and preventing the recursive composition down the graph, may be an important factor to consider is that the creation of a long and complex graph of objects can be expensive and unnecessary.

This is the motivation for MEF to support what we call lazy exports including the Lazy generic type, this type allows delay the load process of the objects until the moment it is first called, minimizing the time required to initialize the application.

In this type of scenarios, associating information with the export becomes critical to inform the importer about the capabilities of a specific implementation of a common contract.

In this exercise, we will tackle both problems in a single scenario.

**Task 1 – Attaching Metadata to an Export**

In some cases, it is necessary to associate information with exports for a variety of reasons. Commonly you use metadata to explain about the capabilities of a specific implementation of a common contract. In this task, you will modify an existing Query implementation to expose some properties as metadata.


2. Open the ContosoAutomotive.sln solution file. By default, this file is located in the folder Source\Ex2\begin (choosing the folder that matches the language of your preference.) Optionally, you can continue working the solution you created in the previous exercise.

3. You will start adding metadata to the WoodgroveQuery implementation. To do this, open the WoodgroveQuery.cs (C#) or WoodgroveQuery.vb (Visual Basic) file under the ContosoAutomotive.Woodgrove project.

4. Since you will attach metadata by using attributes, you no longer required the initialization code. Remove the WoodgroveQuery constructor that initializes the Name, Description and ImagePath properties.

5. To attach metadata to an Export implementation, you decorate the class with the ExportMetadata attribute. In this case, you will attach the same information you removed from the constructor initialization. To do this, decorate the WoodgroveQuery class with the following attributes.

(Code Snippet – Intro to MEF Lab - Ex2 Task1 Step5 -WoodgroveQueryMetadata CSharp)

```csharp
[Export(typeof(ICarQuery))]
[ExportMetadata("Name", "Woodgrove Cycles")]
[ExportMetadata("Description", "Who doesn't love a Woodgrove? The fastest thing on two wheels!")]
[ExportMetadata("ImagePath", "Images/woodgrove.jpg")]
```
public class WoodgroveQuery : CarQueryBase
{
  // Removed code
...
}

(Code Snippet – Intro to MEF Lab - Ex2 Task1 Step5 - WoodgroveQueryMetadata VB)

Visual Basic

<Export(GetType(ICarQuery))>
<ExportMetadata("Name", "Woodgrove Cycles")>
<ExportMetadata("Description", "Who doesn't love a Woodgrove? The fastest thing on two wheels!")>
<ExportMetadata("ImagePath", "Images/woodgrove.jpg")>

Public Class WoodgroveQuery
  Inherits CarQueryBase

    ' Removed code
...
End Class

Note: You decorate a class with the ExportMetadata attribute to include metadata to it. This attribute receive two parameters defining the name and the value of the class’ metadata respectively.

Task 2 – Using a Custom Export Attribute

In order to attach metadata to an export in a more strongly typed fashion than using the ExportMetadata attribute, you can create your own attribute and decorate it with the Metadata attribute. In this task, you will create a custom metadata attribute and derive it from Export attribute, thus creating a custom Export attribute that also specifies metadata.

1. Create a new QueryMetadata class that will serve to both define an export and attach metadata at the same time. To do this, right click on the ContosoAutomotive.Extensions project and select Add | Class. In the Add New Item dialog, name the class as QueryMetadataAttribute and click OK.
Figure 8
Adding QueryMetadataAttribute class (C#)

Figure 9
Adding QueryMetadataAttribute class (Visual Basic)
2. Update the QueryMetadataAttribute class to use the MEF library. To do this, add the following statement at the top of the `QueryMetadataAttribute` class definition.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step2 - Namespaces CSharp)

C#  
```
using System.ComponentModel.Composition;
using ContosoAutomotive.Common;
```

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step2 - Namespaces VB)

Visual Basic  
```
Imports System.ComponentModel.Composition
Imports ContosoAutomotive.Common
```

3. Make the QueryMetadataAttribute class definition public and decorate it with the `MetadataAttribute` and `AttributeUsage` attributes. In addition, make the class inherits from the `ExportAttribute` class. To do this, replace the default class definition with the following code.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step3 - QueryMetadataAttribute CSharp)

C#  
```
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class QueryMetadataAttribute : ExportAttribute
{
}
```

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step3 - QueryMetadataAttribute VB)

Visual Basic  
```
<MetadataAttribute(), AttributeUsage(AttributeTargets.Class,
AllowMultiple:=False)>
Public Class QueryMetadataAttribute
    Inherits ExportAttribute
End Class
```

**Note:** Both attributes, MetadataAttribute and AttributeUsage, are required to be included as decorators of the typed Metadata class.
MetadataAttribute defines that the class can be used to define typed metadata of another class.
AttributeUsage allows set where the attribute can be placed. In this exercise, the attribute can decorate a class, and cannot be used more than one time on the same class.

4. Add the required metadata properties and setup the inherited constructor to export the ICarQuery interface explicitly. To do this, insert the following code inside the QueryMetadataAttribute class.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step4 - QueryMetadataImplementation CSharp)

**C#**

```csharp
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class QueryMetadataAttribute : ExportAttribute
{
    public QueryMetadataAttribute()
    {
        base(typeof(ICarQuery))
    }

    public string Name { get; set; }
    public string Description { get; set; }
    public string ImagePath { get; set; }
}
```

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step4 - QueryMetadataImplementation VB)

**Visual Basic**

```vbnet
<MetadataAttribute(), AttributeUsage(AttributeTargets.Class, AllowMultiple:=False)>
Public Class QueryMetadataAttribute
    Inherits ExportAttribute

    Public Sub New()
        MyBase.New(GetType(ICarQuery))
    End Sub

    Public Property Description As String
    Public Property ImagePath As String
    Public Property Name As String

End Class
```
Note: Inheritance from ExportAttribute and calling the ExportAttribute class’ constructor passing the ICarQuery type automatically sets that the contract implemented by the class is the ICarQuery. This allows avoid specify it when you decorate the class as you did when you used the ExportAttribute directly.

5. Now, you can attach metadata and export a contract using the QueryMetadata attribute in your classes removing both the Export decorating attribute and the initialization constructor. To do this, remove the constructor and replace the class definition in the CohoQuery class.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step5 - CohoQueryMetadata CSharp)

C#  

```csharp
[QueryMetadata(Name = "Coho Auto",
    Description = "Nothing beats a good price on a Coho. We can't keep these things in stock.",
    ImagePath = "Images/coho.jpg")]
public class CohoQuery : CarQueryBase
{
    // Removed code
    ...
}
```

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step5 - CohoQueryMetadata VB)

Visual Basic  

```vbnet
<QueryMetadata(Name:="Coho Auto",
    Description:="Nothing beats a good price on a Coho. We can't keep these things in stock.",
    ImagePath:="Images/coho.jpg")>
Public Class CohoQuery
    Inherits CarQueryBase
    // Removed code
    ...
End Class
```

6. You will repeat the same process on the FabrikamQuery class. To do this, remove the constructor and replace the class definition in the FabrikamQuery class.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step6 - FabrikamQueryMetadata CSharp)

C#  

```csharp
null
```
7. You will repeat the same process on the LitwareQuery class. To do this, remove the constructor and replace the class definition in the LitwareQuery class.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step7 - LitwareQueryMetadata CSharp)

C#

```csharp
[QueryMetadata( Name = "Litware Limousines",
                Description = "Grab a Litware. We can make an absolutely steal off these party favorites.",
                ImagePath = "Images/litware.jpg")]
public class LitwareQuery: CarQueryBase
{
    // Removed code
    ...
}
```

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step7 - LitwareQueryMetadata VB)

Visual Basic

```vbnet
<QueryMetadata( Name:="Litware Limousines",
                Description:="Grab a Litware. We can make an absolutely steal off these party favorites.",
                ImagePath:="Images/litware.jpg")>
Public Class LitwareQuery
    Inherits CarQueryBase
    // Removed code
    ...
End Class
```
8. You will repeat the same process on the TailspinQuery class. To do this, remove the constructor and replace the class definition in the **TailspinQuery** class.

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step8 - TailspinQueryMetadata CSharp)

C#  

```
[QueryMetadata(Name = "Tailspin Motorsport",
   Description = "Life in the fast lane. You can't live without the best: Tailspin.",
   ImagePath = "Images/tailspin.jpg")]
public class TailspinQuery: CarQueryBase
{
   // Removed code
   ...
}
```

(Code Snippet – Intro to MEF Lab - Ex2 Task2 Step8 - TailspinQueryMetadata VB)

**Visual Basic**

```
<QueryMetadata(Name:="Tailspin Motorsport",
   Description:="Life in the fast lane. You can't live without the best: Tailspin.",
   ImagePath:="Images/tailspin.jpg")>
Public Class TailspinQuery
   Inherits CarQueryBase
   // Removed code
   ...
End Class
```

9. You will repeat the same process on the WingtipQuery class. To do this, remove the constructor and replace the class definition in the **WingtipQuery** class.
Task 3 – Importing Metadata

In this task, you will modify the way import occurs to let importers access the metadata attached to the exports.

1. To access metadata in a strongly typed fashion create a metadata view by defining an interface with matching read only properties (names and types). To do this, right click on the ContosoAutomotive.Common project and select Add | New Item. In the Add New Item dialog select Interface type and name it IQueryMetadata.
2. Make the interface public and add the desired metadata properties as read only. To do this, replace the default interface implementation with the following code.
3. Since you are no longer using the Name, Description and ImagePath properties you can remove them from the CarQueryBase class. To do this, open the CarQueryBase.cs (C#) or CarQueryBase.vb (Visual Basic) and remove the Name, Description and ImagePath properties and their associated member fields.

4. Modify the main application to retrieve the query information from metadata instead of retrieving it from the query instance. To do this, open the CashMaker class by right clicking on the CashMaker.xaml file under the ContosoAutomotive project and select View Code in the contextual menu.

5. Now you can start importing using the type System.Lazy<T, TMetadata> where T is the contract type and TMetadata is the interface you have created. To do this, replace the CarQueries field definition inside the CashMaker class with the following code.
**Public Property** CarQueries As ObservableCollection(Of Lazy(Of ICarQuery, IQueryMetadata))

**Note:** As it was explained, Lazy type allows delaying the object instantiation until it is used, specifically, the object will be created when the **Value** property of the Lazy object will be called.

You are declaring the Lazy type using the contract type and the metadata type. The last one is not required but you are defining it because you want to access the metadata information avoiding the object instantiation.

6. Since you have modified the bound collection, you should adapt the SelectionChange event handler to deal with a Lazy Query object. To do this, replace the bolded lines of code below to use a proper approach to run queries.

*(Code Snippet – Intro to MEF Lab - Ex2 Task3 Step6 – CommandList Event CSharp)*

```csharp
private void commandList_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e) {
    if (this.SearchEnabled) {
        this.DisableSearch();
        var thread = new Thread(() => {
            if (e.AddedItems.Count > 0) {
                var lazyQuery = e.AddedItems[0] as Lazy<ICarQuery, IQueryMetadata>;
                if (lazyQuery != null) {
                    lazyQuery.Value.Run(this.cars, true);
                }
            }
        });
        this.EnableSearch();
    })
    thread.Start();
    thread.Join();
}
```
Visual Basic

Private Sub commandList_SelectionChanged(ByVal sender As System.Object, ByVal e As System.Windows.Controls.SelectionChangedEventArgs)
    If (Me.SearchEnabled) Then
        Me.DisableSearch()
        Dim thread = New Thread(Sub()
            If (e.AddedItems.Count > 0) Then
                Dim lazyQuery = TryCast(e.AddedItems(0), Lazy(Of ICarQuery, IQueryMetadata))
                If (lazyQuery IsNot Nothing) Then
                    lazyQuery.Value.Run(Me.cars, True)
                End If
            End If
        End Sub)
        thread.Start()
        thread.Join()
    End If
End Sub

7. Modify the UI XAML code as well to adapt the UI to the new lazy model. To do this, right click on the CashMaker.xaml file and select View Designer in the contextual menu. For simplicity, expand on the XAML tab to show only the markup code.

8. Scroll down to the <DataTemplate> element with the QueryItem key and adapt the Path property of the Binding objects to retrieve the Name, Description and ImagePath from the Metadata property of the Lazy objects. To do this, replace the highlighted bolded lines of markup code as shown below.

XAML

<DataTemplate x:Key="QueryItem">
    <Border Style="{StaticResource RoundedBorder}" Width="350" Height="60" Margin="0" Padding="0">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="60" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Image Grid.Column="0" VerticalAlignment="Center" HorizontalAlignment="Center" Width="45" Height="45" Source="{Binding Path=Metadata.ImagePath}"
                  Margin="0" Padding="1" Background="#E8ECED" />
        </Grid>
    </Border>
</DataTemplate>
Note: The Lazy type expose a property called Metadata, which is the metadata type defined on the Lazy type declaration (in this case, IQueryMetadata). This property is used in the XAML above to access the metadata information of the different query items instead of use the object properties as it was implemented in the previous exercise.

9. Scroll down to the <ContentControl> element at the bottom of the file and adapt the Path property of the Binding object to retrieve the Results object from the Value instance of the Lazy objects. To do this, replace the bolded lines of markup code as shown below.

XAML

<ContentControl Width="700" Margin="1 1 5 1" VerticalAlignment="Top">
   <Border Padding="1 1 5 1" Style="{StaticResource RoundedBorderRight}"
   Grid.Row="0">
      <DataGrid Name="Results" ItemsSource="{Binding Path=Value.Results}"
      AutoGenerateColumns="True" ColumnWidth="*" BorderThickness="0" Width="667"
      Height="381" />
   </Border>
</ContentControl>

Note: As it was explained for the Lazy’s Metadata property. The Value property matches the contract type on the lazy type definition (in this case, ICarQuery) and allows accessing the different object’s properties and methods.
Next Step

Exercise 2: Verification

In this verification, you will run the application to verify that all the queries are shown in the main application. In addition, you will verify the lazy behavior of the query objects.

1. In order to verify the lazy instantiation of an object you will add an empty constructor to any of the ICarQuery implementations and set a breakpoint on it to check when it is being instantiated. To do this, open the FabrikamQuery.cs (C#) or FabrikamQuery.vb (Visual Basic), add a new constructor to it and set a breakpoint by pressing F9 as shown in the following screenshot.

![FabrikamQuery constructor breakpoint (C#)](image1)

![FabrikamQuery constructor breakpoint (C#)](image2)

2. Compile the solution (CTRL+SHIFT+B).
3. Set ContosoAutomotive as the startup project. In the Solution Explorer, right-click ContosoAutomotive and select Set as startup project.
4. Press F5 to run the application. The parent CashMaker window should appear with a list of manufacturers on it. You should note the FabrikamQuery constructor has not been executed.

![Figure 14](contoso.png)
The CashMaker window shows manufacturers.

5. Click on the Fabrikam Motor Company icon on the list of manufacturers in the Cash Maker application. Now, the code will stop in the breakpoint you have set in the FabrikamQuery constructor, because the instance is being created on demand.

![Figure 15](fabrikamquery.cs)
FabrikamQuery constructor breakpoint (C#)
6. Click the close button (×) in the top right of the Cash Maker window to close it.

7. Remove the breakpoint and the constructor you just created in the FabrikamQuery class.

Next Step

Exercise 3: Using MEF and Silverlight 4

Exercise 3: Using MEF and Silverlight 4

Latest version of Silverlight 4 will include MEF out of the box. This means all the functionality described in the previous exercises are available for Silverlight 4 applications.

Some minor tweaks are required because of the different deployment model of a Silverlight application.

In this exercise, you will go through the creation of a Silverlight version of the Cash Maker application reusing most of the code and logic already present in the WPF version of the application.

Task 1 – Convert Extensions to Silverlight

In this task, you will convert all the Class Library projects used by the WPF version of the Cash Maker application to Silverlight. You will reuse all the code present in the WPF version by adding links to the existing files.


2. Open the ContosoAutomotive.sln solution file. By default, this file is located in the folder Source\Ex3\begin (choosing the folder that matches the language of your preference.) Optional, you can continue working the solution you created in the previous exercise.
3. Because a Silverlight 4 application can only reference Silverlight Class Libraries, you will replicate all the extension and contract definitions for Silverlight 4. You will start adding the Silverlight 4 version of Contoso.Automotive.Common library. To do this, right click on the ContosoAutomotive solution node and select Add | New Project. In the Add New Project dialog, select the Silverlight Class Library project type and enter ContosoAutomotive.Common.SL in the Name field.

![Add New Project](image)

**Figure 17**

*Add ContosoAutomotive.Common.SL Silverlight class library project (C#)*
4. In the New Silverlight Class Library options dialog accept the default settings. Ensure **Silverlight 4** is selected in the **Silverlight Version** list.

5. Only in **Visual Basic**, change the root namespace of the ContosoAutomotive.Common.SL project to **ContosoAutomotive.Common**. To do this, right click the **ContosoAutomotive.Common.SL**
project node and select Properties. In the Properties page, select the Silverlight tab and change the Root namespace field value to ContosoAutomotive.Common.

![Image](image.png)

**Figure 20**
*Change Root namespace to ContosoAutomotive.Common (Visual Basic)*

6. Remove the auto-generated class file. To do this, right click on the Class1.cs (C#) or the Class1.vb (Visual Basic) file and select Delete.

7. You will reuse all the code already present in the existing projects by linking the previous class files in the new projects. To do this, right click the ContosoAutomotive.Common.SL project node and select Add | Existing Item. In the Add Existing Item dialog browse one folder up and select all the class files inside the ContosoAutomotive.Common folder. Then, from the Open button drop-down list, select Add As Link.
Figure 21
Add linked files to the ContosoAutomotive.Common.SL project (C#)

Figure 22
8. After adding the link to the class files, your solution should look like the following image.

Figure 23
Linked files in the ContosoAutomotive.Common.SL project (C#)

Figure 24
Linked files in the ContosoAutomotive.Common.SL project (Visual Basic)
9. Repeat the same process for the ContosoAutomotive.Extensions project creating the ContosoAutomotive.Extensions.SL Silverlight Class Library project. To do this, right-click the ContosoAutomotive solution node and select Add | New Project. In the Add New Project dialog, select the Silverlight Class Library project type and enter ContosoAutomotive.Extensions.SL in the name field. Remember to select Silverlight 4 in the Silverlight Version list.

10. Add a reference to the MEF library on the ContosoAutomotive.Extensions.SL project. To do this:

   a. Select the ContosoAutomotive.Extensions.SL project in the Solution Explorer and select Project | Add Reference... The Add References dialog appears.

   b. Select the .NET tab and then select the System.ComponentModel.Composition component. Click the OK button to add a reference to this library.

![Add Reference](image)

Figure 1
Add a Reference to the MEF library.

11. Also, add a reference to the ContosoAutomotive.Common.SL library on the same project. To do this:

   a. Select the ContosoAutomotive.Extensions.SL project in the Solution Explorer and select Project | Add Reference... The Add References dialog appears.
b. Select the **Projects** tab and then select the **ContosoAutomotive.Common.SL** component. Click the **OK** button to add a reference to this library.

![Add Reference](image)

*Figure 1*
*Add a Reference to the ContosoAutomotive.Common.SL library.*

12. Remove the auto-generated class file **Class1.cs** (C#) or **Class1.vb** (Visual Basic) and create a link to all the class files in the **ContosoAutomotive.Extensions** folder.
Task 2 – Create Silverlight Cash Maker UI
In this task, you will create a new Silverlight application to reproduce the same user experience we found in the WPF Cash Maker application.

1. You will start adding a new Silverlight 4 application project to your solution. To do this, right-click on the ContosoAutomotive solution node and selecting Add | New project. In the Add New Project dialog, select Silverlight Application as the project type and enter ContosoAutomotive.Silverlight in the project name field.

![Adding Silverlight 4 project (C#)](image)
2. In the New Silverlight Application options dialog accept all defaults. Ensure **Silverlight 4** is selected in the **Silverlight Version** list.
3. Copy all the images files from the ContosoAutomotive WPF project into the Silverlight version. To do this, right click on the Images folder in the ContosoAutomotive project and select Copy. Then, right click the ContosoAutomotive.Silverlight project and select Paste.

4. Add a reference to the MEF library on the ContosoAutomotive.Silverlight project. You will include the Initialization assembly as well. To do this:

   a. Select the ContosoAutomotive.Silverlight project in the Solution Explorer and select Project | Add Reference... The Add References dialog appears.

   b. Select the .NET tab and then select the System.ComponentModel.Composition and the System.ComponentModel.Composition.Initialization components. Click the OK button to add these references to the project.

   a. Select the ContosoAutomotive.Silverlight project in the Solution Explorer and select Project | Add Reference… The Add References dialog appears.

   b. Select the .NET tab and then select the System.Windows.Controls.Data and the System.Windows.Controls.Data.Input components. Click the OK button to add a reference to this library.

6. Also, add a reference to the ContosoAutomotive.Common.SL and ContosoAutomotive.Extensions.SL the library on the ContosoAutomotive.Silverlight project. To do this, right click on the project and select Add Reference. In the Add Reference dialog, select the Projects tab and choose the ContosoAutomotive.Common.SL and ContosoAutomotive.Extensions.SL libraries.

7. Remove the default MainPage.xaml file in order to replace it with the Cash Maker UI. To do this, right click on the MainPage.xaml file and select Delete.

8. Include the prebuilt version of the Cash Maker UI located in the Assets folder. To do this, right click the ContosoAutomotive.Silverlight project node and select Add | Existing Item. In the Add Existing Item dialog, browse the Assets folder and selecting the folder that matches the language of your choice add the CashMaker.xaml file. The code behind file will be automatically included as well.
Figure 30
Add prebuilt Silverlight Cash Maker UI (C#)

Figure 31
Add prebuilt Silverlight Cash Maker UI (Visual Basic)

Note: The only differences between the version that you were using on the previous step and the recently added from the Assets folder are that the last one uses the Silverlight control
(instead of the WPF version of those), and does not have implemented some code that you will add in the following steps.

Task 3 – Update Your Silverlight Application to Load Composable Parts

In this task, you will modify a Silverlight 4 application to load composable parts using MEF. You will use the same approach used in the WPF version of the Cash Maker application.

1. Open the App class in Code View. To do this, right-click on the App.xaml file in the Solution explorer and select View Code.

2. Update the App class to use the MEF library. To do this, add the following statement on top of the using clause list above the App class definition.

   **C#**
   ```csharp
   using System.ComponentModel.Composition.Hosting;
   ```

   **Visual Basic**
   ```vbnet
   Imports System.ComponentModel.Composition.Hosting
   ```

3. You now will modify the Application class to load the new CashMaker UI. In addition, you will prepare you application to load any available Composable Part from the Silverlight package. To do this, replace the code inside the Application_Startup method with the following code.

   *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step3 - Application_Startup CSharp)*

   **C#**
   ```csharp
   private void Application_Startup(object sender, StartupEventArgs e)
   {
       var catalog = new AggregateCatalog(new DeploymentCatalog());
       var container = new CompositionContainer(catalog);

       this.RootVisual = container.GetExportedValue<CashMaker>();
   }
   ```

   *(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step3 - Application_Startup VB)*

   **Visual Basic**
   ```vbnet
   Private Sub Application_Startup(ByVal o As Object, ByVal e As StartupEventArgs) Handles Me.Startup
       Dim catalog = New AggregateCatalog(New DeploymentCatalog())
       Dim container = New CompositionContainer(catalog)

       Me.RootVisual = container.GetExportedValue(Of CashMaker)()
   End Sub
   ```
As you did for the WPF version, you create a container which discovers the composable parts through a Catalog. The main window is also retrieved using the GetExportedValue of the container.

Silverlight applications run and are deployed in a different way that WPF applications, it causes that there is a new catalog on MEF Silverlight version. This exercise uses an Aggregate Catalog (which is also available on the WPF version) including the new Deployment Catalog which download a Silverlight Package (.xap), if it was not downloaded before, and searches for composable parts inside it. As the Deployment Catalog is instantiated without any parameter, it will look for parts inside the Silverlight Package that contains the application.

4. Since the CashMaker class is now a composable part, you will export the class making it available to the composition container. To do this, decorate the CashMaker class with the Export attribute as shown in the code below.

C# :
```csharp
[Export]
public partial class CashMaker : UserControl,
IPartImportsSatisfiedNotification
{
...
}
```

Visual Basic :
```vbnet
<Export()>
Class CashMaker
    Inherits UserControl
    Implements IPartImportsSatisfiedNotification
...
End Class
```

5. Add the collection of Lazy object containing the contract implementations that will be populated by MEF. To do this, add the following code to the CashMaker.xaml.cs (C#) or CashMaker.xaml.vb (Visual Basic) file, just above its constructor.

(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - SilverlightCarQueries CSharp)

C# :
```csharp
[ImportMany(AllowRecomposition = true)]
```
public ObservableCollection<Lazy<ICarQuery, IQueryMetadata>> CarQueries { get; set; }

(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - SilverlightCarQueries VB)

Visual Basic

<ImportMany(AllowRecomposition:=True)>
Public Property CarQueries As ObservableCollection(Of Lazy(Of ICarQuery, IQueryMetadata))

6. Modify the implementation of the IPartImportsSatisfiedNotification interface and bind the CarQueries collection to the UI by setting the DataContext property of the commandGrid control. To do this, paste the following code inside the CashMaker's OnImportsSatisfied method.

(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - OnImportsSatisfied CSharp)

C#

public void OnImportsSatisfied()
{
    this.commandGrid.DataContext = this.CarQueries;
}

(Code Snippet – Intro to MEF Lab - Ex3 Task2 Step5 - OnImportsSatisfied VB)

Visual Basic

Public Sub OnImportsSatisfied() Implements IPartImportsSatisfiedNotification.OnImportsSatisfied
    Me.commandGrid.DataContext = Me.CarQueries
End Sub

Task 4 – Load Silverlight Extensions Dynamically

In this task, you will create a new Silverlight 4 application and setup your application to load MEF parts dynamically by downloading a new Silverlight package. Additionally, you can mix CLR languages and create the new project in different CLR language than your main application.

1. Create the ContosoAutomotive.Woodgrove.SL project reusing the classes in the ContosoAutomotive.Woodgrove Class Library project selecting a different language for the project than the main application. To do this, right click the ContosoAutomotive solution node and select Add | New Project. In the Add New Project dialog, select the Silverlight Application project type and enter ContosoAutomotive.Woodgrove.SL in the name field.

2. In the New Silverlight Application options dialog select the existing ContosoAutomotive.Silverlight.Web project to host the Silverlight application and ensure
Silverlight 4 is selected in the Silverlight Version list. Also, remove the Add a test page option in the option section and press OK.

![New Silverlight Application options dialog](image)

**Figure 32**
New Silverlight Application options dialog

**Note:** Remember that Silverlight Application projects are packaged into Silverlight Packages (.xap). In the other hand, Silverlight Class Library projects are included inside the application package that references the library. For that reason, you create the ContosoAutomotive.Woodgrove.SL project as a Silverlight application to be able to deploy it in a separated way allowing download it while the Silverlight Main application is running.

3. After creating the project remove the auto-generated files App.xaml and MainPage.xaml and create a link to the WoodgroveQuery class files in the ContosoAutomotive.Woodgrove folder.

4. Add a reference to the MEF library on the ContosoAutomotive.Woodgrove.SL project. To do this, right click on the project and select Add Reference. In the Add Reference dialog, select the .Net tab and choose the System.ComponentModel.Composition library.

5. Add a reference to the ContosoAutomotive.Common.SL library on the ContosoAutomotive.Woodgrove.SL project. To do this, right click on the project and select Add
Reference. In the **Add Reference** dialog, select the **Projects** tab and choose the **ContosoAutomotive.Common.SL** library.

6. Now, your solution is prepared for creating a Silverlight 4 version of the Cash Maker application.

![Solution Explorer](image1.png)

**Figure 33**
All projects converted to Silverlight Class Library projects (**C#**)

![Solution Explorer](image2.png)

**Figure 34**
All projects converted to Silverlight Class Library projects (**Visual Basic**)
7. Modify the ContosoAutomotive.Silverlight application class code to create a catalog to download the new Silverlight package. To do this, replace the catalog definition code inside the **Application_Startup** method with the following.

*(Code Snippet – Intro to MEF Lab - Ex3 Task4 Step7 - UriDeploymentCatalog CSharp)*

```csharp
private void Application_Startup(object sender, StartupEventArgs e)
{
    var xapCatalog = new DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap");
    xapCatalog.DownloadAsync();
    var catalog = new AggregateCatalog(new DeploymentCatalog(), xapCatalog);
    var container = new CompositionContainer(catalog);
    this.RootVisual = container.GetExportedValue<CashMaker>();
}
```

*(Code Snippet – Intro to MEF Lab - Ex3 Task4 Step7 - UriDeploymentCatalog VB)*

```vbnet
Private Sub Application_Startup(ByVal o As Object, ByVal e As StartupEventArgs) Handles Me.Startup
    Dim xapCatalog = New DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap")
    xapCatalog.DownloadAsync()
    Dim catalog = New AggregateCatalog(New DeploymentCatalog(), xapCatalog)
    Dim container = New CompositionContainer(catalog)
    Me.RootVisual = container.GetExportedValue(Of CashMaker)()
End Sub
```

**Note:** In the code above, a new Deployment Catalog is included to the Aggregate Catalog. Because it is instantiated specifying a url where a Silverlight Package (.xap) is available, it will be in charge of downloading it and find the composable parts inside it.

The call to the DownloadAsync method of the DeploymentCatalog object forces to download the package (it is performed in a separated thread). Take into account that the package will not be downloaded until this method is called.

---

**Next Step**

*Exercise 3: Verification*
Exercise 3: Verification

In this verification, you will run the application to verify that all the queries are shown in the main application. The Woodgrove manufacturer will be loaded from a different Silverlight package downloaded from the website.

1. Compile the solution (CTRL+SHIFT+B).

2. Expand the ClientBin folder under the ContosoAutomotive.Silverlight.Web project and verify that there are two Silverlight packages present.

Figure 35
Silverlight packages in the ContosoAutomotive.Silverlight.Web project (C#)
3. You will modify the way the application loads the Woodgrove package to emphasize the fact it is being downloaded from the website. Open the `App` class in the `ContosoAutomotive.Silverlight` project and modify the `Application_Startup` method including the bolded line of code below.

(Code Snippet – Intro to MEF Lab - Ex3 Verification Step3 - DownloadCompleteAddHandler CSharp)

```csharp
private void Application_Startup(object sender, StartupEventArgs e)
{
    var xapCatalog = new DeploymentCatalog("/ClientBin/ContosoAutomotive.Woodgrove.SL.xap");

    xapCatalog.DownloadCompleted += new EventHandler<System.ComponentModel.AsyncCompletedEventArgs>(xapCatalog_DownloadCompleted);
    xapCatalog.DownloadAsync();
}
```
```csharp
var catalog = new AggregateCatalog(new DeploymentCatalog(), xapCatalog);
var container = new CompositionContainer(catalog);
this.RootVisual = container.GetExportedValue<CashMaker>();
}
```

4. Add the `xapCatalog_DownloadCompleted` method handler to increment in two seconds the delay experienced by downloading a new package from internet. In the `App` class, add the following methods below the `Application_Startup` method.

```csharp
private void xapCatalog_DownloadCompleted(object sender, System.ComponentModel.AsyncCompletedEventArgs e)
{
    System.Threading.Thread.Sleep(2000);
}
```

```vbnet
Private Sub xapCatalog_DownloadCompleted(ByVal sender As Object, ByVal e As System.ComponentModel.AsyncCompletedEventArgs)
End Sub
```
5. Set **ContosoAutomotive.Silverlight.Web** as the startup project. To do this, in the Solution Explorer, right-click **ContosoAutomotive.Silverlight.Web** and select **Set as Startup Project**.

6. Press F5 to run the application. The CashMaker application should appear with a list of manufacturers on it. You should note that the Woodgrove manufacturer item would appear after some delay.

![Contoso Automotive Resellers](image)

**Figure 37**  
*The CashMaker Silverlight application shows manufacturers.*

7. Close the browser by clicking on the close button (×).
In this lab you have used the Microsoft Managed Extensibility Framework to add extensibility points to an application and build extensions to plug into those extensibility points.

Using MEF, you set exported properties, loaded external assemblies without explicit references and dynamically extended your application at runtime. You have seen the interaction of Imports and Exports, and used Contracts via Interfaces to specify what Export components to deal with.

In addition, you created a Silverlight 4 application and enabled loading composable parts using MEF.