Unity Application Block Hands-On Lab for Enterprise Library

This walkthrough should act as your guide to learn about the Unity Application Block and to practice how to leverage its capabilities in various application contexts.

This hands-on lab includes the following five labs:

- Lab 1: Using a Unity Container
- Lab 2: Configuring Injection Using the Configuration API
- Lab 3: Using a Configuration File to Set Up a Container
- Lab 4: Configuring Containers
- Lab 5: Integrating with ASP.NET and Child Containers

Authors

These Hands-On Labs were produced by the following individuals:

- Product/Program Management: Grigori Melnik (Microsoft Corporation)
- Development: Chris Tavares (Microsoft Corporation), Nicolas Botto (Digit Factory), Olaf Conijn (Olaf Conijn BV), Fernando Simonazzi (Clarius Consulting), Erik Renaud (nVentive Inc.)
- Testing: Rick Carr (DCB Software Testing, Inc) plus everyone above
- Documentation: Alex Homer and RoAnn Corbisier (Microsoft Corporation) and Dennis DeWitt (Linda Werner & Associates Inc)

Lab 1: Using a Unity Container

Estimated time to complete this lab: **15 minutes**

Introduction

In this lab, you will practice using a Unity container to create application objects and wire them together. You will update a simple stocks ticker application to replace calls to constructors and property setters with requests to a properly configured Unity container.

To begin this lab, open the StocksTicker.sln file located in the \Lab01\begin\StocksTicker folder.

Running the Application

Start by launching the application. To launch the application, on the Visual Studio® **Debug** menu click **Start Without Debugging** or press Ctrl-F5. This opens a form and a console window. The console window opens to display the messages logged to the console while the application runs.

On the application form, you can enter stock quote symbols, consisting of only letters, and subscribe to them by clicking the **Subscribe** button. When the **Refresh** check box is selected, the form displays the latest information about each of the subscribed stock symbols. The form will flash each time an update for a stock symbol is received. The following figure illustrates a sample stocks ticker application.
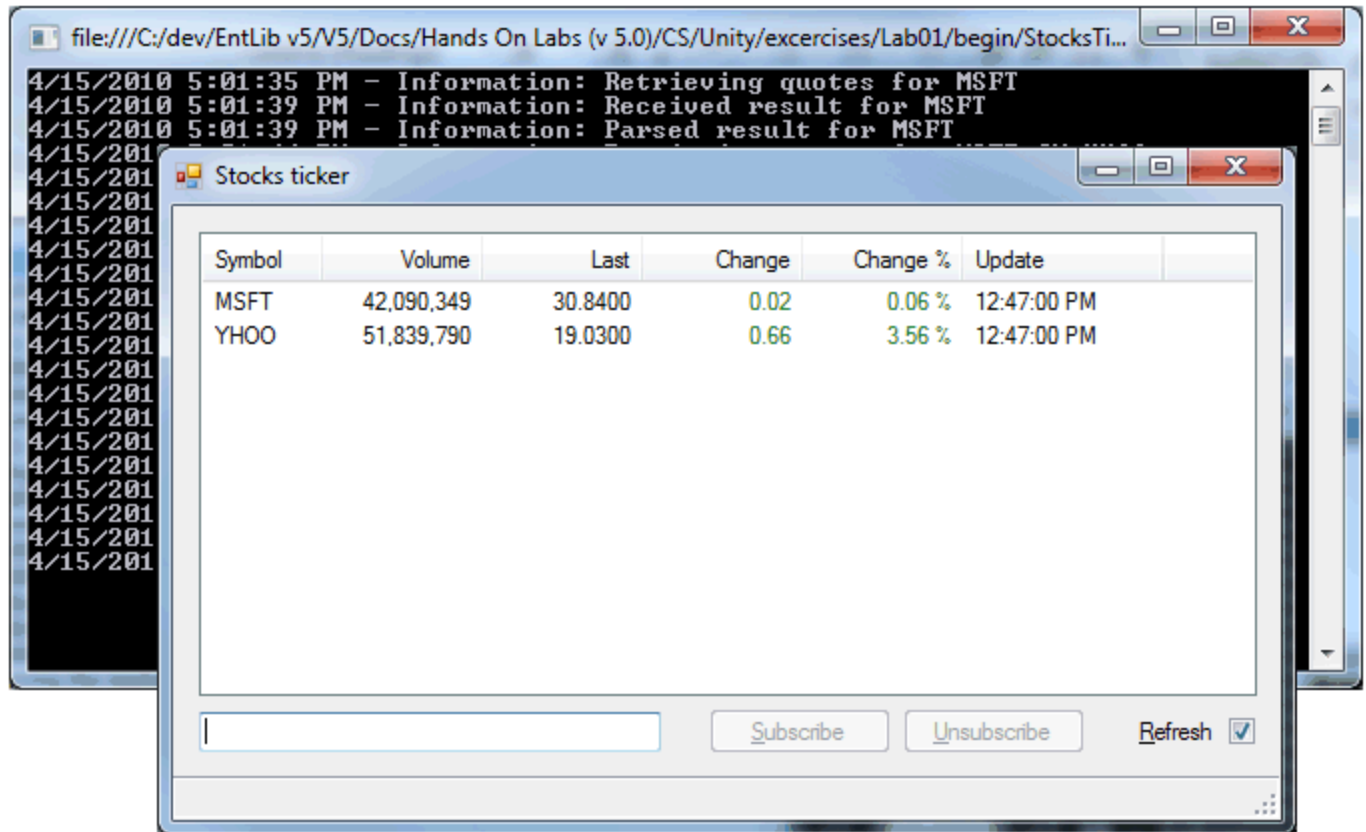
Using the application's default configuration, the actual stock information is retrieved from MoneyCentralRemote. This is a live market data feed, so there may not be updates if the markets are closed when you run the application. If you would like to do this lab without having a live Internet connection, replace the **MoneyCentralStockQuoteService** class with the **RandomStockQuoteService** whenever it is referenced in the instructions.

By default, the console is used to log information about the retrieval of updated quotes. The ui.log file (located in the StocksTicker\bin\Debug folder) contains information about the operations performed by the user interface (UI). After you close the application, you can review the contents of the log file.

Reviewing the Application

The application is built by using the Model-View-Presenter (MVP) pattern. For information about the MVP pattern, see Separated Presentation on MSDN®. The following figure**Error! Reference source not**

found. shows the classes involved in the application and their relationships.
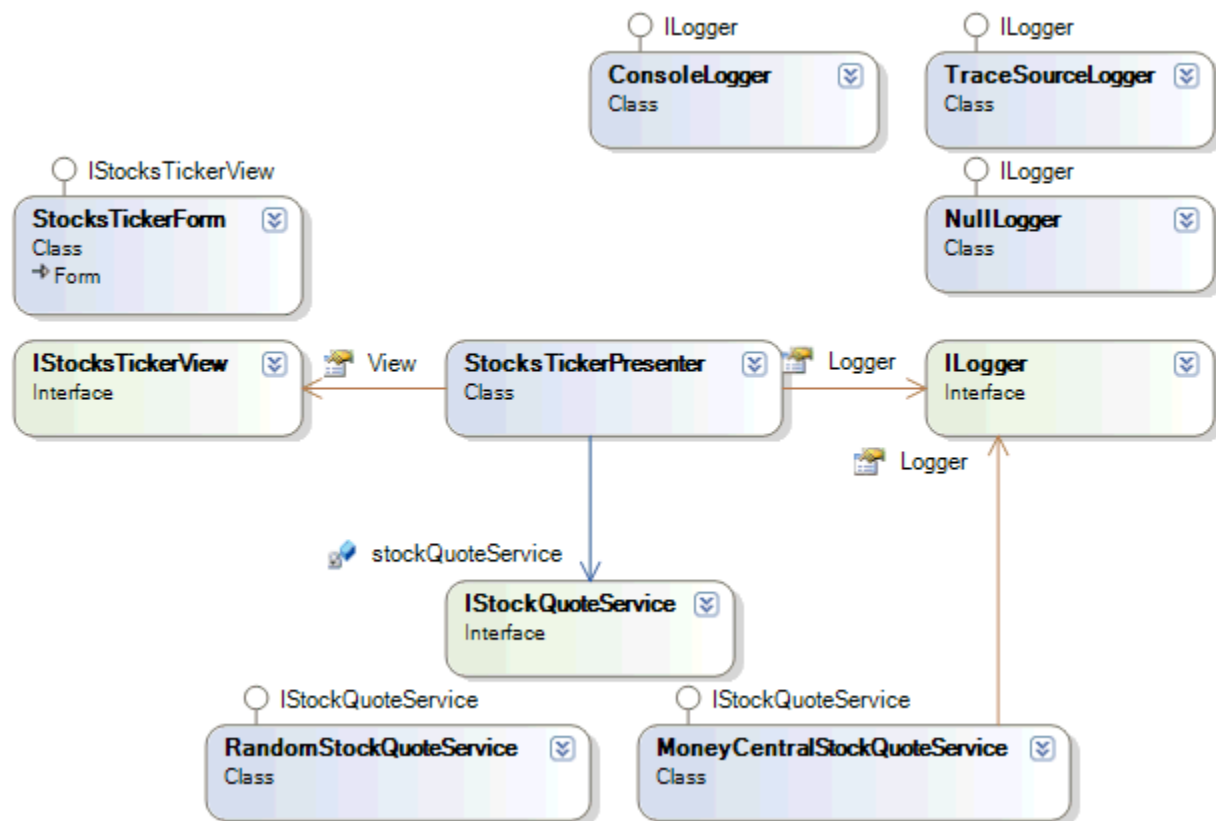
The **StocksTickerForm** class implements the user interface, and the **IStockQuoteService** interface defines a service for retrieving current stock quotes, the **MoneyCentralStockQuoteService** is the concrete implementation of **IStockQuoteService** that is used in these labs. The **ILogger** interface and its implementations are used to log messages about the operation of the application. Finally, the **StocksTickePresenter** class plays the presenter role.

All classes in the solution, including the presenter class, have constructors that take all of the required collaborators as parameters and properties where optional collaborators can be set. In the presenter's case, the required collaborators are its view and the service used to poll for updates, and logger is an optional collaborator.

The static **Program.Main()** method creates all the involved objects in order and connects them together before launching the user interface (UI). The purpose of this lab is to replace the explicit creation of these objects with the use of a Unity container.

Task 1: Using a Container

In this task, the application's startup code will be updated to use a Unity container to create and connect the application's objects. This will replace the use of explicit calls to the classes' constructors and property setters.

Adding References to the Required Assemblies

In Solution Explorer, select the **StocksTicker** project, and then click **Add Reference** on the **Project** menu. This opens the **Add Reference** dialog box. Select the **Microsoft.Practices.Unity** assembly, and then click **OK**.
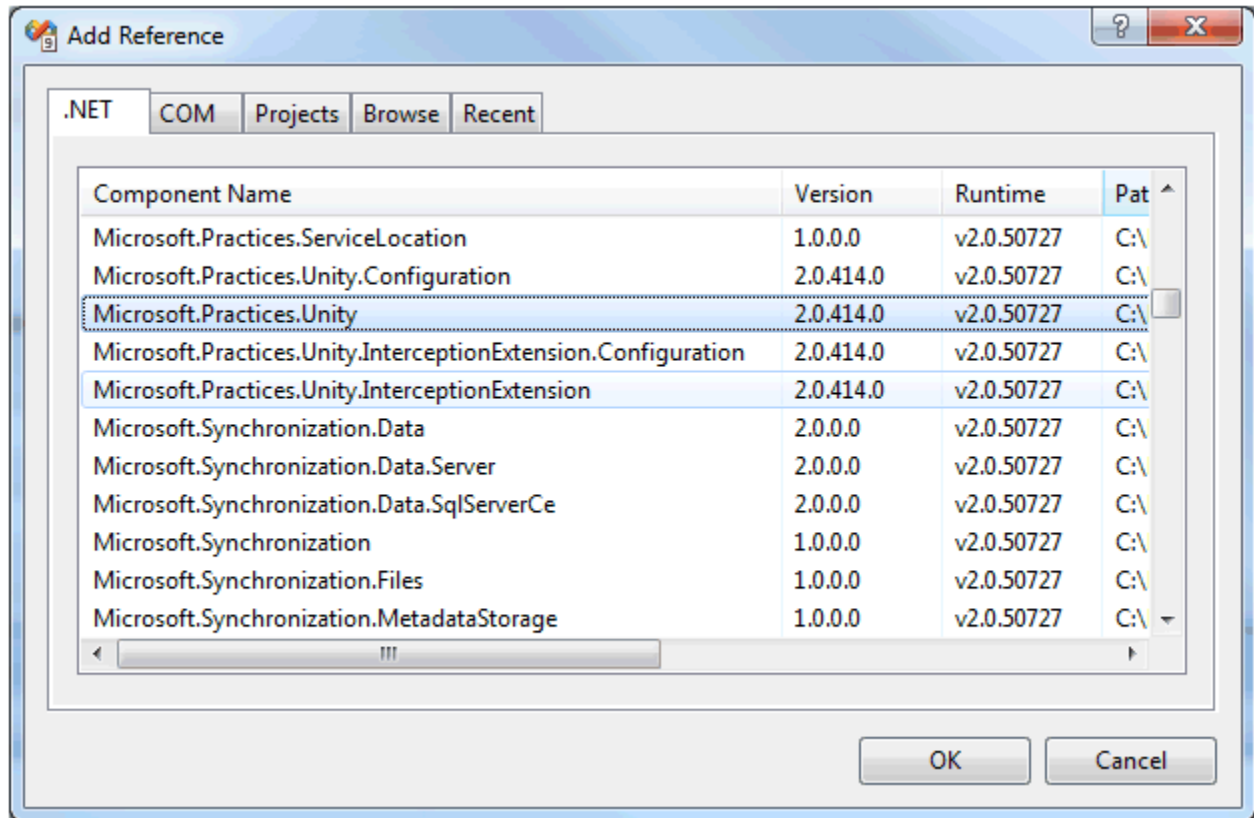
Update the Startup Code to Use a Container

To update the startup code to use a container

1. Open the Program.cs file.

2. Add a **using** directive for the **Unity** namespace.

   ```
   using Microsoft.Practices.Unity;
   ```

3. Replace the creation of the instances (view, presenter, service, loggers) with a **Resolve** request to a new **UnityContainer** for the **StocksTickerPresenter**, as shown in the following highlighted code.

   ```
   static void Main()
   {
     Application.EnableVisualStyles();
     Application.SetCompatibleTextRenderingDefault(false);

     using (IUnityContainer container = new UnityContainer())
   ```

```
  {
    StocksTickerPresenter presenter
          = container.Resolve<StocksTickerPresenter>();

    Application.Run((Form)presenter.View);
  }
}
```

Unity containers implement the **IDisposable** interface. This enables the new code to take advantage of the **using** statement to dispose the new container. Lab 2 elaborates on the result of disposing a container.

> These changes are enough to successfully build the application. However, even though the Unity container can build objects using the default rules ("autowiring"), additional set up is required before the container can successfully resolve this application's objects.

4. Debug the application. To do this, click **Start Debugging** on the Visual Studio **Debug** menu or Press F5. The debugger will break with an unhandled **ResolutionFailedException**. This indicates that the attempt to resolve the **StocksTickerPresenter** has failed.

Although the container requires additional configuration to resolve the requested presenter, the messages for the **ResolutionFailedException** and its chain of **InnerExceptions** reveal some interesting details:

   ◦ The request to resolve the presenter is identified with the build key **type = "StocksTicker.UI.StocksTickerPresenter", name = "(none)"**. In this case no name has been provided.

   ◦ The error occurred while resolving an object.

   ◦ The container determined that the constructor for **StocksTickerPresenter** with the signature **(IStocksTickerView view, IStockQuoteService stockQuoteService)** should be used to create the object, by using the autowiring rules.

   ◦ The root cause of the failure was the inability to build an instance of the **IStocksTickerView** interface to be the value for the **view** parameter in the chosen **StocksTickerPresenter** constructor. This problem is indicated by an **InvalidOperationException** with the following message: "The current type, StocksTicker.UI.IStocksTickerView, is an interface and cannot be constructed. Are you missing a type mapping?"

For information about Unity's autowiring rules, see the topic "Annotating Objects for Constructor Injection" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660875(v=PandP.20).aspx.

5. Add the required type mappings by using the **RegisterType** method, as shown in the following highlighted code .This will enable the container to resolve the required objects.

```
using (IUnityContainer container = new UnityContainer())
{
  container
      .RegisterType<IStocksTickerView, StocksTickerForm>()
      .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>();

  StocksTickerPresenter presenter
            = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

Using the **RegisterType** method is primarily how you will set up a container. In addition to mapping abstract types to concrete ones as described in this step, you can use the **RegisterType** method to override the default injection rules and specify lifetime managers. The next exercises will show additional uses of the **RegisterType** method. For details about mapping types see the topic "Registering Types and Type Mappings" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660923(v=PandP.20).aspx.

> There are no mappings for the **ILogger** interface. They are not required at this point because properties are not injected unless they are explicitly configured for injection. The application will run now because the involved objects' required collaborators can be resolved by the container, Logging will still not be available because loggers will not be injected into the resolved objects because loggers are optional collaborators in this application.

Running the Application

Launch the application and use it. The application behaves as it did before the changes, except that no logging will be available in the console or the ui.log file.

Task 2: Using Attributes to Control Injection

Attributes can be used to override the default injection rules. In some cases, attributes are used to opt-in for injection on members ignored by the default rules. In other cases, attributes must be used to override the default rules or disambiguate cases where the rules cannot be used.

Using Attributes to Enable Property Injection

It is very common to have the container set properties, in addition to passing in dependencies by using constructor arguments.

To set up injection for the Logger property on the MoneyCentralStockQuoteService class

1. Open the StockQuoteServices\MoneyCentralStockQuoteService.cs file.

2. Add a **using** directive for the Unity namespace.

```
using Microsoft.Practices.Unity;
```

3. Add the **Dependency** attribute to the **Logger** property, as shown in the following highlighted code.

```
private ILogger logger;
[Dependency]
public ILogger Logger
{
    get { return logger; }
    set { logger = value; }
}
```

> The **Dependency** attribute can be used to indicate that the property should be injected, which results in resolving the property's type, **ILogger**, in the container. For information about property (setter) injection, see the topic "Annotating Objects for Property (Setter) Injection" in the Unity 2.0 documentation available at http://msdn.microsoft.com/en-us/library/ff660903(v=PandP.20).aspx.

Adding a Type Registration to Resolve the ILogger Interface

Because the container must now resolve the **ILogger** interface, a mapping from the interface to a concrete type must be added to the container. In this example, the interface will be mapped to the **ConsoleLogger** class in order to match the code originally used to wire-up the objects.

To add a type registration to resolve the ILogger interface

1. Open the Program.cs file.

2. Use the **RegisterType** method to map the **ILogger** interface to the **ConsoleLogger** class, as shown in the following highlighted code.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>()
    .RegisterType<ILogger, ConsoleLogger>();

  StocksTickerPresenter presenter
            = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> Notice how the new call to the **RegisterType** method is chained to the previous calls.

Running the Application

Launch and use the application. Operations performed by the service are logged to the console, but UI operations are not logged to the console because the **Logger** property on the presenter class is still not injected.

Setting Up Injection for the Logger Property on the StocksTickerPresenter Class

To inject the **Logger** property on the presenter class, this lab uses the same **Dependency** attribute. However, because a different logger instance is to be injected, you need a way to differentiate the loggers. The Unity container lets you register the same type multiple times, and give each registration a different name. When that dependency is resolved, the name can be used to specify exactly which instance you want. In this lab, you will use different names so that you can tell the difference between the logger objects.

To set up injection for the Logger property on the StocksTickerPresenter class

1. Open the UI\ StocksTickerPresenter.cs file.

2. Add a using directive for the Unity namespace.

   ```
   using Microsoft.Practices.Unity;
   ```

3. Add the **Dependency** attribute to the **Logger** property using "**UI"** for the name, as shown in the following highlighted code.

   ```
   private ILogger logger;

   [Dependency("UI")]
   public ILogger Logger
   {
     get { return logger; }
     set { logger = value; }
   }
   ```

Adding a Type Registration to Resolve the ILogger Interface with the "UI" Name

To add a type registration to resolve the ILogger interface with the "UI" name

1. Open the Program.cs file.

2. Use the **RegisterType** method to map the **ILogger** interface to the **TraceSourceLogger** class with the "**UI"** name, as shown in the following highlighted code.

   ```
   using (IUnityContainer container = new UnityContainer())
   {
     container
       .RegisterType<IStocksTickerView, StocksTickerForm>()
   ```

```
        .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>()
        .RegisterType<ILogger, ConsoleLogger>()
        .RegisterType<ILogger, TraceSourceLogger>("UI");

  StocksTickerPresenter presenter
              = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> The name parameter for the **RegisterType** method is optional. The default is **null**.

Debugging the Application

Launch the application. The debugger should break with an unhandled exception that indicates a new failure while trying to resolve the **Logger** property on the presenter to an instance of **TraceSourceLogger**. Examining the chain of **InnerExceptions** shows that the root cause of the failure is indicated with an **InvalidOperationException** and the message "The type TraceSourceLogger has multiple constructors of length 1. Unable to disambiguate." In this case, Unity's default injection rules cannot determine how to build the instance, so the container must be explicitly configured to build it.

Indicating Which Constructor to Use When Building an Instance of TraceSourceLogger with the InjectionConstructor Attribute

This lab uses an attribute to indicate which of the two available constructors should be used to create an instance of the **TraceSourceLogger** class.

To indicate which constructor to use when building an instance of TraceSourceLogger with the InjectionConstructor attribute

1. Open the Loggers\TraceSourceLogger.cs file.

2. Add a **using** directive for the Unity namespace.

    ```
    using Microsoft.Practices.Unity;
    ```

3. Add the **InjectionConstructor** attribute to the constructor that takes a **TraceSource** as its sole parameter, as shown in the following highlighted code.

    ```
    [InjectionConstructor]
    public TraceSourceLogger(TraceSource traceSource)
    {
      this.traceSource = traceSource;
    }
    ```

Adding an Instance Registration to Resolve the TraceSource Type

After being pointed to one of the constructors, Unity's default injection rules take effect and an instance of the .NET Framework **TraceSource** class will be resolved to be used as the argument for the constructor call. Instead of instructing the container about how to build such an instance, which would be problematic since the class cannot be annotated with attributes, a pre-built instance will be supplied to the container.

To add an instance registration to resolve the TraceSource type

1. Open the Program.cs file.

2. Use the **RegisterInstance** method to indicate the instance to return when resolving the **TraceSource** class, as shown in the following highlighted code.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>()
    .RegisterType<ILogger, ConsoleLogger>()
    .RegisterType<ILogger, TraceSourceLogger>("UI")
    .RegisterInstance(new TraceSource("UI", SourceLevels.All));

  StocksTickerPresenter presenter
          = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> The container has generic and non-generic versions of the **RegisterInstance** method. In this task, the generic version is used, relying on the compiler's inference to avoid specifying the generic type argument. This works in this task because the type of the expression used to obtain the reference to register has the same type (**TraceSource**) that will be resolved by the container. If there was a need for a mapping—for example, if an instance is registered to supply the implementation for an interface— the generic type argument should have been provided.

Running the Application

Launch and use the application. Now the application should work exactly as it did initially, with messages from the UI being logged to the ui.log file (located in the StocksTicker\bin\Debug folder) and messages from the service being logged to the console.

Attributes are a convenient mechanism to override or disambiguate the container's default injection rules but can result in brittle dependencies. Using names can make particularly brittle dependencies because they get hard-coded in the source code. The next labs show alternative mechanisms to externalize the setup of a container without involving the objects being created.

To verify you have completed the lab correctly, you can use the solution provided in the \Lab01\end\StocksTicker folder.

Lab 2: Configuring Injection Using the Configuration API

Estimated time to complete this lab: **20 minutes**

Introduction

In this lab, you will practice configuring a container to perform dependency injection at run time, without relying on annotating the classes to resolve with attributes and set up lifetime managers.

To begin, open the StocksTicker.sln file located in the Lab02\begin\StocksTicker folder. This is the same application used in Lab 1.

Task 1: Configuring the Container Using the Fluent API

Updating the Container Configuration to Override the Default Injection Rules

Throughout this task, calls to the **RegisterType** method will be added or modified in order to configure the container.

To provide injection configuration, calls to the container's **RegisterType** methods receive **InjectionMember** objects. **InjectionMember** is a base class. Its subclasses **InjectionProperty** and **InjectionConstructor** are used in this lab. For information about configuring injection in a Unity container, see the topic "Registering Injected Parameter and Property Values" in the Unity 2.0 documentation available at http://msdn.microsoft.com/en-us/library/ff660882(v=PandP.20).aspx.

To update container configuration to override the default injection rules

1. Open the Program.cs file.

2. Update the **RegisterType** call for the **IStockQuoteService** interface in order to inject the **Logger** property using the **InjectionProperty** object, as shown in the following highlighted code.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>(
                new InjectionProperty("Logger"))
    .RegisterType<ILogger, ConsoleLogger>()
    .RegisterType<ILogger, TraceSourceLogger>("UI")
    .RegisterInstance(new TraceSource("UI", SourceLevels.All));

  StocksTickerPresenter presenter
```

```
                    = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> The **InjectionProperty** object as configured in the preceding code indicates that the property
> with the name "**Logger**" should be injected. Because there is no further configuration for this
> property, the unnamed instance for the property's type, **ILogger**, will be resolved when
> obtaining the value to inject to the property. This is similar to the previous lab which used the
> **Dependency** attribute without further configuration to annotate the property.

3. Use the **RegisterType** method to set up the injection of the **StocksTickerPresenter** class, as
    shown in the following highlighted code.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>(
                  new InjectionProperty("Logger"))
    .RegisterType<ILogger, ConsoleLogger>()
    .RegisterType<ILogger, TraceSourceLogger>("UI")
    .RegisterInstance(new TraceSource("UI", SourceLevels.All))
    .RegisterType<StocksTickerPresenter>(
            new InjectionProperty("Logger",
                new ResolvedParameter<ILogger>("UI")));

  StocksTickerPresenter presenter
            = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> In this case, the **RegisterType** call is not used to perform a mapping. It is used only to inject the
> **Logger** property. Because of this, there is a single generic type argument. Additionally, the
> **InjectionProperty** is configured with a **ResolvedParameter**. This parameter is used to indicate
> the name of the instance to resolve (which was not necessary in the previous step, so it was
> omitted.)

This configuration is sufficient to successfully run the application. The **InjectionConstructor**
attribute drives the creation of the **TraceSourceLogger** object, injecting it with the registered
TraceSource instance. However, there are many cases where you cannot apply an attribute or
where you want to override the attribute and inject something different. The next step shows
how to use the **RegisterType** method to override the default creation rules and use a different
constructor to create the **TraceSourceLogger**.

4. Update the **RegisterType** call for the **ILogger** interface with the "**UI**" name in order to inject the constructor with a single string parameter, as shown in the following highlighted code.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>(
                   new InjectionProperty("Logger"))
    .RegisterType<ILogger, ConsoleLogger>()
    .RegisterType<ILogger, TraceSourceLogger>(
                           "UI",
                           new InjectionConstructor("UI"))
    .RegisterInstance(new TraceSource("UI", SourceLevels.All))
    .RegisterType<StocksTickerPresenter>(
                   new InjectionProperty("Logger",
                       new ResolvedParameter<ILogger>("UI")));

  StocksTickerPresenter presenter
              = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

The **InjectionConstructor** indicates which constructor to use based on its arguments. In this case, the supplied "**UI**" string causes the constructor of **TraceSourceLogger** with a single string parameter to be called, passing the "**UI**" value as its argument and taking precedence over the constructor annotated with the **InjectionConstructor** attribute. There are three kinds of arguments for an **InjectionConstructor** (and all other members of the **InjectionMember** hierarchy): instances of concrete subclasses of the **InjectionParameterValue** class, instances of the **Type** class, and the remaining objects. For information about how these values are interpreted, see the topic "Registering Injected Parameter and Property Values" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660882(v=PandP.20).aspx.

5. Remove the **RegisterInstance** call, which is no longer necessary.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>(
                   new InjectionProperty("Logger"))
    .RegisterType<ILogger, ConsoleLogger>()
    .RegisterType<ILogger, TraceSourceLogger>(
                           "UI",
                           new InjectionConstructor("UI"))
    .RegisterInstance(new TraceSource("UI", SourceLevels.All))
    .RegisterType<StocksTickerPresenter>(
```

```
                    new InjectionProperty("Logger",
                        new ResolvedParameter<ILogger>("UI")));

  StocksTickerPresenter presenter
            = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> The container needs information on how to resolve an instance of the **TraceSource** class in order to successfully resolve the configured **TraceSourceLogger** using the annotated constructor. Because this constructor is no longer used, the registered instance is no longer necessary. Also note that, using the mechanisms described in this exercise, the container could be provided the necessary information to build a **TraceSource** instance instead of being limited to registering an instance created elsewhere.

Running the Application

Launch and use the application. Its behavior is the same as it was before the changes, but now the information required by the container to build the application types does not reside in the types themselves (with the exception of the **InjectionConstructor** attribute in the **TraceSourceLogger** type, which was left in place to illustrate how the API calls took precedence over the attributes).

Using the **RegisterType** method to configure injection provides a degree of flexibility that is not available when using attributes. However, it also requires injection to be specified for each container.

Task 2: Using Lifetime Managers

Lifetime managers are used by a container for two purposes:

- To make sure the result of resolving a particular identifier is always the same instance for a particular context (for example, the same container, the same thread, or the same HTTP session)

- To properly dispose the resolved objects

In this task, the built-in **ContainerControlledLifetimeManager** will be used to dispose the **TraceSourceLogger**.

Updating the TraceSourceLogger Class to Implement the IDisposable Interface

The **TraceSourceLogger**, as implemented at this stage, flushes its **TraceSource** after each message is logged and does not close it. In this task, the class is changed to implement the **IDisposable** interface in order to properly close its trace source.

To update the TraceSourceLogger class to implement the IDisposable interface

1. Open the Loggers\TraceSourceLogger.cs file.

2. Add the **IDisposable** interface to the list of interfaces implemented by the **TraceSourceLogger** class, as shown in following highlighted code.

```
public class TraceSourceLogger : ILogger, IDisposable
{
   ...
}
```

3. Remove the call to the **Flush** method in the implementation of the **Log** method, as shown in the following highlighted code.

```
public void Log(string message, TraceEventType eventType)
{
   this.traceSource.TraceEvent(eventType, 0, message);
   this.traceSource.Flush();
}
```

This change enables buffering of the logged messages by not forcing the trace source to flush each time a message is logged.

4. Add the following code to implement the **Dispose** method defined in the **IDisposable** interface.

```
public void Dispose()
{
   if (this.traceSource != null)
   {
      this.traceSource.TraceInformation("Shutting down logger");
      this.traceSource.Close();
      this.traceSource = null;
   }
}
```

This implementation logs a "shutting down" message and closes the trace source.

Updating the Container Configuration Calls to Use a Lifetime Manager for the TraceSourceLogger Class

To update the container configuration calls to use a lifetime manager for the TraceSourceLogger class

- Update the **RegisterType** call for the **ILogger** interface with the "**UI**" name to use a new **ContainerControlledLifetimeManager** instance, as shown in the following highlighted code.

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>(
                  new InjectionProperty("Logger"))
    .RegisterType<ILogger, ConsoleLogger>()
```

```
        .RegisterType<ILogger, TraceSourceLogger>(
                "UI",
                new ContainerControlledLifetimeManager(),
                new InjectionConstructor("UI"))
     .RegisterType<StocksTickerPresenter>(
                new InjectionProperty("Logger",
                    new ResolvedParameter<ILogger>("UI")));

  StocksTickerPresenter presenter
         = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> A lifetime manager is an optional parameter of the **RegisterType** type. For information about lifetime managers in general, and the **ContainerControlledLifetimeManager** in particular, see the topic "Understanding Lifetime Managers" in the Unity 2.0 documentation available at http://msdn.microsoft.com/en-us/library/ff660872(v=PandP.20).aspx.

Running the Application

Launch the application, use it, and then close its main form to ensure the shutdown code is executed. Open the ui.log file (located in the StocksTicker\bin\Debug folder) and ensure the last two lines show an entry like the following.

```
UI Information: 0 : Shutting down logger
    DateTime=2009-02-11T19:02:07.8990000Z
```

To verify you have completed the lab correctly, you can use the solution provided in the Lab02\end\StocksTicker folder.

Lab 3: Using a Configuration File to Set Up a Container

Estimated time to complete this lab: **25 minutes**

Introduction

In this lab, you will practice using settings retrieved from an application's configuration file to set up a Unity container. Setting up a container with settings from a configuration file is similar to setting up a container by invoking the configuration API used in the previous lab. In fact, configuration settings can be thought of as *script* representing calls to the API.

To begin, open the StocksTicker.sln file located in the Lab03\begin\StocksTicker folder.

Task 1: Using a Configuration File to Store the Configuration for a Container

In this exercise, the container set-up code will be replaced with its equivalent representation as configuration file settings.

Adding References to the Required Assemblies

Add a reference to the **Microsoft.Practices.Unity.Configuration** and **System.Configuration** assemblies. The .NET Framework classes from the **System.Configuration** assembly are used to retrieve information from the application's configuration file.

Updating the Startup Code to Use Information from the Configuration File to Set Up the Container

To update the startup code to use information from the configuration file to set up the container

1. Open the Program.cs file.

2. Add a **using** directive for the **Unity.Configuration** namespace.

   ```
   using Microsoft.Practices.Unity.Configuration;
   ```

3. Retrieve the Unity configuration section from the configuration file and use it to configure the container. Replace the calls to **RegisterType** with a call to the **LoadConfiguration** method, as shown in the following highlighted code.

   ```
   using (IUnityContainer container = new UnityContainer())
   {
   ```

```
container.LoadConfiguration();

StocksTickerPresenter presenter
        = container.Resolve<StocksTickerPresenter>();

Application.Run((Form)presenter.View);
}
```

The **LoadConfiguration** extension method is the simplest way to configure a container. It retrieves the configuration information for the default container element from the unity configuration section in the executing application's configuration file. There are alternative mechanisms for more sophisticated scenarios such as retrieving the configuration from configuration files other than the executing application's configuration file.

For information about how to load configuration information into a container, see the topic "Loading Configuration File Information into a Container" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660935(v=PandP.20).aspx#load_config.

Updating the Configuration File with the Container Configuration

Only a subset of Unity's configuration elements is used during this task. For an overall description of Unity's configuration schema, see the topic "The Unity Configuration Schema" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660914(v=PandP.20).aspx.

To update the configuration file with the container configuration

1. Open the application's configuration, App.config, file.

2. Add a declaration for the **unity** configuration section.

    ```xml
    <configSections>
      ...
      <section name="unity"
              type="
    Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
    Microsoft.Practices.Unity.Configuration, Version=2.0.414.0, Culture=neutral,
    PublicKeyToken=31bf3856ad364e35"/>
    </configSections>
    ```

    The name "**unity**" for the configuration section is only a convention; any name is valid, as long as it matches the name used to retrieve the configuration at run time. Most simplified configuration access methods assume the "unity" section name.

3. Add the **unity** section element.

    ```xml
      </system.diagnostics>
      <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
      </unity>
    </configuration>
    ```

The section's element name must match the name used to register the section in the **configSections** collection.

> The **xmlns** attribute is not required by the configuration runtime. If you do set it and use **"unity"** as the section name it enables the Visual Studio XML editor, which is used by default to edit configuration files, to take advantage of the Unity Configuration XSD file installed with Unity to provide support for IntelliSense. This greatly simplifies the authoring of configuration information involving the Unity container.

4. Add an **alias** element for the **TraceSource** type.

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <alias
    alias="TraceSource"
    type="System.Diagnostics.TraceSource, System, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" />
</unity>
```

> Type aliases are not required. Type aliases and the assembly and namespace elements typically make the configuration files less verbose and easier to read. Aliases for common types, such as .NET framework types like **int** and **string** or Unity types like **singleton**, are built-in so you do not need to define them.

5. Add an **assembly** element for the application's assembly and **namespace** elements for the namespaces containing types that are used to configure the container.

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <alias
    alias="TraceSource"
    type="System.Diagnostics.TraceSource, System, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" />

  <assembly name="StocksTicker"/>
  <namespace name="StocksTicker.Loggers"/>
  <namespace name="StocksTicker.StockQuoteServices"/>
  <namespace name="StocksTicker.UI"/>
</unity>
```

> While the **alias** element can be used to assign an arbitrary name to a single type, the **assembly** and **namespace** elements indicate the configuration run-time in which assemblies and namespaces search for types for which only the name is specified in the configuration file.

6. Add an element for the default (unnamed) **container**.

```
  <namespace name="StocksTicker.UI"/>

  <container>
  </container>
</unity>
```

> Use of the default container only differs from named containers, in that it is used if no container name is supplied when loading the configuration.

7. Add a **register** element to map the **IStocksTickerView** interface to the **StocksTickerForm** class.

```
<container>
  <register type="IStocksTickerView" mapTo="StocksTickerForm"/>
</container>
```

> Notice only the type names are used with no indication of namespace or assembly. The full type names could have also been used but the configuration run-time will be able to retrieve the right types by using the information in the **namespace** and **assembly** elements.

8. Add a **register** element to map the **IStockQuoteService** interface to the **MoneyCentralStockQuoteService** class, with a child **property** element to configure the **Logger** property to be injected.

```
<container>
  <register type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <property name="Logger"/>
  </register>
</container>
```

9. Add a **register** element to map the **ILogger** interface to the **ConsoleLogger** class.

```
<container>
  <register type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <property name="Logger"/>
  </register>
  <register type="ILogger" mapTo="ConsoleLogger"/>
</container>
```

10. Add a **register** element to map the **ILogger** interface to the **TraceSourceLogger** using the "**UI**" name. Add a child **constructor** element to indicate the use of the constructor with a single parameter named **traceSourceName** to build the instance, and pass the "UI" string as the argument with the **value** attribute. Use a child **lifetime** element to indicate the use of the **ContainerControlledLifetimeManager** through the use of the built-in **singleton** alias.

```
<container>
  <register type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <property name="Logger"/>
  </register>
  <register type="ILogger" mapTo="ConsoleLogger"/>
  <register name="UI" type="ILogger" mapTo="TraceSourceLogger">
    <lifetime type="singleton"/>
    <constructor>
```

```
        <param name="traceSourceName" value="UI"/>
      </constructor>
   </register>
</container>
```

> The **constructor** element is equivalent to the **InjectionConstructor** class used in the [previous lab](). The **value** attribute, used as a shortcut for the **value** child element, indicates that its value should be supplied as the value for the parameter it represents. If necessary, it is converted to the parameter's type. For information about the **value** element, see the topic "The value Element" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660914(v=PandP.20).aspx#config_value.

11. Add a **register** element for the **StocksTickerPresenter** class with a child **property** element. The **property** element configures the **Logger** property to be injected with the value of resolving the **ILogger** interface (the property's type), with the "**UI**" name (which requires using the **dependency** element, as shown below. Alternatively you could use the **dependencyName** attribute on the **property** element.

```
<container>
  <register type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <property name="Logger"/>
  </register>
  <register type="ILogger" mapTo="ConsoleLogger"/>
  <register name="UI" type="ILogger" mapTo="TraceSourceLogger">
    <lifetime type="singleton"/>
    <constructor>
      <param name="traceSourceName" value="UI"/>
    </constructor>
  </register>
  <register type="StocksTickerPresenter">
    <property name="Logger">
      <dependency name="UI"/>
    </property>
  </register>
</container>
```

> Notice how "**UI**" is used to indicate the name of the instance to resolve, while it is used as a literal string value in the previous step. Also note the absence of the **mapTo** attribute in the **register** element. The **dependency** element indicates that the value of the **name** attribute ("UI" in this case) is to be used, together with the property's type, as the key to resolve the object to inject into the property. This element is an alternative to the **value** attribute used in the previous step; for details on the **dependency** element see the topic "The dependency Element" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660914(v=PandP.20).aspx#config_dependency.

Running the Application

Launch and use the application. It behaves as it did before the changes, including the logging behavior.

Now all the information required to set up the container is stored in the configuration file; no code changes are required to change the way the container is to resolve the application objects.

To verify you have completed the lab correctly, you can use the solution provided in the Lab03\end\StocksTicker folder.

Lab 4: Configuring Containers

Estimated time to complete this lab: **15 minutes**

Introduction

In this lab, you will practice using some of Unity's more advanced features: generic decorator chains, resolver overrides and array injection.

The application used in this lab is an updated version of the stocks ticker application used in the previous labs, with the additional requirement to save updated stock quotes to a persistent repository using a third-party *persistence framework*. This persistence framework defines a generic **IRepository<>** interface, and a concrete generic **DebugRepository<>** class will be used in this lab.

To begin, open the StocksTicker.sln file located in the Lab04\begin\StocksTicker folder.

Task 1: Configuring Open Generics to Resolve Closed Generics

A Unity container can be configured using closed generic types, which work exactly like non-generic types, or using open generic types. In the latter case, configuration applies to any closed generic type built from the open one as long as there is not a more specific configuration for the closed generic type.

Reviewing Code

To review the code

1. Open the UI\StocksTickerPresenter.cs file.

2. Note the changes in the code.

The **StocksTickerPresenter** is injected with a suitable repository through a new constructor parameter.

```
public StocksTickerPresenter(
  IStocksTickerView view,
  IStockQuoteService stockQuoteService,
  IRepository<StockQuote> repository)
{
    ...
}
```

The repository is used to store updated stock quotes.

```
private void SaveQuote(StockQuote updatedQuote)
{
  try
  {
```

```
    this.repository.Save(updatedQuote);
  }
  catch (RepositoryException e)
  {
     this.logger.Log(
          string.Format(
                "Error saving the updated quote for '{0}': {1}",
                updatedQuote.Symbol,
                e.Message),
          TraceEventType.Warning);
  }
}
```

Reviewing the Configuration File

To review the configuration file

1. Open the App.config file.

2. Note the changes in the configuration file.

Just as in the previous lab, there is no explicit constructor injection configuration for the
StocksTickerPresenter: the default injection rules kick in to find the presenter's single constructor and
invoke it with the result of resolving each of its parameters. To support a new parameter of type
IRepository<StockQuote>, a new **register** element maps the closed generic interface to an
implementation, which happens to be a closed generic class.

```
<register type="IRepository[StockQuote]" mapTo="DebugRepository[StockQuote]"/>
```

For information about how to specify type names involving generics, see the topic "Specifying Types in
the Configuration File" in the Unity 2.0 documentation at [http://msdn.microsoft.com/en-](http://msdn.microsoft.com/en-us/library/ff660933(v=PandP.20).aspx)
[us/library/ff660933(v=PandP.20).aspx](http://msdn.microsoft.com/en-us/library/ff660933(v=PandP.20).aspx).

Running the Application

Launch and use the application; each time an update is received for a stock quote, the corresponding
entry is logged to the console. Note that loggers are not used in this case; the **DebugRepository<>** class
writes output directly through the **Console** class.

Updating the Configuration File to Register Open Generic Types

To update the configuration file to register open generic types

1. Open the App.config file.

2. Replace the **register** element mapping the closed **IRepository** interface to the closed
    DebugRepository class with a new element mapping the open types.

    ```
    <container>
      <register type="IStocksTickerView" mapTo="StocksTickerForm"/>
    ```

```
    <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
      ...
    </register>
    <register type="IRepository[]" mapTo="DebugRepository[]"/>
    <register type="ILogger" mapTo="ConsoleLogger"/>
    <register name="UI" type="ILogger" mapTo="TraceSourceLogger">
      ...
    </register>
    <register type="StocksTickerPresenter">
      ...
    </register>
</container>
```

Running the Application

Launch and use the application; it will run as it did before the changes.
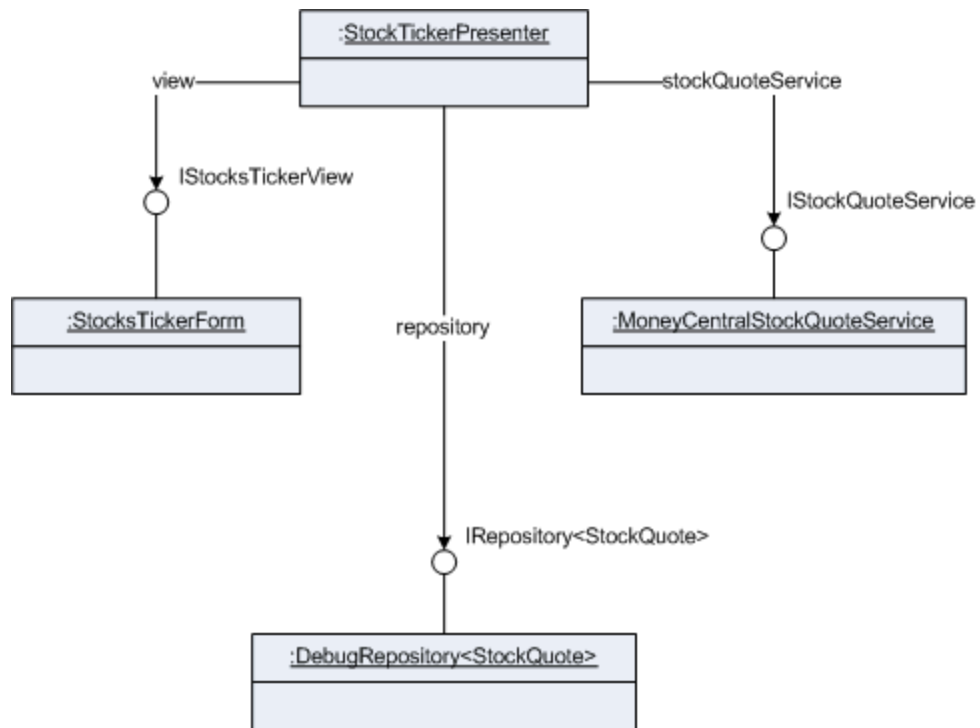
When resolving a (closed) generic type, the Unity container looks for definitions (such as mappings and injection specifications) for the closed generic type. If no such definitions are found, the container looks for definitions for the open generic type from which the resolved type is created and uses them if they are available, resulting in mapping the generic type arguments from the original closed generic type as appropriate.

In this exercise, the closed **IRepository<StockQuote>** generic interface needs to be resolved when gathering the arguments for the constructor of the **StocksTickerPresenter** class (following the container's default injection rules). After the configuration change described in this task, no entry is found for the closed generic interface, so an entry for the open **IRepository<>** interface is looked for and found. The container uses this mapping and determines that the open generic **DebugRepository<>** class should be used. The closed type's generic type argument **StockQuote** is applied to the mapped open generic class. This results in resolving the closed **DebugRepository<StockQuote>** class, which is instantiated and used as the constructor argument for the **StocksTickerPresenter**.

Configuring open generic types helps avoid duplication. Because definitions for closed generic types take precedence over definitions for open generic types, these definitions can be overridden for a specific closed generic type, if necessary.

Task 2: Resolving Generic Decorator Chains

At the end of Task 1, the runtime structure of the **StocksTickerPresenter** looks like the following schematic.

```
                    :StockTickerPresenter
view────────────[                              ]────stockQuoteService
                                │
        │ IStocksTickerView                    │ IStockQuoteService
        ▼                                      ▼
        ○                                      ○

:StocksTickerForm        repository    :MoneyCentralStockQuoteService



                                │ IRepository<StockQuote>
                                ▼
                                ○

              :DebugRepository<StockQuote>
```

In this task, the container will be configured to insert a decorator class between the presenter and the repository. The final runtime objects will look like the following schematic.
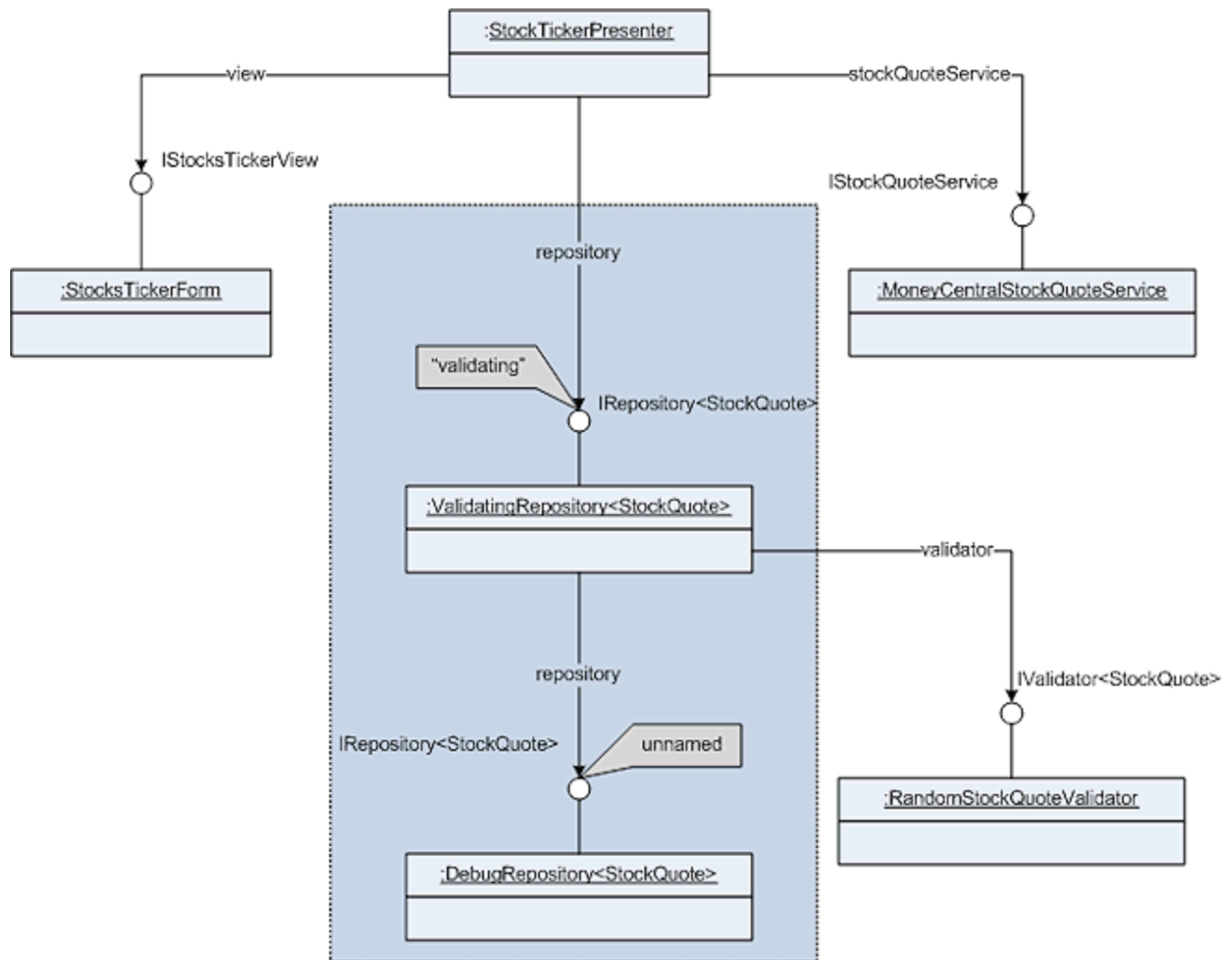
In order to accomplish this, the named registration feature of the container will be used. The names used in each registration are shown in the callout tags in the schematic.

In this task, the settings from the configuration file will be partially overridden in the application code registering new types using the **RegisterType** method and using **Resolve** overrides. As a result, the **StocksTickerPresenter** will be injected with a **ValidatingRepository<StockQuote>** wrapping the original debug repository instead of getting an instance of **DebugRepository<StockQuote>**. Mixing configuration from different sources is permitted because API calls and settings from the configuration file are equivalent. This **ValidatingRepository<>** class works as a decorator; the container is responsible for figuring out how to properly wire it up.

Reviewing the ValidatingRepository Class and the IValidator Implementation for the StockQuote Class

To review the ValidatingRepository class and the IValidator implementation for the StockQuote class

1. Open the ValidatingRepository.cs file in the PersistenceFramework project.

2. Examine the **ValidatingRepository** class.

```
public ValidatingRepository(
    IRepository<T> repository,
    IValidator<T> validator)
{
  ...
}
```

The constructor for the **ValidatingRepository<T>** class receives two parameters, a wrapped repository and a validator. For the purposes of this lab, a validator that relies on random values to determine whether an instance is valid will be used for stock quotes.

3. Open the RandomStockQuoteValidator.cs file in the StocksTicker project.

4. Examine the **RandomStockQuoteValidator** class. It is a non-generic class that implements the closed **IValidator<StockQuote>** generic interface.

Updating the Container Setup with API Calls

To update the container setup code with API calls

1. Open the **Program.cs** file in the **StocksTicker** project.

2. After configuration is applied to the container, use the **RegisterType** method to map the open **IRepository<>** generic interface to the open **ValidatingRepository<>** class with the "**validating**" name, as shown in the following code.

```
using (IUnityContainer container = new UnityContainer())
{
  container.LoadConfiguration();

  container
    .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating");

  StocksTickerPresenter presenter
          = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> Because open generic types cannot be used as generic type arguments, the version of the **RegisterType** method taking **Type** instances as parameters is used instead of the version with generic type parameters used previously. These versions are mostly equivalent, although the version with generic type parameters benefits from compile-time type checks.

3. Use the **RegisterType** method to map the closed **IValidator<StockQuote>** interface to the non-generic **RandomStockQuoteValidator** class, as shown in the following code.

```
using (IUnityContainer container = new UnityContainer())
{
  container.LoadConfiguration();

  container
        .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating")
        .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>();

  StocksTickerPresenter presenter
        = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

> This registration is necessary to resolve the **IValidator<StockQuote>** argument when resolving the **ValidatingRepository<StockQuote>** mapped with the "**validating**" name. Mixing registrations for open and closed generic types is possible; it is also possible to map closed generic types to non-generic types. However, any Resolve request for an **IRepository<T>** with name **validating** will require a proper registration of **IValidator<T>** to avoid a resolve failure.

Resolve objects using overrides

To resolve objects with overrides

1. Update the call to the **Resolve** method to supply a new **ParamterOverride** object for the "**repository**" parameter so that the repository with name "**validating**" is resolved and injected.

```
using (IUnityContainer container = new UnityContainer())
{
  container.LoadConfiguration();

  container
        .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating")
        .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>();

  StocksTickerPresenter presenter
        = container.Resolve<StocksTickerPresenter>(
            new ParameterOverride(
                "repository",
                new ResolvedParameter<IRepository<StockQuote>>("validating"))
                    .OnType<StocksTickerPresenter>());
```

```
    Application.Run((Form)presenter.View);
}
```

> The **ParameterOverride** object instructs the constructor to use the supplied value when injecting parameters with the supplied name. In this case the override indicates that the repository named **"validating"** should be resolved an injected for parameter **"repository"**. Values for all overrides follow the same rules as values for injection members like **InjectionConstructor**. Overrides apply to any object being resolved, not just the "top level" one. Invoking the **OnType** method on the parameter override constrains the override to a specific type (which may or may not be the type supplied in the **Resolved** call, which is **StocksTickerPresenter** in the example above).

Running the Application

Launch and use the application for some time. Look at the ui.log file, where you should find entries describing a **RepositoryException** thrown by the validating repository, as shown in the following code.

```
UI Information: 0 : Refresh timer elapsed
    DateTime=2009-02-16T18:02:28.1595997Z
UI Information: 0 : StockQuote for MSFT was updated
    DateTime=2009-02-16T18:02:29.4395997Z
UI Warning: 0 : Error saving the updated quote for 'MSFT': Invalid instance to save
    DateTime=2009-02-16T18:02:29.4805997Z
```

Task 3: Using Array Injection

In this task, a **CompositeLogger** will be injected to the resolved **StocksTickerPresenter** instance instead of the **TraceSourceLogger** used earlier. This logger requires an array of loggers and forwards the logging requests it receives to the elements in this array.

Although an array can be injected as a value, supplying it when configuring injection like any other CLR object, this approach is not always appropriate. Unity can be configured to inject an array, using the result of resolving other keys as elements.

There are two kinds of array injection:

- Injecting an array that contains all the instances of the array's element type registered in the container (in the order they were registered). This is the result of resolving the array type (such as **ILogger[]** in this case).

- Injecting an array containing the result of resolving specific keys.

The second approach will be used in this task.

Updating the Container Setup to Inject a CompositeLogger to the StocksTickerPresenter

To update the container setup to inject a CompositeLogger to the StocksTickerPresenter

1. Open the Program.cs file.

2. Use the **RegisterType** method to map the **ILogger** interface to the **CompositeLogger** class with the "**composite**" name, as shown in the following code.

```
using (IUnityContainer container = new UnityContainer())
{
  container.LoadConfiguration();

  container
        .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating")
        .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>()
        .RegisterType<ILogger, CompositeLogger>("composite");

  StocksTickerPresenter presenter
        = container.Resolve<StocksTickerPresenter>(
            new ParameterOverride(
                "repository",
                new ResolvedParameter<IRepository<StockQuote>>("validating"))
                    .OnType<StocksTickerPresenter>());

  Application.Run((Form)presenter.View);
}
```

> Without additional configuration, the default injection rules indicate that the container would attempt to resolve the argument for the **CompositeLogger**'s constructor of type **ILogger[]**. This would result in an attempt to resolve all the instances registered to the container with a name, which include the **CompositeLogger** being built in the first place, thus resulting in a **StackOverflowException** caused by unbounded recursion; this issue would not occur if the **CompositeLogger** had not been registered with a name, because only named instances are included when resolving an array.

3. Update the **RegisterType** call, mapping the **ILogger** interface to the **CompositeLogger** class to inject an array of specific instances through the constructor.

```
using (IUnityContainer container = new UnityContainer())
{
  container.LoadConfiguration();

  container
        .RegisterType(
```

```
                typeof(IRepository<>),
                typeof(ValidatingRepository<>),
                "validating")
            .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>()
            .RegisterType<ILogger, CompositeLogger>(
                "composite",
                new InjectionConstructor(
                    new ResolvedArrayParameter<ILogger>(
                        typeof(ILogger),
                        new ResolvedParameter<ILogger>("UI"))));

    StocksTickerPresenter presenter
            = container.Resolve<StocksTickerPresenter>(
                new ParameterOverride(
                    "repository",
                    new ResolvedParameter<IRepository<StockQuote>>("validating"))
                        .OnType<StocksTickerPresenter>);

    Application.Run((Form)presenter.View);
}
```

> The **ResolvedArrayParameter** works like any other **InjectionParameterValue** used to specify
> how to supply a constructor argument or a property value. In this case, the **ILogger[]**
> parameter will be injected with a two-element array (because the **ResolvedArrayParameter** is
> built with two arguments). The first element will be the result of resolving the **ILogger**
> interface without a name (mapped to the **ConsoleLogger** in the configuration file), indicated by
> the **typeof(ILogger)** argument (which is shorthand for **new ResolvedParameter<ILogger>()**),
> while the second element will be the result of resolving the **ILogger** interface with the "**UI**"
> name (again mapped in the configuration file), indicated by the explicit **new**
> **ResolvedParameter<ILogger>("UI")** expression. When using this kind of array injection, literal
> values can be specified as members of the array and as unnamed instances.

4. Update the call to the **Resolve** method to supply a new **PropertyOverride** object for the
    "Logger" property so that the logger with name **"composite"** is resolved and injected.

```
using (IUnityContainer container = new UnityContainer())
{
    container.LoadConfiguration();

    container
            .RegisterType(
                typeof(IRepository<>),
                typeof(ValidatingRepository<>),
                "validating")
            .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>()
            .RegisterType<ILogger, CompositeLogger>(
                "composite",
                new InjectionConstructor(
```

```
                    new ResolvedArrayParameter<ILogger>(
                        typeof(ILogger),
                        new ResolvedParameter<ILogger>("UI"))));

    StocksTickerPresenter presenter
          = container.Resolve<StocksTickerPresenter>(
              new ParameterOverride(
                  "repository",
                  new ResolvedParameter<IRepository<StockQuote>>("validating"))
                      .OnType<StocksTickerPresenter>(),
              new PropertyOverride("Logger",
              new ResolvedParameter<ILogger>("composite")));

    Application.Run((Form)presenter.View);
}
```

> The **PropertyOverride** is not limited to a single type, so it will apply to all resolved objects with a **"Logger"** property.

Running the Application

Launch and use the application. All entries will be logged to both the ui.log file (located in the StocksTicker\bin\Debug folder) and the console.

To verify you have completed the lab correctly, you can use the solution provided in the Lab04\end\StocksTicker folder.

Lab 5: Integrating with ASP.NET and Child Containers

Estimated time to complete this lab: **20 minutes**

Introduction

In this lab, you will practice using Unity with an ASP.NET application. Because the creation of ASP.NET objects is managed by the ASP.NET infrastructure, the Unity container cannot be used to create them. Instead, the container will be used to apply injection to existing objects using the **BuildUp** method. For information about the **BuildUp** methods, see the topic "Using BuildUp to Wire Up Objects Not Created by the Container" in the Unity 2.0 documentation at http://msdn.microsoft.com/en-us/library/ff660934(v=PandP.20).aspx.

To begin, open the StocksTicker.sln file located in the Lab05\begin\StocksTicker folder.

Reviewing the Application

The application is a simplified ASP.NET version of the stocks ticker application used throughout these labs. Just like with the Windows® Forms version of the application, stock symbols can be subscribed to, and the latest updates on the stocks are displayed in a table, as illustrated in the following figure.
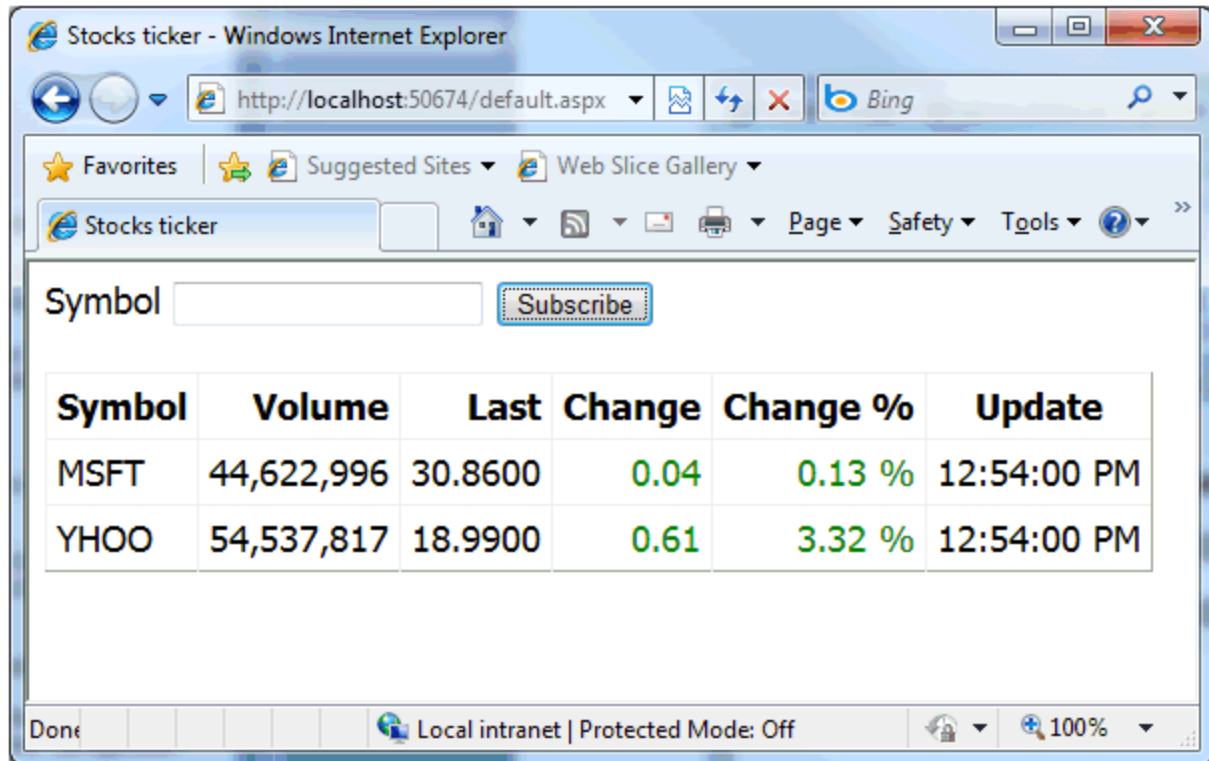
Task 1: Integrating with ASP.NET

In this task, the application will be updated to use a single, application-wide container to perform injection on its Web forms before they process requests.

Adding References to the Required Assemblies

To add references to the required assemblies

1. Add references to the **Microsoft.Practices.Unity** and **Microsoft.Practices.Unity.Configuration** assemblies from the Lib folder of the labs.

2. Add a reference to the .NET Framework's **System.Configuration** assembly.

Updating the Default Page to Use Property Injection

To update the default page to use property injection

1. Open the Default.aspx.cs file. If it is not visible, expand the node for the Default.aspx file.

2. Add a **using** directive for the **Unity** namespace.

   ```
   using Microsoft.Practices.Unity;
   ```

3. Remove the initialization code from the **Page_Load** method, as shown in the following highlighted code.

```
protected void Page_Load(object sender, EventArgs e)
{
  this.stockQuoteService = new MoneyCentralStockQuoteService();
  if (!this.IsPostBack)
  {
    UpdateQuotes();
  }
}
```

4. Add the **Dependency** attribute to the **StockQuoteService**, as shown in the following highlighted code.

```
private IStockQuoteService stockQuoteService;
[Dependency]
public IStockQuoteService StockQuoteService
{
  get { return stockQuoteService; }
  set { stockQuoteService = value; }
}
```

> Configuration files cannot be used in this case because the assembly name for the page class is not known in advance so assembly-qualified type names cannot be expressed.

Adding a Global.asax File to Manage the Container

The **HttpApplication** class is used to define common methods in an ASP.NET application, so the code to manage the Unity container will be added to a new Global.asax file. For information about HTTP application classes, see HttpApplication Class.

To add a Global.asax file to manage the container

1. Add a Global.asax file. Right-click the **StocksTicker** project node, point to **Add**, and then click **New Item**. Select the **Global Application Class** template, and then click **Add**.
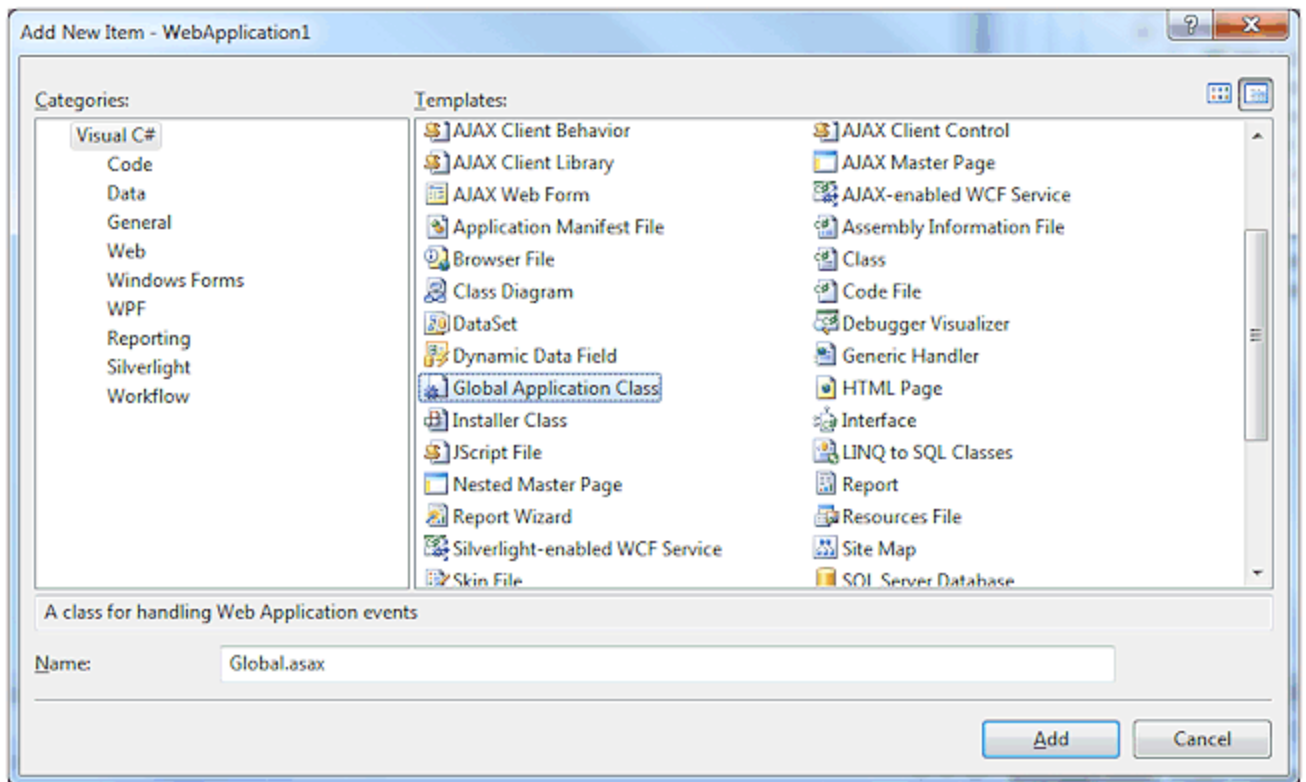
2. Add **using** statements for the required namespaces:

```
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.Configuration;
using System.Web.Configuration;
using System.Web.UI;
```

3. Add a constant named **AppContainerKey** with **"application container"** as its value as shown in the following highlighted code.

```
public class Global : System.Web.HttpApplication
{
  private const string AppContainerKey = "application container";

  protected void Application_Start(object sender, EventArgs e)
  {
    ...
  }
```

4. Add the following code to implement an **ApplicationContainer** property to store a Unity container in the application state using the key defined earlier.

```
private IUnityContainer ApplicationContainer
{
  get
  {
    return (IUnityContainer)this.Application[AppContainerKey];
```

```
    }
    set
    {
      this.Application[AppContainerKey] = value;
    }
  }
}
```

> The container is stored in the application's shared state.

5. Update the empty **Application_Start** method to create a Unity container, configure it with the information for the **"application"** container from the configuration file and set it as the value for the **ApplicationContainer** property as shown in the following highlighted code.

```
protected void Application_Start(object sender, EventArgs e)
{
  IUnityContainer applicationContainer = new UnityContainer();
  applicationContainer.LoadConfiguration("application");
  ApplicationContainer = applicationContainer;
}
```

6. Update the empty **Application_End** method to dispose the application's container as shown in the following highlighted code.

```
protected void Application_End(object sender, EventArgs e)
{
  IUnityContainer applicationContainer = this.ApplicationContainer;
  if (applicationContainer != null)
  {
    applicationContainer.Dispose();
    this.ApplicationContainer = null;
  }
}
```

7. Add the following code to implement a method named **Application_PreRequestHandlerExecute** to intercept the ASP.NET execution pipeline and use the container to inject dependencies into the request handler if the handler is a **Page**.

```
protected void Application_PreRequestHandlerExecute(
                              object sender, EventArgs e)
{
  Page handler = HttpContext.Current.Handler as Page;

  if (handler != null)
  {
    IUnityContainer container = ApplicationContainer;

    if (container != null)
    {
      container.BuildUp(handler.GetType(), handler);
    }
```

```
    }
}
```

Adding Unity Configuration to the Web.config File

The configuration in a Web.config file uses the same schema as in an App.config file.

To add Unity configuration to the Web.config file

1. Open the Web.config file.

2. Add a declaration for the **unity** configuration section.

    ```
    <configSections>
      ...
      <section name="unity"
    type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
    Microsoft.Practices.Unity.Configuration, Version=2.0.414.0, Culture=neutral,
    PublicKeyToken=31bf3856ad364e35"/>
    </configSections>
    ```

3. Add the **unity** section element.

    ```
      </configSections>
      ...
      <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
      </unity>
    ```

4. Add **alias** elements for the types used throughout the labs.

    ```
    <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
      <alias alias="TraceSource" type="System.Diagnostics.TraceSource, System,
            Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
      <alias alias="ILogger" type="StocksTicker.Loggers.ILogger, StocksTicker"/>
      <alias alias="TraceSourceLogger"
            type="StocksTicker.Loggers.TraceSourceLogger, StocksTicker"/>
      <alias alias="IStockQuoteService"
            type="StocksTicker.StockQuoteServices.IStockQuoteService,
                StocksTicker"/>
      <alias alias="MoneyCentralStockQuoteService"
            type="StocksTicker.StockQuoteServices.MoneyCentralStockQuoteService,
                StocksTicker"/>
    </unity>
    ```

5. Add a **container** element with the name **application**.

    ```
      <alias alias="MoneyCentralStockQuoteService"
            type="StocksTicker.StockQuoteServices.MoneyCentralStockQuoteService,
                StocksTicker"/>
      <container name="application">
      </container>
    </unity>
    ```

6. Add a **register** element to map the **IStockQuoteService** interface to the
   MoneyCentralStockQuoteService class with a **property** element to inject the **Logger** property
   and define a singleton lifetime manager.

```
<container name="application">
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <lifetime type="singleton"/>
    <property name="Logger"/>
  </register>
</container>
```

7. Add a **register** element to map the **ILogger** interface to the **TraceSourceLogger** class, defining a
   singleton lifetime manager and injecting the "**default**" string in its **traceSourceName**
   constructor parameter. This mapping is required to inject the **Logger** property on the
   MoneyCentralStockQuoteService instance.

```
<container name="application">
  <register type="ILogger" mapTo="TraceSourceLogger">
    <lifetime type="singleton"/>
    <constructor>
      <param name="traceSourceName">
        <value value="default"/>
      </param>
    </constructor>
  </register>
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <lifetime type="singleton"/>
    <property name="Logger"/>
  </register>
</container>
```

Running the Application

To run the application

1. Launch the application. Open two browser instances, as shown in the following figure, and open
   the application's URL in them both. Use the application for a while in both browsers, subscribing
   to different sets of symbols.
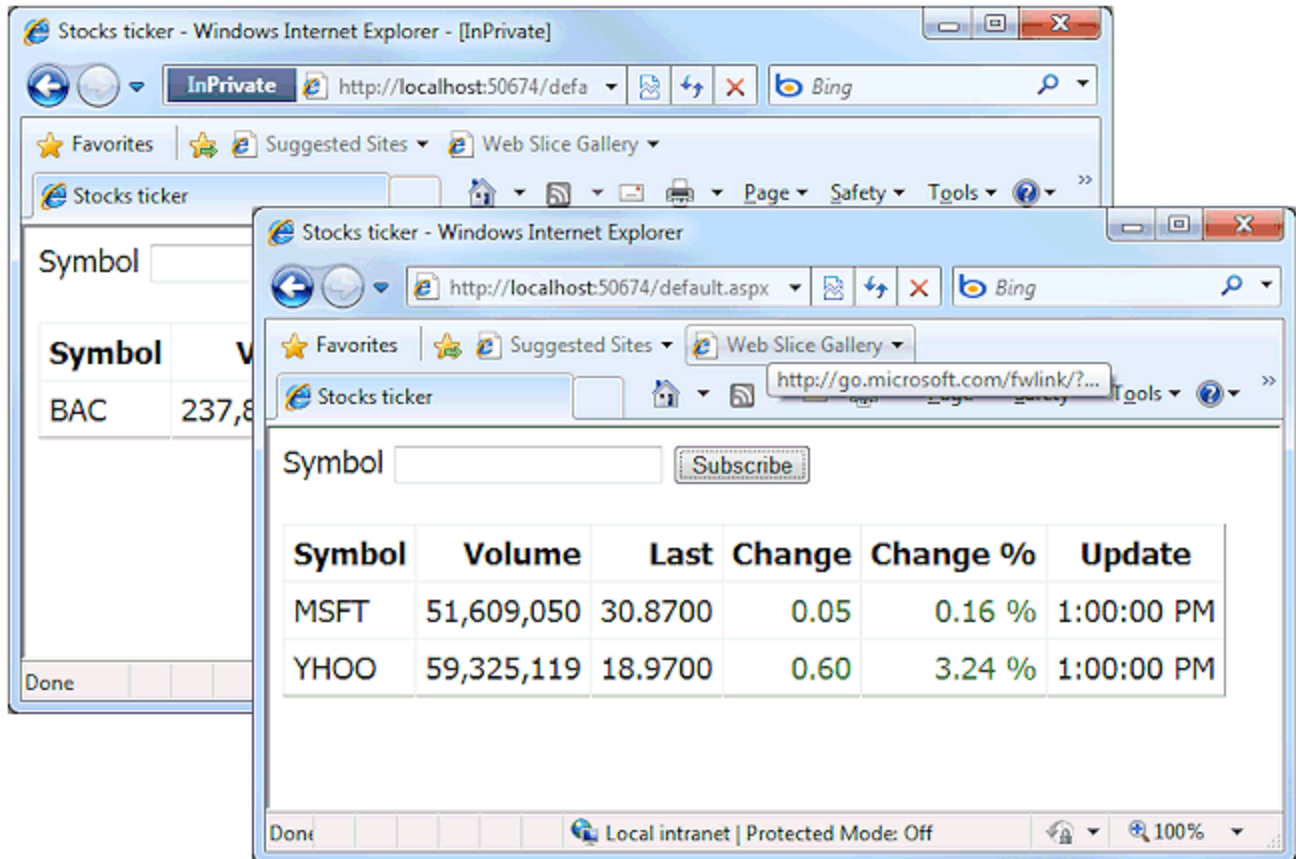
2. Close the browsers and stop the development server. To do this, right-click the Visual Studio development Web server icon 📄 in the notification area of the taskbar and click **Stop**.

3. Open the trace.log file located in the StocksTicker folder. The contents at the bottom of the file should look like the following log, with messages indicating the processing of the different sets of requests and the final two entries indicating that the service and the logger were disposed.

```
default Information: 0 : Parsed result for YHOO
    DateTime=2009-02-23T18:15:25.4761924Z
default Information: 0 : Retrieving quotes for MSFT
    DateTime=2009-02-23T18:15:29.1101924Z
default Information: 0 : Received result for MSFT
    DateTime=2009-02-23T18:15:29.5251924Z
default Information: 0 : Parsed result for MSFT
    DateTime=2009-02-23T18:15:29.5251924Z
default Information: 0 : Shutting down service
    DateTime=2009-02-23T18:15:44.6961924Z
default Information: 0 : Shutting down logger
    DateTime=2009-02-23T18:15:44.6971924Z
```

Task 2: Using Per-Session Child Containers

In this task, a more sophisticated container management strategy will be implemented: the application-wide container will hold services shared by all sessions, but other objects will be managed by a session-specific container. In this case, the service used to retrieve quotes will not be shared, but its lifetime will be managed. Of course, this approach works only when session state is enabled and stored InProc.

Container hierarchies can be used to control the scope and lifetime of objects and to register different mappings in different context. In this task, a two-level hierarchy will be implemented. For information about container hierarchies, see "Using Container Hierarchies" at http://msdn.microsoft.com/en-us/library/ff660895(v=PandP.20).aspx.

Updating the Global.asax File to Manage Per-Session Containers

To update the Global.asax file to manage per-session containers

1. Open the Global.asax.cs file.

2. Add a constant named **SessionContainerKey** with **"session container"** as its value as shown in the following highlighted code.

    ```
    public class Global : System.Web.HttpApplication
    {
      private const string AppContainerKey = "application container";
      private const string SessionContainerKey = "session container";
    ```

3. Add the following code to implement a **SessionContainer** property to store a Unity container in the application state using the key defined earlier.

    ```
    private IUnityContainer SessionContainer
    {
      get
      {
        return (IUnityContainer)this.Session[SessionContainerKey];
      }
      set
      {
        this.Session[SessionContainerKey] = value;
      }
    }
    ```

4. Add the following highlighted code to the **Session_Start** method to create a child container of the application's container using the **CreateChildContainer** method, configure it with the information for the **"session"** container from the configuration file and set it as the value for the **SessionContainer** property.

    ```
    protected void Session_Start(object sender, EventArgs e)
    {
      IUnityContainer applicationContainer = this.ApplicationContainer;
    ```

```
    if (applicationContainer != null)
    {
        IUnityContainer sessionContainer
                = applicationContainer.CreateChildContainer();
        sessionContainer.LoadConfiguration("session");

        this.SessionContainer = sessionContainer;
    }
}
```

5. Add the following highlighted code to the **Session_End** method to dispose the session's container.

```
protected void Session_End(object sender, EventArgs e)
{
    IUnityContainer sessionContainer = this.SessionContainer;
    if (sessionContainer != null)
    {
        sessionContainer.Dispose();
        this.SessionContainer = null;
    }
}
```

6. Update the **Application_PreRequestHandlerExecute** method to use the session container to **BuildUp** the request's handler as shown in the following highlighted code.

```
protected void Application_PreRequestHandlerExecute(
                                    object sender, EventArgs e)
{
  Page handler = HttpContext.Current.Handler as Page;

  if (handler != null)
  {
    IUnityContainer container = SessionContainer;

    if (container != null)
    {
      container.BuildUp(handler.GetType(), handler);
    }
  }
}
```

Updating the Unity Configuration with a Session Container

To update the Unity configuration with a session container

1. Open the Web.config file.

2. Add a new **container** element with name **session**.

```
      </container>
      <container name="session">
      </container>
  </unity>
```

3. Add a **register** element to map the **IStockQuoteService** interface to the **MoneyCentralStockQuoteService** class with a **property** element to inject the **Logger** property and define a singleton lifetime manager.

```
<container name="session">
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <lifetime type="singleton"/>
    <property name="Logger"/>
  </register>
</container>
```

4. Remove the **register** element mapping the **IStockQuoteService** interface from the **application** container element.

```
<container name="application">
  <register type="ILogger" mapTo="TraceSourceLogger">
    <lifetime type="singleton"/>
    <constructor>
      <param name="traceSourceName">
        <value value="default"/>
      </param>
    </constructor>
  </register>
  <register type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <lifetime type="singleton"/>
    <property name="Logger"/>
  </register>
</container>
```

Running the Application

To run the application

1. Launch the application. Open two browser instances and open the application's URL in them. Use the application for a while in both browsers, subscribing to different sets of symbols.

2. Close one of the browser instances and wait for 90 seconds. The session timeout interval is set to one minute in the configuration file, so this wait should be enough for the session to expire.

3. Close the other browser instance, and then stop the development Web server.

4. Open the trace.log file located in the StocksTicker folder. The contents at the bottom of the file should look like the following log, with entries for each of the service instances and for the shared logger.

```
default Information: 0 : Retrieving quotes for MSFT BAC
     DateTime=2009-02-23T19:06:45.7471924Z
default Information: 0 : Received result for MSFT BAC
     DateTime=2009-02-23T19:06:46.1491924Z
default Information: 0 : Parsed result for MSFT BAC
     DateTime=2009-02-23T19:06:46.1491924Z
```
default Information: 0 : Shutting down service
     DateTime=2009-02-23T19:07:00.0781924Z
```
default Information: 0 : Retrieving quotes for MSFT BAC
     DateTime=2009-02-23T19:07:05.7731924Z
default Information: 0 : Received result for MSFT BAC
     DateTime=2009-02-23T19:07:06.1841924Z
default Information: 0 : Parsed result for MSFT BAC
     DateTime=2009-02-23T19:07:06.1841924Z
default Information: 0 : Retrieving quotes for MSFT BAC
     DateTime=2009-02-23T19:07:25.7691924Z
default Information: 0 : Received result for MSFT BAC
     DateTime=2009-02-23T19:07:26.1791924Z
default Information: 0 : Parsed result for MSFT BAC
     DateTime=2009-02-23T19:07:26.1791924Z
```
default Information: 0 : Shutting down service
     DateTime=2009-02-23T19:07:44.2521924Z
default Information: 0 : Shutting down logger
     DateTime=2009-02-23T19:07:44.2541924Z

To verify you have completed the lab correctly, you can use the solution provided in the Lab05\end\StocksTicker folder.

Copyright