Contents

Punch is not one monolithic framework, but rather lots of little pieces of functionality that are composed together when the application starts up. Configuration of this composition, including customizing any of the pieces, is done via the bootstrapper.

Creating a bootstrapper

A bootstrapper in Punch is created by extending the generic class CocktailMefBootstrapper<TRootModel>.

The bootstrapper requires the type of the root ViewModel, so it can instantiate and activate it once the configuration steps have finished. The root ViewModel must be annotated with the proper MEF export attribute in order for the bootstrapper to create an instance through MEF.

```
public class AppBootstrapper : CocktailMefBootstrapper<ShellViewModel>
{
  // Add additional logic if required.
}
```

For the bootstrapper to come to live, it must be added as a static resource to your application's App.xaml.

Note: In WPF, the static resource must be contained in a ResourceDictionary.

Silverlight:

```
<Application x:Class="TempHire.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:TempHire">
    <Application.Resources>
        <local:AppBootstrapper x:Key="AppBootstrapper" />
    </Application.Resources>
</Application>
```

WPF:

```
<Application x:Class="TempHire.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:TempHire">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary>
                    <local:AppBootstrapper x:Key="AppBootstrapper" />
                </ResourceDictionary>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Runtime configuration

The StartRuntime method of the CocktailMefBootstrapper provides the means to perform synchronous application configuration during startup of the application. This is the place where you as an application developer can put your own configuration logic.

The following code shows an example of such a configuration. It initializes the DevForce Fake Backing Store in a WPF application during startup.

```
public class AppBootstrapper : CocktailMefBootstrapper<ShellViewModel>
{
   [Import]
  public IEntityManagerProvider<TempHireEntities> EntityManagerProvider;
  protected override void StartRuntime()
   {
     base.StartRuntime();
      EntityManagerProvider.InitializeFakeBackingStore();
   }
}
```

DesignTime configuration

Whenever you open a XAML file in the Visual Studio Designer or Blend, the application starts in design-mode. At design time, the Punch bootstrapper skips all the runtime configuration, because at design time there are only certain things that can be done. For example you cannot connect to a server and fetch data while your application is executing inside the Visual Studio Designer or Blend.

Occasionally the need arises to perform some configuration steps specifically for when the application runs in design-mode. The StartDesignTime method provides the means to perform this configuration.

```
public class AppBootstrapper : CocktailMefBootstrapper<ShellViewModel>
{
  protected override void StartDesignTime()
   {
     base.StartDesignTime();
     // Additional configuration only executed at design time.
   }
}
```

Asynchronous configuration

Certain configuration steps can only be executed asynchronously. In particular in Silverlight, where all server communication must be asynchronous. The CocktailMefBootstrapper allows for such asynchronous configuration. The bootstrapper automatically waits for the asynchronous configuration to complete before continuing the application startup sequence.

The StartRuntimeAsync method provides the means to perform asynchronous configuration.

Note that asynchronous configuration is only possible at runtime. There's no equivalent for asynchronous configuration at design time.

Examples for asynchronous configuration steps are eager loading certain XAP files, initializing the DevForce Fake Backing Store in Silverlight as opposed to WPF, where that same task can be performed synchronously, etc.

The following code demonstrates how to initialize the DevForce Fake Backing Store in Silverlight during startup.

```
public class AppBootstrapper : CocktailMefBootstrapper<ShellViewModel>
{
   [Import]
  public IEntityManagerProvider<TempHireEntities> EntityManagerProvider;
  protected override async Task StartRuntimeAsync()
   {
      await EntityManagerProvider.InitializeFakeBackingStoreAsync();
   }
}
```

Customizing the MEF catalog

Punch uses MEF as the Inversion-of-Control container. The CocktailMefBootstrapper configures the container with a default parts catalog that is obtained from DevForce. This catalog contains all assemblies that were discovered during DevForce's Discovery phase.

The PrepareCompositionCatalog method allows for the developer to add to the default catalog or completely replace it with their own.

The following example demonstrates how to wrap the default catalog in an InterceptingCatalog available from MefContrib, for the purpose of centrally subscribing new instances to the EventAggregator.

```
public class AppBootstrapper : CocktailMefBootstrapper<ShellViewModel>, IExportedValueInterceptor
{
  object IExportedValueInterceptor.Intercept(object value)
  {
     SubscribeToEventAggregator(value);
    return value;
  }
  protected override void BuildUp(object instance)
  {
    base.BuildUp(instance);
     SubscribeToEventAggregator(instance);
  }
  // Use InterceptingCatalog from MefContrib to centrally handle EventAggregator subscriptions.
  protected override ComposablePartCatalog PrepareCompositionCatalog()
  {
     var cfg = new InterceptionConfiguration().AddInterceptor(this);
    return new InterceptingCatalog(base.PrepareCompositionCatalog(), cfg);
  }
  private void SubscribeToEventAggregator(object instance)
  {
    if (instance is IHandle)
    {
       LogFns.DebugWriteLine(string.Format("Automatically subscribing instance of {0} to EventAggregator.",
                                           instance.GetType().Name));
       EventFns.Subscribe(instance);
    }
  }
}
```

Customizing the MEF CompositionContainer

The CocktailMefBootstrapper automatically sets up an application scoped MEF container properly configured to work out of the box. It initializes the container with the default parts catalog obtained from DevForce or the custom catalog provided through the above PrepareCompositionCatalog method. In addition, it ensures that the container gets populated with the default EventAggregator, WindowManager and DialogManager.

Replacing the default EventAggregator, WindowManager or DialogManager

The CocktailMefBootstrapper always ensures that the container contains an EventAggregator, WindowManager and DialogManager. The EventAggregator provides a publish/subscribe service to loosely communicate between different components within the application. The WindowManager is responsible for displaying new windows and popups and the Punch DialogManager provides a convenient way to handle user prompts and messages.

As the application developer you may choose to replace the default implementation of any or all of these services. To replace any of these services, simply provide your own implementation and ensure that it will get discovered by annotating it with an ExportAttribute.

The following example demonstrates how to replace the EventAggregator with your own implementation.

```
[Export(typeof(IEventAggregator))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class MyEventAggregator : IEventAggregator
{
  public void Subscribe(object instance)
  {
    // Custom implementation
  }
  public void Unsubscribe(object instance)
  {
    // Custom implementation
  }
  public void Publish(object message)
  {
    // Custom implementation
  }
  public void Publish(object message, Action<Action> marshal)
  {
```

```
    // Custom implementation
  }
  public Action<Action> PublicationThreadMarshaller { get; set; }
}
```

Windows Store apps don't have WindowManager and DialogManager. The design language for Windows Store apps has no notion of popups other than the standard MessageDialog. In addition, replacing the EventAggregator is currently not supported.

Registering additional objects with the container

On top of replacing any of the default services by implementing and exporting your own, the PrepareCompositionContainer method allows for the developer to manually register objects with the container through the provided CompositionBatch

The following example demonstrates how to add the Punch AuthenticationService to the container.

```
public class AppBootstrapper : CocktailMefBootstrapper<ShellViewModel>
{
  protected override void PrepareCompositionContainer(CompositionBatch batch)
  {
    base.PrepareCompositionContainer(batch);
    batch.AddExportedValue<IAuthenticationService>(new AuthenticationService());
  }
}
```

Differences in Windows Store apps

The application model for Windows Store apps is quite different from WPF and Silverlight. A Windows Store app can have more than one entry point and can be suspended, resumed or terminated by the Operating System at will. Because of the nature of the activation model of these apps, a static Bootstrapper resource is not possible. Instead we have to subclass the Application class. The Application class provides the hooks to handle the multiple entry points as well as suspending and resuming.

To bootstrap a Windows Store app we use the CocktailMefWindowsStoreApplication class as the base for the Application object. Out-of-the-box it handles configuration of Punch just like CocktailMefBootstrapper. It provides many of the same familiar methods. In addition, it handles the normal entry point when the user launches the application by touching a tile. Other entry points such as if your application implements the search contract must be implemented manually.

We must do two things in order to build a Windows Store app using Punch. First we need to modify App.xaml.

```
<cocktail:CocktailMefWindowsStoreApplication
  xmlns:cocktail="using:Cocktail" x:Class="Todo.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <!--
            Styles that define common aspects of the platform look and feel
            Required by Visual Studio project and item templates
         -->
        <ResourceDictionary Source="Common/StandardStyles.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</cocktail:CocktailMefWindowsStoreApplication>
```

Then we modify App.xaml.cs accordingly.

```
sealed partial class App : CocktailMefWindowsStoreApplication
{
  public App() : base(typeof (MainPageViewModel))
  {
    InitializeComponent();
  }
  protected override void StartRuntime()
  {
    base.StartRuntime();
    // Alternatively we can provide this information via an app.config embedded as a resource
    IdeaBladeConfig.Instance.ObjectServer.RemoteBaseUrl = "http://localhost";
    IdeaBladeConfig.Instance.ObjectServer.ServerPort = 55123;
    IdeaBladeConfig.Instance.ObjectServer.ServiceName = "EntityService.svc";
  }
```

```
}
```

We provide the type of the main ViewModel to the base constructor. CocktailMefWindowsStoreApplication handles the launching of the main ViewModel and associated Page when the user launches the application normally for example by tapping the app tile. The following snippet shows how that is done and serves as a model for implementing additional optional entry points.

```
/// <summary>
///   Configures the Framework and displays initial content when the user launched the application normally.
/// </summary>
/// <param name="args"> Details about the launch request. </param>
protected override async void OnLaunched(LaunchActivatedEventArgs args)
{
   var rootFrame = Window.Current.Content as Frame;
  // Do not repeat app initialization when the Window already has content,
  // just ensure that the window is active.
  if (rootFrame == null)
   {
      await InitializeRuntimeAsync();
      rootFrame = CreateApplicationFrame();
      RootNavigator = CreateRootNavigator(rootFrame);
     if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
         await RestoreApplicationStateAsync();
     // Place the root frame in the current Window
     Window.Current.Content = rootFrame;
   }
  if (rootFrame.Content == null)
   {
     // When the navigation stack isn't restored, navigate to the first page,
     // configuring the new page by passing the arguments that were passed to the app
     // as a navigation parameter.
     await RootNavigator.NavigateToAsync(
         _rootViewModelType, target => Navigator.TryInjectParameter(target, args.Arguments, "Arguments"));
   }
  // Ensure the current window is active
  Window.Current.Activate();
}
```

Saving and restoring application state

As previously mentioned, a Windows Store app can be suspended, resumed or terminated by the Operating System at will. Depending on our application we may need to specifically code for this to make sure the application activates properly based on the previous execution state. CocktailMefWindowsStoreApplication provides three virtual methods where logic can be placed to deal with suspending, resuming and restoring application state if the application got terminated and evicted from memory since the last time it was used.

As of version 2.4, Punch has built-in support for saving and restoring application state. See Suspend and Resume.

```
sealed partial class App : CocktailMefWindowsStoreApplication
{
  protected override async void OnSuspending(SuspendingEventArgs args)
   {
     // Defer supspension until we saved the current application state.
     var deferral = args.SuspendingOperation.GetDeferral();
     // Place logic here to save application state. We won't get another chance.
     // Typically this step is asynchronous
     deferral.Complete();
   }
  protected override void OnResuming()
   {
     // Place optional logic to be performed if the application resumed from suspension
  }
  protected override Task RestoreApplicationStateAsync()
   {
     // Place logic here to restore the application state saved in OnSuspending.
  }
}
```

Alternate entry points

A Windows Store application has a series of alternate entry points besides being launched normally by the user. For example a Windows Store app can implement the search contract such that it can participate in the Search Charm. Typically each alternate entry point follows the following pattern to initialize the application and display appropriate content.

```
sealed partial class App : CocktailMefWindowsStoreApplication
{
  protected override async void OnSearchActivated(SearchActivatedEventArgs args)
   {
      var rootFrame = Window.Current.Content as Frame;
     // Do not repeat app initialization when the Window already has content,
     // just ensure that the window is active.
     if (rootFrame == null)
      {
         await InitializeRuntimeAsync();
         rootFrame = CreateApplicationFrame();
         RootNavigator = CreateRootNavigator(rootFrame);
        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
            await RestoreApplicationStateAsync();
        // Place the root frame in the current Window
        Window.Current.Content = rootFrame;
      }
     // TODO: Navigate to search page and perform search
     // e.g. await RootNavigator.NavigateToAsync<SearchPageViewModel>(target => target.Search(args.QueryText));
     // Ensure the current window is active
     Window.Current.Activate();
   }
}
```

Registering additional parts with the MEF container

MEF for Windows Store apps is significantly different from MEF in .NET and Silverlight. It is a simplified, lightweight rethinking of the full MEF available in .NET and Silverlight. One of the main differences is that MEF for Windows Store apps doesn't have the concept of a Catalog and configuring the CompositionContainer is completely different.

Punch provides two ways to register additional parts. MEF for Windows Store apps supports convention-based registration of parts.

```
sealed partial class App : CocktailMefWindowsStoreApplication
{
  protected override void PrepareConventions(ConventionBuilder conventions)
   {
     base.PrepareConventions(conventions);
     // Export type Foo by it's concrete type
     conventions.ForType<Foo>()
         .Export();
     // Export all types implementing IBar as IBar
     conventions.ForTypesDerivedFrom<IBar>()
         .Export<IBar>();
   }
}
```

The second way is to register existing instances via AddExportedValue().

```
sealed partial class App : CocktailMefWindowsStoreApplication
{
  protected override void StartRuntime()
   {
     base.StartRuntime();
     // Create an instance of Foo and register it with the CompositionContainer
     AddExportedValue<IFoo>(new Foo());
   }
}
```