Contents

You can gain finer-grained control over the discovery process by using type and metadata filters with a composition **context**.

The CompositionContext

You may have noticed a *CompositionContext* argument can be passed into the *EntityManager* constructor, and the probe messages in your log all mention it too:

```
CompositionContext: '-IbDefault-' - Probed for any 'EntityServerQueryInterceptor' and found
'ModelLib.QueryInterceptor'.
```

Just what is this, and why does it keep showing up everywhere? In the discovery topic we discussed how DevForce uses MEF to discover the extensible components of your application, but what we didn't mention is that this discovery is contextual. The default context is just that, a default, and always used when you haven't specified a context. The *CompositionContext.Default* provides no special type or metadata filtering.

When and why to use a CompositionContext?

The CompositionContext is used in DevForce's built-in support for faking, but it's also useful for creating mocks during testing, and in any situation in which you might want several different custom implementations of an interface or base class. For example, you might have a requirement that in most cases you'll use an *EntityServerQueryInterceptor* in a certain way, but in particular situations you'd like different interceptor logic. With a custom *CompositionContext* you can easily define that "particular situation", and define a custom *EntityServerQueryInterceptor* to be used only for it.

Most, but not all, of the DevForce extensible components support a custom context. Those that do not are ones needed during application initialization and not as part of *EntityManager* activities. For example, custom loggers and components controlling service or proxy configuration cannot be discovered or composed contextually.

You can define and use any number of *CompositionContexts* within your application. As we'll see below it's generally the *EntityManager*, via its constructor, which determines the context in use.

Creating a custom CompositionContext

You can implement the *ICompositionContextResolver* interface or extend the *BaseCompositionContextResolver* to define and resolve your custom *CompositionContexts*. The *BaseCompositionContextResolver* will automatically register any statically defined composition contexts found within the sub-class, but you can also override its *GetCompositionContext* method to programmatically define new contexts.

Here's a sample resolver defining two custom contexts, "MockQuery" and "Admin":

```
public class MyCompositionContextResolver : BaseCompositionContextResolver {
static MyCompositionContextResolver() { }
public static CompositionContext MockQuery = CompositionContext.Fake
    .WithGenerator(typeof(MockEntityServerQueryInterceptor))
    .WithName("MockQuery");
public static CompositionContext Admin = CompositionContext.Default
  .WithGenerator(typeof(AdminEntityServerSaveInterceptor))
  .WithGenerator(typeof(AdminEntityServerQueryInterceptor))
  .WithName("Admin");
}
```

```
Public Class MyCompositionContextResolver
 Inherits BaseCompositionContextResolver
 Shared Sub New()
 End Sub
 Public Shared MockQuery As CompositionContext = _
    CompositionContext.Fake. _
    WithGenerator(GetType(MockEntityServerQueryInterceptor)). _
```

```
    WithName("MockQuery")
  Public Shared Admin As CompositionContext = _
    CompositionContext.Default. _
    WithGenerator(GetType(AdminEntityServerSaveInterceptor)). _
    WithGenerator(GetType(AdminEntityServerQueryInterceptor)). _
    WithName("Admin")
End Class
```

Note the static class constructor here: it ensures the static fields defining your custom resolvers will be initialized properly in release builds.

We also see above that these custom contexts are defined from existing *built-in* contexts. Once created a *CompositionContext* is immutable, but you can easily create new contexts based on an existing context and supply different composition criteria. Above we see a new context based on the *Fake* context but using its own custom *EntityServerQueryInterceptor*, and another context based on the *Default* context but using custom query and save interceptors. The sample above is also defining the extensible types specific to the context (although we didn't show the code for them).

In defining a custom composition context you must both register the context and define the extensible components it will use. There are two mechanisms used to accomplish this: one allows you to specify the types to be used when you create the context, while the other allows you to define metadata filters to be applied during discovery.

Discovery with types specified

This approach is the one you'll see most often in discovery examples, and was also used in the sample above. Here you specify the *generators* or the types to be used by the custom context. Remember to do two things:

1. Use the *WithGenerator* method to register types to the context.
2. Mark these types with the System.ComponentModel.Composition.PartNotDiscoverable attribute. This is required to hide the types from standard MEF discovery, since here DevForce is preempting that discovery.

Here's a sample resolver defining the composition contexts. Instead of defining static fields we're overriding the *GetCompositionContext* method to define the contexts at run time.

```
public class ContextResolver : BaseCompositionContextResolver {
  public override CompositionContext GetCompositionContext(string compositionContextName) {
    if (compositionContextName == "Test") {
      return CompositionContext.Default
        .WithGenerator(new Type[] { typeof(TestLoginManager), typeof(TestQueryInterceptor) })
        .WithName("Test");
    } else if (compositionContextName == "Dev") {
      return CompositionContext.Default
        .WithGenerator(new Type[] { typeof(DevLoginManager) })
        .WithName("Dev");
    } else return base.GetCompositionContext(compositionContextName);
  }
}
```

```
Public Class ContextResolver
 Inherits BaseCompositionContextResolver
 Public Overrides Function GetCompositionContext(ByVal compositionContextName As String) As CompositionContext
  If compositionContextName = "Test" Then
    Return CompositionContext.Default.WithGenerator(New Type() {GetType(TestLoginManager),
GetType(TestQueryInterceptor)}).WithName("Test")
  ElseIf (compositionContextName = "Dev") Then
    Return CompositionContext.Default.WithGenerator(New Type() { GetType(DevLoginManager)}).WithName("Dev"); }
  Else
    Return MyBase.GetCompositionContext(compositionContextName)
  End If
 End Function
End Class
```

Remember that your resolver may need to be defined on both client and server, depending on the type generators it defines and where they will be used. For example, an *IEntityLoginManager* is used only on the server.

Be sure to decorate the types used by your custom contexts with the MEF *PartNotDiscoverable* attribute. This tells both MEF and DevForce to ignore the types during standard discovery.

Below are a few very simple samples of custom types for the contexts defined above.

```
[PartNotDiscoverable]
public class DevLoginManager : IEntityLoginManager {
 public IPrincipal Login(ILoginCredential credential, EntityManager entityManager) {
```

```
  return new UserBase(new UserIdentity("Dev", "Custom", true), new string[] { "Dev" });
  }
 public void Logout(IPrincipal principal, EntityManager entityManager) {
  }
}
[PartNotDiscoverable]
public class TestLoginManager : IEntityLoginManager {
 public IPrincipal Login(ILoginCredential credential, EntityManager entityManager) {
  return new UserBase(new UserIdentity("Test", "Custom", true), new string[] { "Test" });
  }
 public void Logout(IPrincipal principal, EntityManager entityManager) {
  }
}
[PartNotDiscoverable]
public class TestQueryInterceptor : EntityServerQueryInterceptor {
 protected override bool ExecuteQuery() {
   TraceFns.WriteLine("User " + this.Principal.Identity.Name + " is executing query " + this.Query.ToString());
  return base.ExecuteQuery();
  }
}
```

```
<PartNotDiscoverable>
Public Class DevLoginManager
Implements IEntityLoginManager
Public Function Login(ByVal credential As ILoginCredential, ByVal entityManager As EntityManager) As IPrincipal
  Return New UserBase(New UserIdentity("Dev", "Custom", True), New String() { "Dev" })
End Function
Public Sub Logout(ByVal principal As IPrincipal, ByVal entityManager As EntityManager)
End Sub
End Class
<PartNotDiscoverable>
Public Class TestLoginManager
Implements IEntityLoginManager
Public Function Login(ByVal credential As ILoginCredential, ByVal entityManager As EntityManager) As IPrincipal
  Return New UserBase(New UserIdentity("Test", "Custom", True), New String() { "Test" })
End Function
Public Sub Logout(ByVal principal As IPrincipal, ByVal entityManager As EntityManager)
End Sub
End Class
<PartNotDiscoverable>
Public Class TestQueryInterceptor
Inherits EntityServerQueryInterceptor
Protected Overrides Function ExecuteQuery() As Boolean
TraceFns.WriteLine("User " & Me.Principal.Identity.Name & " is executing query " & Me.Query.ToString())
  Return MyBase.ExecuteQuery()
End Function
End Class
```

Discovery with metadata filters

Instead of defining the types to be used with the custom context when you create the context you can instead define metadata filtering rules. Here you'll do the following:

1. Build one or more metadata filters with *BuildExportFilter*.
2. Registers the filters with the context using *WithFilter* and *WithTypeFilter* methods.
3. Mark your types with the System.ComponentModel.Composition.ExportMetadata and System.ComponentModel.Composition.InheritedExport attributes.

Here's a sample resolver defining the composition contexts. As above we overrode the *GetCompositionContext* method to define the contexts at run time. The *BuildExportFilter* method is used to build a filter for the metadata attributes to be used. Here we'll be using only a single metadata attribute named "Env". We're also using the *WithFilter* method to indicate that any types meeting the metadata criteria will be used. We could also use the *WithTypeFilter* to indicate specific metadata by type.

```
public class ContextResolver : BaseCompositionContextResolver {
 public override CompositionContext GetCompositionContext(string compositionContextName) {
  if (compositionContextName == "Test") {
    var filter = CompositionContext.BuildExportFilter("Env", "Test");
   return CompositionContext.Default
     .WithFilter(filter)
     .WithName("Test");
  } else if (compositionContextName == "Dev") {
```

```
    var filter = CompositionContext.BuildExportFilter("Env", "Dev");
    return CompositionContext.Default
      .WithFilter(filter)
      .WithName("Dev");
   } else return base.GetCompositionContext(compositionContextName);
  }
}
```

```
Public Class ContextResolver
Inherits BaseCompositionContextResolver
Public Overrides Function GetCompositionContext(ByVal compositionContextName As String) As CompositionContext
  If compositionContextName = "Test" Then
 Dim filter = CompositionContext.BuildExportFilter("Env", "Test")
Return CompositionContext.Default.WithFilter(filter).WithName("Test")
ElseIf compositionContextName = "Dev" Then
 Dim filter = CompositionContext.BuildExportFilter("Env", "Dev")
Return CompositionContext.Default.WithFilter(filter).WithName("Dev")
Else
Return MyBase.GetCompositionContext(compositionContextName)
End If
End Function
End Class
```

We must decorate our types to add the metadata wanted. This requires both the *ExportMetadata* attribute, to add metadata to the type, and the *InheritedExport* attribute, which is required by MEF to ensure that the custom metadata is found.

Here are the same simple classes as defined above, but this time using metadata.

```
[InheritedExport(typeof(IEntityLoginManager))]
[ExportMetadata("Env", "Dev")]
public class DevLoginManager : IEntityLoginManager {
 public IPrincipal Login(ILoginCredential credential, EntityManager entityManager) {
  return new UserBase(new UserIdentity("Dev", "Custom", true), new string[] { "Dev" });
  }
 public void Logout(IPrincipal principal, EntityManager entityManager) {
  }
}
[InheritedExport(typeof(IEntityLoginManager))]
[ExportMetadata("Env", "Test")]
public class TestLoginManager : IEntityLoginManager {
 public IPrincipal Login(ILoginCredential credential, EntityManager entityManager) {
  return new UserBase(new UserIdentity("Test", "Custom", true), new string[] { "Test" });
  }
 public void Logout(IPrincipal principal, EntityManager entityManager) {
  }
}
[InheritedExport(typeof(EntityServerQueryInterceptor))]
[ExportMetadata("Env", "Test")]
public class TestQueryInterceptor : EntityServerQueryInterceptor {
 protected override bool ExecuteQuery() {
   TraceFns.WriteLine("User " + this.Principal.Identity.Name + " is executing query " + this.Query.ToString());
  return base.ExecuteQuery();
  }
}
```

```
<InheritedExport(GetType(IEntityLoginManager)), ExportMetadata("Env", "Dev")>
Public Class DevLoginManager
Implements IEntityLoginManager
Public Function Login(ByVal credential As ILoginCredential, ByVal entityManager As EntityManager) As IPrincipal
  Return New UserBase(New UserIdentity("Dev", "Custom", True), New String() { "Dev" })
End Function
Public Sub Logout(ByVal principal As IPrincipal, ByVal entityManager As EntityManager)
End Sub
End Class
<InheritedExport(GetType(IEntityLoginManager)), ExportMetadata("Env", "Test")>
Public Class TestLoginManager
Implements IEntityLoginManager
Public Function Login(ByVal credential As ILoginCredential, _
  ByVal entityManager As EntityManager) As IPrincipal
  Return New UserBase(New UserIdentity("Test", "Custom", True), New String() { "Test" })
End Function
Public Sub Logout(ByVal principal As IPrincipal, ByVal entityManager As EntityManager)
End Sub
```

```
End Class
<InheritedExport(GetType(EntityServerQueryInterceptor)), ExportMetadata("Env", "Test")>
Public Class TestQueryInterceptor
Inherits EntityServerQueryInterceptor
Protected Overrides Function ExecuteQuery() As Boolean
  TraceFns.WriteLine("User " & Me.Principal.Identity.Name & " is executing query " & _
   Me.Query.ToString())
 Return MyBase.ExecuteQuery()
End Function
End Class
```

Using a CompositionContext

The *CompositionContext* is usually specified by name during the construction of an *EntityManager*:

```
var em = new EntityManager(compositionContextName: MyCompositionContextResolver.MockQuery.Name);
```

```
Dim em As New EntityManager(compositionContextName:=MyCompositionContextResolver.MockQuery.Name)
```

We saw earlier that every custom *CompositionContext* was also given a unique name, via the *WithName* method. It's this name which is provided in the *EntityManager* constructor. The name defines the context to be used for all composition performed during the login, query, save, and other operations performed by the *EntityManager*, and the *EntityServer* it's connected to. In a few types, such as the *VerifierEngine*, the *CompositionContext* may also be set explicitly. This is because these classes may be instantiated and used independently of any *EntityManager*.

If the *CompositionContext* name isn't provided to an *EntityManager* the *CompositionContext.Default* will be used.

Changing the Default CompositionContext

You can use type or metadata filtering with the *Default* context too. This is particularly useful when you're also using custom contexts but also need to ensure the behavior of the *Default* context.

To modify the *Default* context you'll create a new *CompositionContext* based on *CompositionContext.Default*, and assign it the same name as the *Default*. Note this is a bit of a trick: the *CompositionContext.Default* will remain untouched, but by taking its name you will effectively preempt use of the prior one.

For example, here we add a custom *EntityServerQueryInterceptor* to the *Default* context. Note the *WithName* clause: it ensures the *Default* name is retained.

```
public class TestCompositionContextResolver : BaseCompositionContextResolver {
 static TestCompositionContextResolver() { }
 public static CompositionContext MyDefault = CompositionContext.Default
      .WithGenerator(typeof(MyDefaultEntityServerQueryInterceptor))
      .WithName(CompositionContext.Default.Name);
}
```

Here's the same thing using a metadata filter:

```
public class TestCompositionContextResolver : BaseCompositionContextResolver {
 static TestCompositionContextResolver() { }
 public static CompositionContext MyDefault = CompositionContext.Default
      .WithFilter(CompositionContext.BuildExportFilter("Env", "Default"))
      .WithName(CompositionContext.Default.Name);
}
```

Custom fakes and mocks

Custom composition contexts are particularly useful in creating custom fakes and mocks for testing. The built-in support for faking is discussed separately in more detail, but here we show how to create custom contexts to aid in testing.

Let's assume we want to test how well our client code copes with a query exception thrown on the server. You generally don't want to force a query exception by modifying your production code, but you can test this easily with a mock.

```
public class TestCompositionContextResolver : BaseCompositionContextResolver {
 static TestCompositionContextResolver() { }
 ///<summary>
 /// A version of the DevForce FakeCompositionContext
 /// extended by a custom EntityServerQueryInterceptor
 ///</summary>
```

```
 public static CompositionContext FailingQueryContext = CompositionContext.Fake
      .WithGenerator(typeof(FailingEntityServerQueryInterceptor));
}
///<summary>
/// An EntityServerQueryInterceptor that throws an
/// exception no matter what the query.
///</summary>
[PartNotDiscoverable]
public class FailingEntityServerQueryInterceptor : EntityServerQueryInterceptor {
 protected override bool ExecuteQuery() {
  throw new InvalidOperationException();
  }
}
```

```
Public Class TestCompositionContextResolver
 Inherits BaseCompositionContextResolver
 Shared Sub New()
 End Sub
 '''<summary>
 ''' A version of the DevForce FakeCompositionContext
 ''' extended by a custom EntityServerQueryInterceptor
 '''</summary>
 Public Shared FailingQueryContext As CompositionContext = _
    CompositionContext.Fake.WithGenerator(GetType(FailingEntityServerQueryInterceptor))
End Class
'''<summary>
''' An EntityServerQueryInterceptor that throws an
''' exception no matter what the query
'''</summary>
<PartNotDiscoverable()> _
Public Class FailingEntityServerQueryInterceptor
 Inherits EntityServerQueryInterceptor
 Protected Overrides Function ExecuteQuery() As Boolean
   Throw New InvalidOperationException()
 End Function
End Class
```

Here we've written a *FailingEntityServerQueryInterceptor* that always throws an exception no matter what the query. Note the *PartNotDiscoverable* attribute on the custom class. This tells MEF to skip this class when discovering *EntityServerQueryInterceptors*. We don't want this one popping up in our production application!

Now that we've created a custom context that incorporates our interceptor, let's see how it affects a client-side query.

```
[TestMethod]
public void MockQueryFail() {
  var em = new DomainModelEntityManager(compositionContextName:
     TestCompositionContextResolver.FailingQueryContext.Name);

  var query = em.Customers.Where(c => c.CompanyName.StartsWith("S"));
  query.ToList();
 // Yep, this fails, better think about some try/catch logic.
}
```

```
<TestMethod> _
Public Sub MockQueryFail()
  Dim em As New DomainModelEntityManager(compositionContextName :=
          TestCompositionContextResolver.FailingQueryContext.Name)
  Dim query = em.Customers.Where(Function(c) c.CompanyName.StartsWith("S"))
   query.ToList()
 ' Yep, this fails, better think about some try/catch logic.
End Sub
```