Unity Application Block Hands-On Lab for Enterprise Library

This walkthrough should act as your guide to learn about the Unity Application Block and 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

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 Labs\Lab01\begin\StocksTicker folder.

Running the Application

You should first launch the application. To do this, click **Start Without Debugging** on the **Debug** menu. 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. While the **Refresh** check box is selected, the form displays the latest information about each of the subscribed stock symbols; it flashes each time an update for a stock symbol is received.

Figure 1 illustrates a sample stocks ticker application.

Figure 1

The stocks ticker application

Note: **I**n the application's default configuration, 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 and 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 using the Model-View-Presenter (MVP) pattern. For information about the MVP pattern, see Separated Presentation on MSDN. Figure 2 shows the classes involved in the application

and their relationships.

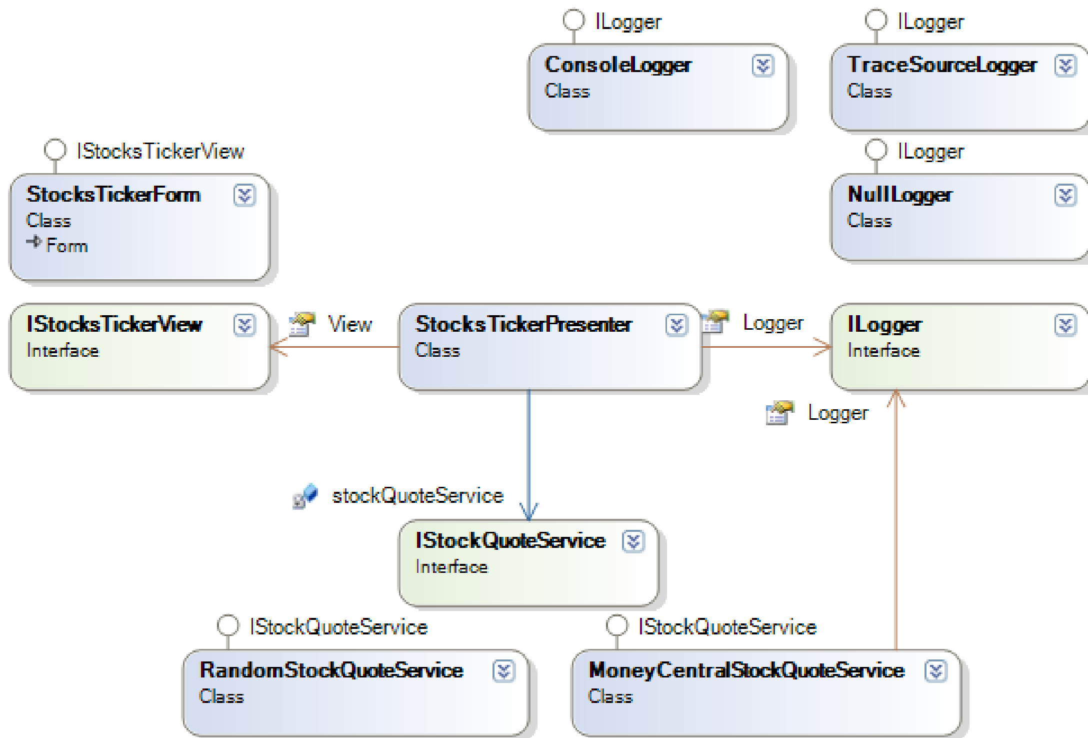


Figure 2

Simplified class diagram for the stocks ticker application

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

All classes in the solution, including the presenter class, have constructors taking all 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, while a 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). Replacing the explicit creation of these objects with the use of a Unity container is the purpose of this lab.

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, instead of relying on 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, as shown in Figure 3. Select the **Microsoft.Practices.Unity** and **Microsoft.Practices.ObjectBuilder2** assemblies, and then click **OK**.

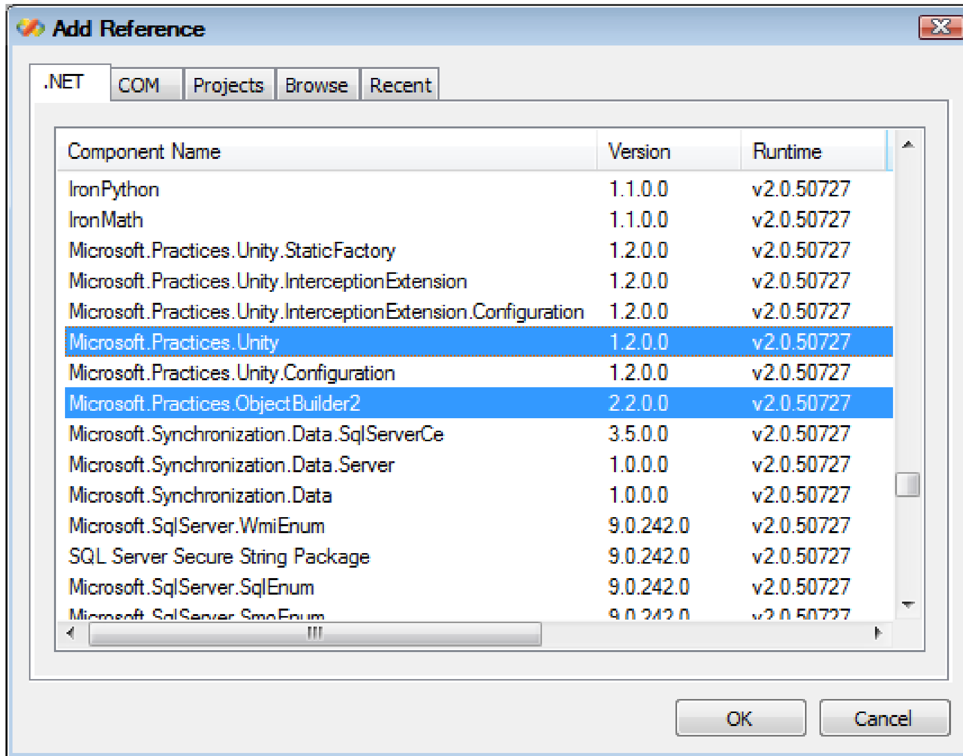


Figure 3

Add Reference dialog box

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;
```

Replace the creation of the instances (view, presenter, service, loggers) with a **Resolve** request for the **StocksTickerPresenter** to a new **UnityContainer**, as shown in the following 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, so the new code takes advantage of the **using** statement to dispose the new container; Lab 2 elaborates on the result of disposing a container.

> **Note:** These changes are enough to successfully build the application, and even though the Unity container can figure out, to some extent, how to build objects using default rules ("autowiring"), additional set up is required before the container can successfully resolve this application's objects.

3. Debug the application. To do this, click **Start Debugging** on the **Debug** menu. 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 **[StocksTicker.UI.StocksTickerPresenter, null]**. Keys always consist of a type and a name (which defaults to **null** when it is not supplied).
- The container determined that the constructor for **StocksTickerPresenter** with the signature **(IStocksTickerView view, IStockQuoteService stockQuoteService)** should be used to create the object, following the autowiring rules.
- The root cause for 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, represented 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 Annotating Objects for Constructor Injection on MSDN.

4. Add the type mappings required by the container to resolve the interfaces involved using the **RegisterType** method, as shown in the following code.

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

```
        .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>();

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

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

The **RegisterType** method is the main mechanism used to set up a container; it can be used to map abstract types to concrete ones as described in this step, but it can also be used to override the default injection rules and specify lifetime managers. The next exercises will show additional uses of the method. For details about mapping types see Dependency Injection Types and Mappings.

> **Note: T**here are no mappings for the **ILogger** interface; they are not required at this point because properties are not injected unless they are explicitly configured to be so. The application will run now because the involved objects' required collaborators can be resolved by the container, but logging will not be available because loggers will not be injected into the resolved objects (recall that 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, they 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 want properties to be set by the container, in addition to passing in dependencies via 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 code.

```
private ILogger logger;
[Dependency]
```

```
public ILogger Logger
{
    get { return logger; }
    set { logger = value; }
}
```

Note: The **Dependency** attribute can be used to indicate that the property should be injected with the result of resolving the property's type, **ILogger**, in the container. For information about property (setter) injection, see Annotating Objects for Property (Setter) Injection.

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 case, the interface will be mapped to the **ConsoleLogger** class 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 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);
}
```

Note: 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, the same **Dependency** attribute is used. However, because a different logger instance is to be injected, a way to differentiate the loggers is necessary. The Unity container lets you register the same type multiple times, but give each one a different name. Then when that dependency is resolved, the name can be used to specify exactly which instance you want. In this case, 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 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 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);
```

```
}
```

> **Note:** The name parameter for the **RegisterType** method is optional, and it defaults to **null** when it is not supplied.

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**. Drilling into the chain of **InnerExceptions** shows that the root cause of the failure is represented 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

An attribute will be used to indicate which among 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** to the constructor taking a **TraceSource** as its sole parameter, as shown in the following 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 will kick in 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 prove tricky because 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 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);
}
```

Note: The container has generic and non-generic versions of the **RegisterInstance** method. In this case, the generic version is used, relying on the compiler's inference to avoid specifying the generic type argument. This works in this case 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, particularly when names are involved 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 Labs\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 setting up lifetime managers.

To begin, open the StocksTicker.sln file located in the Labs\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 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** will be used in this lab. For information about configuring injection in a Unity container, see [Configuring Constructor, Property, and Method Injection]() on MSDN.

To update container configuration to override the default injection rules

1. Update the **RegisterType** call for the **IStockQuoteService** interface to inject the **Logger** property using the **InjectionProperty** object, as shown in the following 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);
}
```

Note: 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 use of the **Dependency** attribute without further configuration to annotate the property in the previous lab.

2. Use the **RegisterType** method to set up the injection of the **StocksTickerPresenter** class, as shown in the following 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);
}
```

Note: In this case, the **RegisterType** call is not used to perform a mapping; instead, 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 enough 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**.

3. Update the **RegisterType** call for the **ILogger** interface with the "**UI"** name to inject invoke the constructor with a single string parameter, as shown in the following 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);
}
```

Note: 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 will be called, passing the "**UI**" value as its argument, 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 Configuring Constructor, Property, and Method Injection.

4. 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);
}
```

> **Note:** The container needed to be instructed on how to resolve an instance of the **TraceSource** class 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 instructed on how 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 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 code.

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

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

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

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

4. Add the **Dispose** method to implement the **IDisposable** interface, as shown in the following code.

```
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**, as shown in the following 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);
}
```

Note: A lifetime manager is an optional parameter of the **RegisterType** type. For information about lifetime managers in general, and the **ContainerControlledLifetimeManager** in particular, see Using Lifetime Managers.

Running the Application

Launch the application, use it, and finally 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 Labs\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 Labs\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 using directives for the **System.Configuration** and **Unity.Configuration** namespaces.

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

3. Retrieve the Unity configuration section from the configuration file and use it to configure the container, replacing the calls to **RegisterType**, as shown in the following code.

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

```
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

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

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

The **ConfigurationManager** class is used to retrieve configuration from a configuration file as a graph of .NET Framework objects.

The preceding code obtains the **UnityConfigurationSection** representing the information from the configuration file using the "**unity**" name and indicates the default container configuration (obtained with the **config.Containers.Default** expression) to configure the container, resulting in a sequence of API calls.

For information about how to load configuration information into a container, see Loading Configuration Information into a Container.

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 and Source Schema for the Unity Application Block, see Unity Configuration Schematic.

To update the configuration file with the container configuration

1. Open the App.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=1.2.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
</configSections>
```

> **Note:** The name "**unity**" for the configuration section is only a convention; any name works, as long as it matches the name used to retrieve the section at run time.

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

```
  </system.diagnostics>
  <unity>
  </unity>
</configuration>
```

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

4. Add **typeAlias** elements for the types used throughout the exercises.

```
<unity>
  <typeAliases>
    <typeAlias
      alias="string"
      type="System.String, mscorlib" />
    <typeAlias
      alias="TraceSource"
      type="System.Diagnostics.TraceSource, System, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" />

    <typeAlias
      alias="singleton"
      type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager,
Microsoft.Practices.Unity, Version=1.2.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />

    <typeAlias
      alias="ILogger"
      type="StocksTicker.Loggers.ILogger, StocksTicker" />
    <typeAlias
      alias="ConsoleLogger"
      type="StocksTicker.Loggers.ConsoleLogger, StocksTicker" />
    <typeAlias
      alias="TraceSourceLogger"
      type="StocksTicker.Loggers.TraceSourceLogger, StocksTicker" />

    <typeAlias
      alias="IStockQuoteService"
      type="StocksTicker.StockQuoteServices.IStockQuoteService, StocksTicker"
/>
    <typeAlias
      alias="MoneyCentralStockQuoteService"
      type="StocksTicker.StockQuoteServices.MoneyCentralStockQuoteService,
StocksTicker" />

    <typeAlias
      alias="IStocksTickerView"
      type="StocksTicker.UI.IStocksTickerView, StocksTicker" />
    <typeAlias
      alias="StocksTickerForm"
```

```
      type="StocksTicker.UI.StocksTickerForm, StocksTicker" />
    <typeAlias
      alias="StocksTickerPresenter"
      type="StocksTicker.UI.StocksTickerPresenter, StocksTicker" />
  </typeAliases>
</unity>
```

Note: Type aliases are not required, but they typically make the configuration files less verbose and easier to read. Each **typeAlias** element contains an **alias**, which can be used in most places in Unity's configuration schema where a type name is expected, and the **type** name the alias is intended to represent. Shorter versions of the type names can be used as aliases, but this is not required; the "singleton" alias is typically used for Unity's **Microsoft.Practices.Unity.ContainerControlledLifetimeManager,** but it is not part of the type name.

5. Add elements for the **containers** collection and the default (unnamed) **container**.

```
  </typeAliases>

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

Note: The default container does not differ from named containers other than it the slightly easier way it can be retrieved from the configuration objects through the **Default** property.

6. Add a **types** element to default container.

```
<containers>
  <container>
    <types>
    </types>
  </container>
</containers>
```

Note: The children of the **types** element correspond to calls to the **RegisterType** method group.

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

```
<types>
  <type type="IStocksTickerView" mapTo="StocksTickerForm"/>
</types>
```

Note: Notice how aliases are used for type names, although actual type names could have also been used.

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

```
<types>
  <type type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <typeConfig>
      <property name="Logger" propertyType="ILogger"/>
    </typeConfig>
  </type>
</types>
```

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

```
<types>
  <type type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    <typeConfig>
      <property name="Logger" propertyType="ILogger"/>
    </typeConfig>
  </type>
  <type type="ILogger" mapTo="ConsoleLogger"/>
</types>
```

10. Add a **type** element to map the **ILogger** interface to the **TraceSourceLogger** for the "**UI**" name, with a child **typeConfig** element to indicate the use of the single-string parameter constructor to build the instance, passing the "UI" string as the argument with the **value** element, and with a child **lifetime** element to indicate the use of the **ContainerControlledLifetimeManager** through the use of the **singleton** alias.

```
<types>
  <type type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    ...
  </type>
  <type type="ILogger" mapTo="ConsoleLogger"/>
  <type name="UI" type="ILogger" mapTo="TraceSourceLogger">
    <lifetime type="singleton"/>
    <typeConfig>
      <constructor>
        <param name="sourceName" parameterType="string">
          <value value="UI"/>
        </param>
      </constructor>
    </typeConfig>
  </type>
</types>
```

> **Note:** The **constructor** element is equivalent to the **InjectionConstructor** class used in the previous lab. The **value** element indicates that the contents of its **value** attribute 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 value Element in the configuration schema.

11. Add a **type** element for the **StocksTickerPresenter** class with a child **typeConfig** element to configure the **Logger** property to be injected with the value of resolving the **ILogger** interface, with the "**UI**" name (which requires using the **dependency** element).

```
<types>
  <type type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    ...
  </type>
  <type type="ILogger" mapTo="ConsoleLogger"/>
  <type name="UI" type="ILogger" mapTo="TraceSourceLogger">
    ...
  </type>
  <type type="StocksTickerPresenter">
    <typeConfig>
      <property name="Logger" propertyType="ILogger">
        <dependency name="UI"/>
      </property>
    </typeConfig>
  </type>
</types>
```

> **Note:** Notice how "**UI**" is used to indicate the name of the instance to resolve, although it is used as a literal string value in the previous step. Also note the absence of the **mapTo** attribute in the **type** 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** element used in the previous step; for details on the **dependency** element see dependency Element in the configuration schema.

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 Labs\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 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 Labs\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, assuming 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 **type** element maps the closed generic interface to an implementation, which happens to be a closed generic class, using aliases.

```
<type type="IRepositoryOfStockQuote" mapTo="DebugRepositoryOfStockQuote"/>
```

The closed-generic-type aliases required to support the type element are defined as shown in the following code.

```
<typeAlias
  alias="IRepositoryOfStockQuote"
  type="PersistenceFramework.IRepository`1[[StocksTicker.StockQuote, StocksTicker]],
PersistenceFramework" />
<typeAlias
  alias="DebugRepositoryOfStockQuote"
  type="PersistenceFramework.DebugRepository`1[[StocksTicker.StockQuote,
StocksTicker]], PersistenceFramework" />
```

For information about how to specify type names involving generics, see the samples included in the reference documentation for the Type.GetType Method.

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. Add aliases for the open generic types **IRepository<>** and **DebugRepository<>**.

```
  <typeAlias
    alias="IRepositoryOfStockQuote"
    type="PersistenceFramework.IRepository`1[[StocksTicker.StockQuote, 
StocksTicker]], PersistenceFramework" />
  <typeAlias
    alias="DebugRepositoryOfStockQuote"
    type="PersistenceFramework.DebugRepository`1[[StocksTicker.StockQuote, 
StocksTicker]], PersistenceFramework" />
  <typeAlias
    alias="IRepository"
    type="PersistenceFramework.IRepository`1, PersistenceFramework" />
  <typeAlias
    alias="DebugRepository"
    type="PersistenceFramework.DebugRepository`1, PersistenceFramework" />
</typeAliases>
```

Compare the original entries for the closed generic types with the new entries for the open generic types.

3. Replace the **type** element mapping the closed **IRepository** interface to the closed **DebugRepository** class with a new element mapping the open types using the aliases defined in the preceding step.

```
<types>
  <type type="IStocksTickerView" mapTo="StocksTickerForm"/>
  <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
    ...
  </type>
  <type type="IRepository" mapTo="DebugRepository"/>
  <type type="ILogger" mapTo="ConsoleLogger"/>
  <type name="UI" type="ILogger" mapTo="TraceSourceLogger">
    ...
  </type>
  <type type="StocksTickerPresenter">
    ...
  </type>
</types>
```

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 Figure 4.

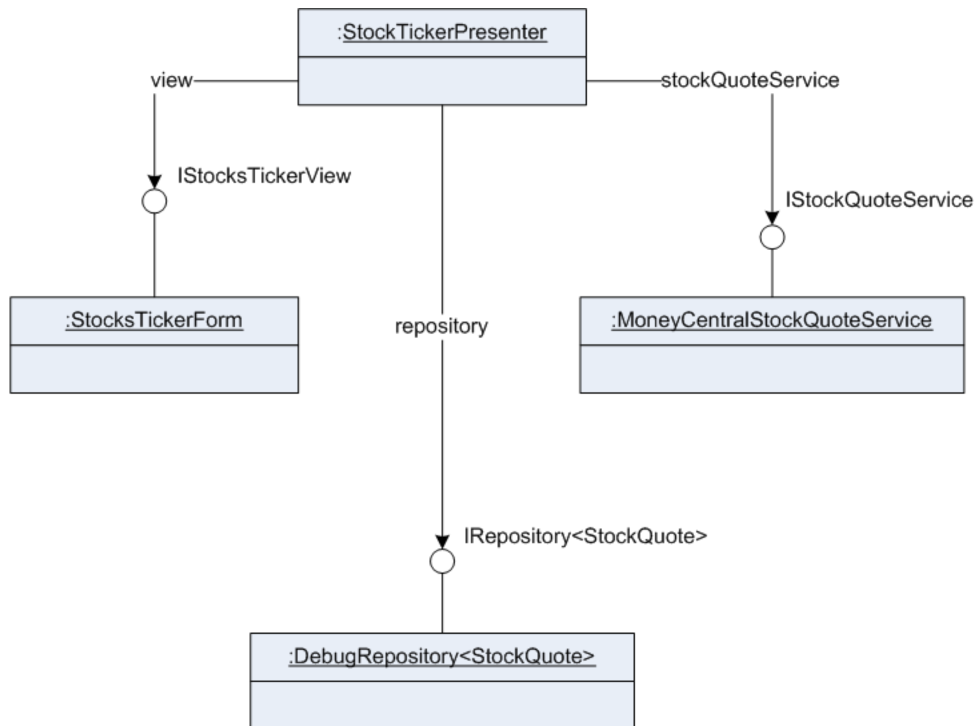


Figure 4
StocksTicker UML Object Diagram

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 Figure 5.

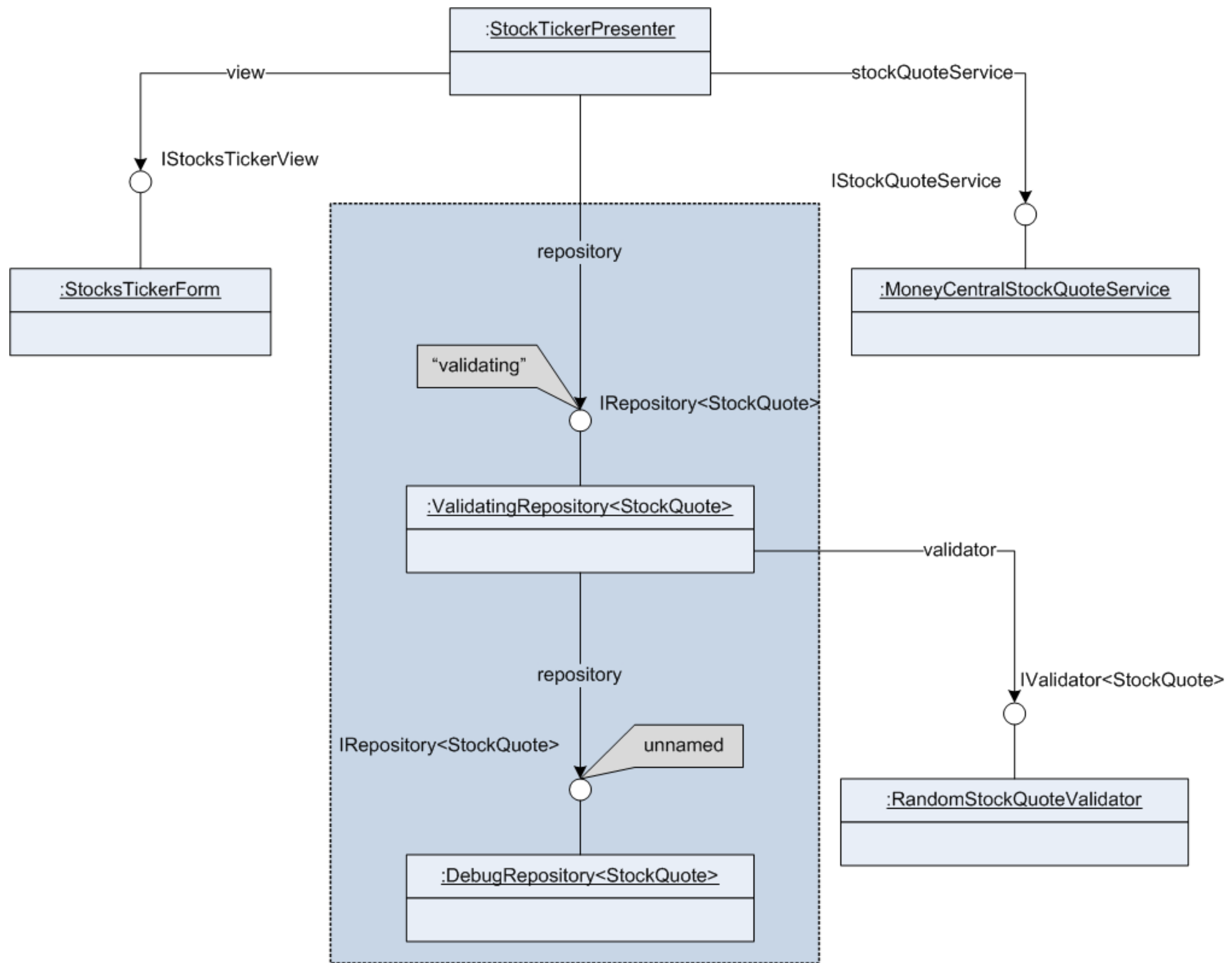


Figure 5

After Decorator added

In order to accomplish this, the named registration feature of the container will be used. The named used in each registration are shown in the callout tags in Figure 5.

In this task, the settings from the configuration file will be partially overridden in the application code. 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 Code 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 set up the injection of the **StocksTickerPresenter** class, as shown in the following code.

```
using (IUnityContainer container = new UnityContainer())
{
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

    container
        .RegisterType<StocksTickerPresenter>(
            new InjectionConstructor(
                typeof(IStocksTickerView),
                typeof(MoneyCentralStockQuoteService),
                new
ResolvedParameter<IRepository<StockQuote>>("validating")));

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

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

```
}
```

This constructor injection specification does not override settings from the configuration file, but it does override the default injection rules. All three **StocksTickerPresenter** constructor arguments are references that must be resolved; for the first two arguments, the unnamed instances are to be injected so types are supplied, but in the case of the third argument, the "**validating"** name will be used to resolve the **IRepository<StockQuote>** interface so an explicit **ResolveParameter** object is necessary. Keep in mind that using types to indicate that arguments should be resolved without names is not strictly necessary, but it makes code less verbose.

> **Note:** This is a partial injection specification for the **StocksTickerPresenter** class; the specification for the injection of its **Logger** property will still be supplied by the settings in the configuration file.

3. 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())
{
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

    container
        .RegisterType<StocksTickerPresenter>(
            new InjectionConstructor(
                typeof(IStocksTickerView),
                typeof(MoneyCentralStockQuoteService),
                new ResolvedParameter<IRepository<StockQuote>>("validating")))
        .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating");

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

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

> **Note:** 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.

The "**validating"** name ties this presenter's injection configuration to the new mapping.

4. 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())
{
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

    container
        .RegisterType<StocksTickerPresenter>(
            new InjectionConstructor(
                typeof(IStocksTickerView),
                typeof(IStockQuoteService),
                new ResolvedParameter<IRepository<StockQuote>>("validating")))
        .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating")
        .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>();

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

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

Note: 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.

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. Update the **RegisterType** call for the **StocksTickerPresenter** class to inject the **Logger** property, as shown in the following code.

```
using (IUnityContainer container = new UnityContainer())
{
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

    container
        .RegisterType<StocksTickerPresenter>(
            new InjectionConstructor(
                typeof(IStocksTickerView),
                typeof(MoneyCentralStockQuoteService),
                new ResolvedParameter<IRepository<StockQuote>>("validating")),
            new InjectionProperty(
```

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

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

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

Note: Injection of properties can be specified simultaneously in the configuration file and through code. In this case, the injection specification for the **Logger** property using the API overrides the previous specification from the configuration file, but non-overlapping definitions, which are not shown here, will be observed regardless of their origin because the underlying mechanism is the same. However, injection of properties indicated through attributes in the resolved types is observed only when there are no other injection specifications for any property in the same type.

3. 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())
{
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

    container
        .RegisterType<StocksTickerPresenter>(
            new InjectionConstructor(
                typeof(IStocksTickerView),
                typeof(MoneyCentralStockQuoteService),
                new ResolvedParameter<IRepository<StockQuote>>("validating")),
            new InjectionProperty(
                "Logger",
                new ResolvedParameter<ILogger>("composite")))
        .RegisterType(
            typeof(IRepository<>),
            typeof(ValidatingRepository<>),
            "validating")
        .RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>()
```

```
        .RegisterType<ILogger, CompositeLogger>(
            "composite");

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

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

Note: 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.

4. 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())
{
    UnityConfigurationSection config
        = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
    if (config != null)
    {
        config.Containers.Default.Configure(container);
    }

    container
        .RegisterType<StocksTickerPresenter>(
            new InjectionConstructor(
                typeof(IStocksTickerView),
                typeof(MoneyCentralStockQuoteService),
                new ResolvedParameter<IRepository<StockQuote>>("validating")),
            new InjectionProperty(
                "Logger",
                new ResolvedParameter<ILogger>("composite")))
        .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>();

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

Note: 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.

Running the Application

Launch and use the application. Entries for the user interface (UI) 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 Labs\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 [Using BuildUp to Wire Up Objects Not Created by the Container](#).

To begin, open the StocksTicker.sln file located in the Labs\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 Figure 6.

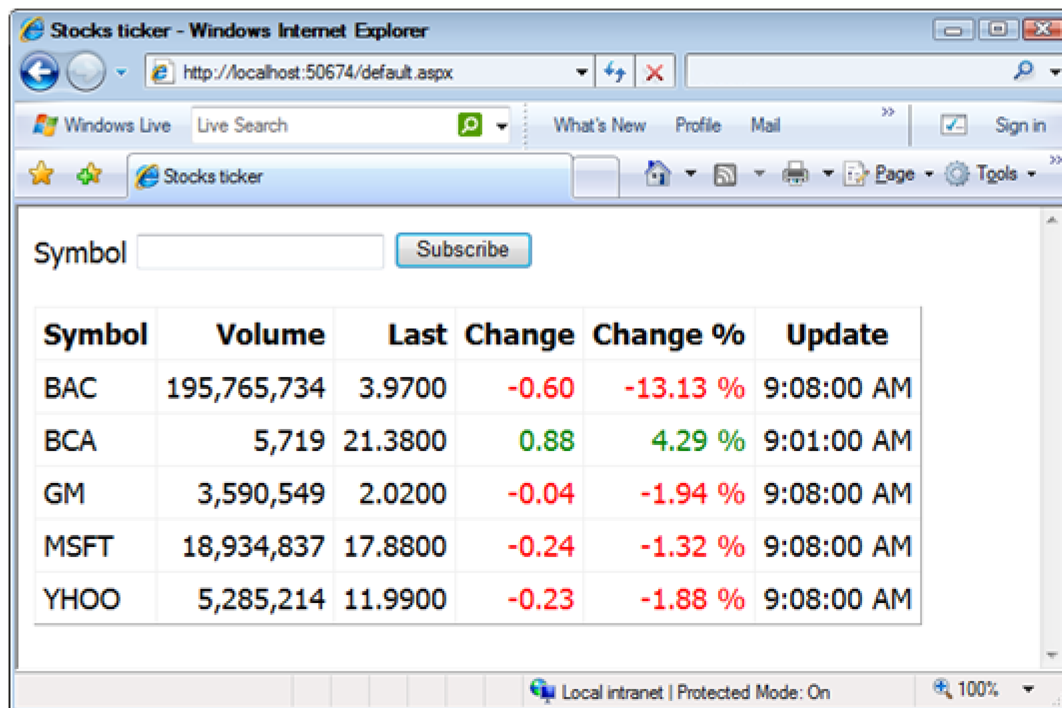


Figure 6

Stocks Ticker application

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, Microsoft.Practices.Unity.Configuration and Microsoft.Practices.ObjectBuilder2 assemblies from the Unity bin folder.
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 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 code.

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

> **Note:** 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**, as shown in Figure 7.

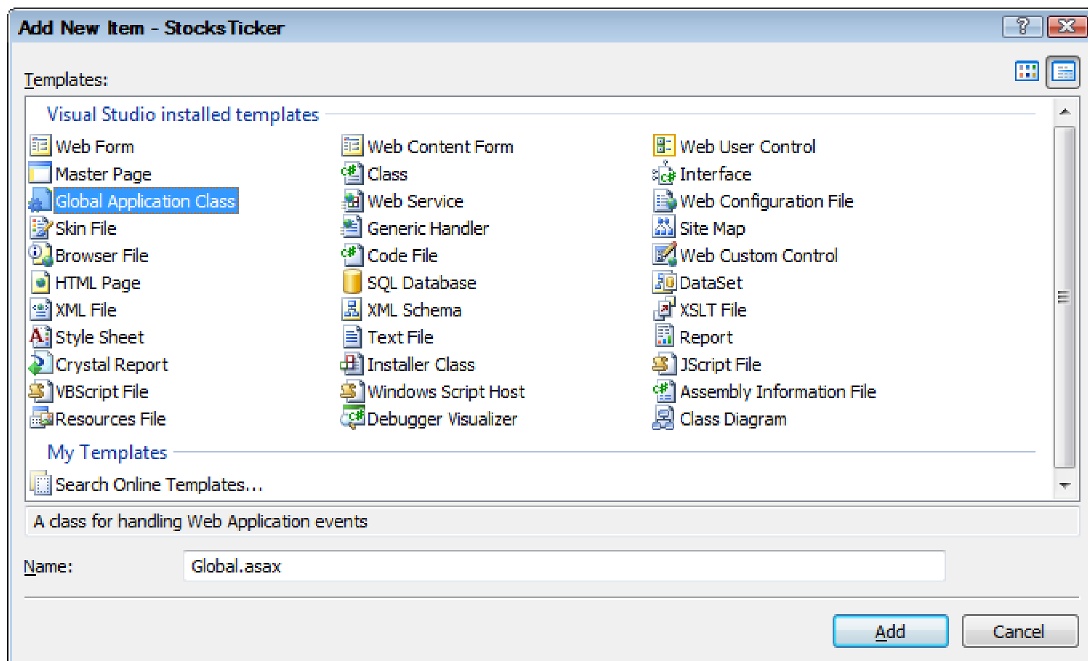


Figure 7

Adding a new application class

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:

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

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

```
    }
```

4. Add 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;
    }
}
```

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

5. Add the following method to set up a container using settings from the configuration file.

```
private static void ConfigureContainer(
    IUnityContainer container,
    string containerName)
{
    UnityConfigurationSection section
        = WebConfigurationManager.GetWebApplicationSection("unity")
            as UnityConfigurationSection;

    if (section != null)
    {
        UnityContainerElement containerElement
            = section.Containers[containerName];
        if (containerElement != null)
        {
            containerElement.Configure(container);
        }
    }
}
```

6. 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.

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

    ApplicationContainer = applicationContainer;
```

```
}
```

7. Update the empty **Application_End** method to dispose the application's container.

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

    if (applicationContainer != null)
    {
        applicationContainer.Dispose();

        this.ApplicationContainer = null;
    }
}
```

8. Add 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=1.2.0.0, Culture=neutral, 
PublicKeyToken=31bf3856ad364e35"/>
</configSections>
```

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

```
  </system.diagnostics>
  <unity>
  </unity>
</configuration>
```

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

```
<unity>
  <typeAliases>
    <typeAlias
      alias="string"
      type="System.String, mscorlib" />
    <typeAlias
      alias="TraceSource"
      type="System.Diagnostics.TraceSource, System, Version=2.0.0.0, 
Culture=neutral, PublicKeyToken=b77a5c561934e089" />

    <typeAlias
      alias="singleton"
      type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager, 
Microsoft.Practices.Unity, Version=1.2.0.0, Culture=neutral, 
PublicKeyToken=31bf3856ad364e35" />

    <typeAlias
      alias="ILogger"
      type="StocksTicker.Loggers.ILogger, StocksTicker" />
    <typeAlias
      alias="TraceSourceLogger"
      type="StocksTicker.Loggers.TraceSourceLogger, StocksTicker" />

    <typeAlias
      alias="IStockQuoteService"
      type="StocksTicker.StockQuoteServices.IStockQuoteService, StocksTicker" 
/>
    <typeAlias
      alias="MoneyCentralStockQuoteService"
      type="StocksTicker.StockQuoteServices.MoneyCentralStockQuoteService, 
StocksTicker" />
  </typeAliases>
</unity>
```

5. Add elements for the **containers** collection and a **container** with the name **application**.

```
  </typeAliases>
```

```
  <containers>
    <container name="application">
    </container>
  </containers>
</unity>
```

6. Add the **types** element.

```
<container name="application">
  <types>
  </types>
</container>
```

7. Add a **type** 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">
  <types>
    <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
      <lifetime type="singleton"/>
      <typeConfig>
        <property name="Logger" propertyType="ILogger"/>
      </typeConfig>
    </type>
  </types>
</container>
```

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

```
<container name="application">
  <types>
    <type type="ILogger" mapTo="TraceSourceLogger">
      <lifetime type="singleton"/>
      <typeConfig>
        <constructor>
          <param name="sourceName" parameterType="string">
            <value value="default"/>
          </param>
        </constructor>
      </typeConfig>
    </type>
    <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
      <typeConfig>
        <property name="Logger" propertyType="ILogger"/>
      </typeConfig>
    </type>
  </types>
```

```
</container>
```

Running the Application

To run the application

1. Launch the application. Open two browser instances, as shown in Figure 8, and open the application's URL in them. Use the application for a while in both browsers, subscribing to different sets of symbols.

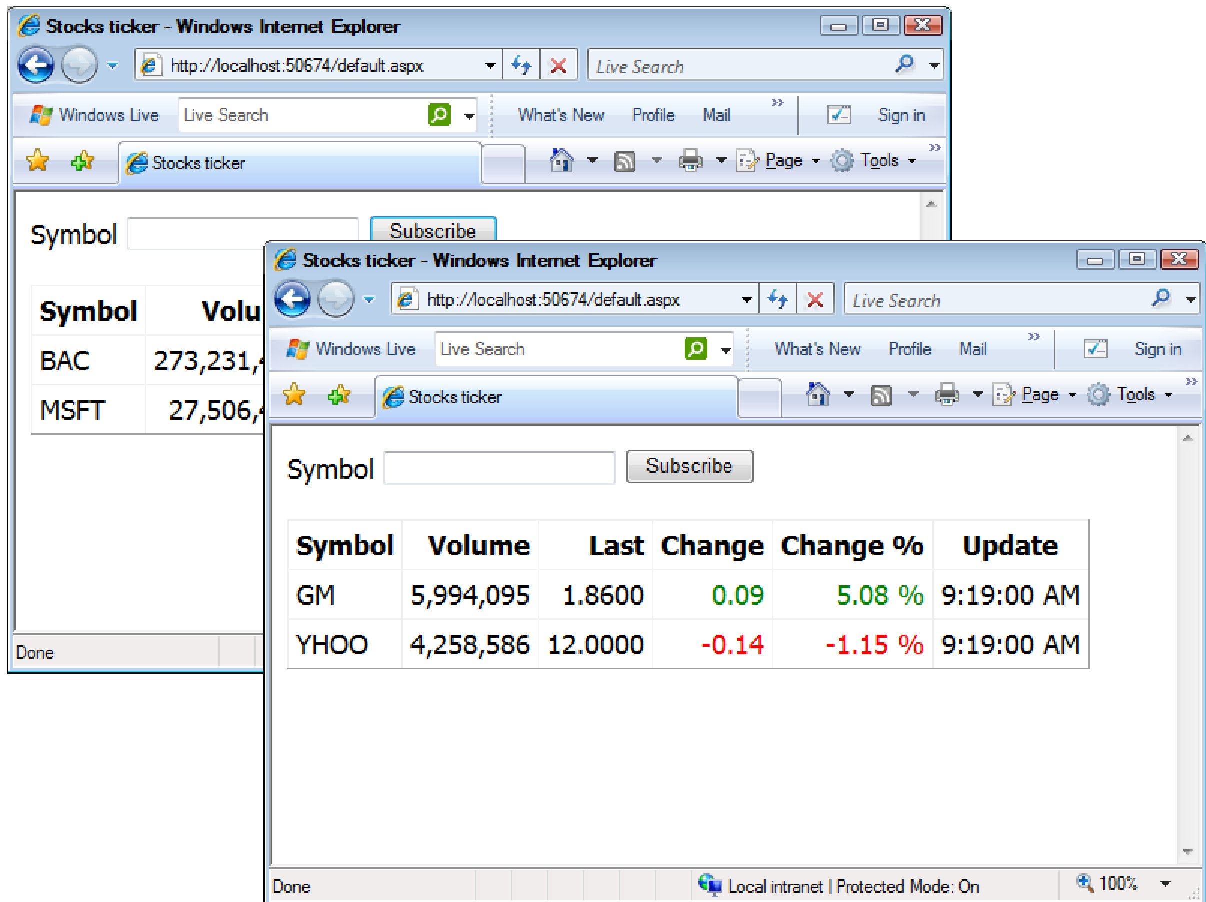
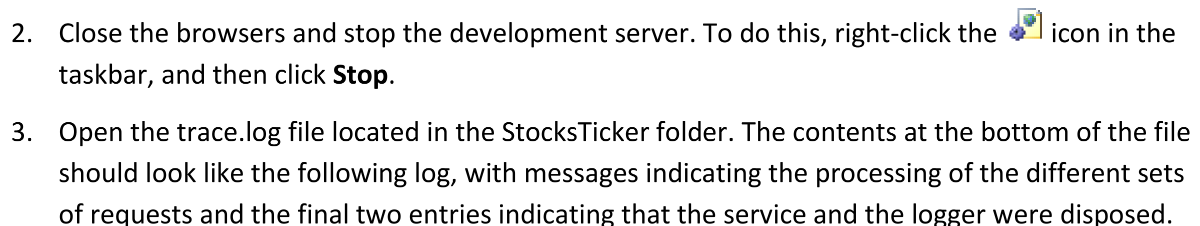


Figure 8
Multiple instances of the application

2. Close the browsers and stop the development server. To do this, right-click the  icon in the taskbar, and then 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 GM
    DateTime=2009-02-23T18:15:25.4761924Z
default Information: 0 : Retrieving quotes for BAC MSFT
```

```
    DateTime=2009-02-23T18:15:29.1101924Z
default Information: 0 : Received result for BAC MSFT
    DateTime=2009-02-23T18:15:29.5251924Z
default Information: 0 : Parsed result for BAC 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.

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 file.
2. Add a constant named **SessionContainerKey** with **"session container"** as its value:

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

3. Add 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 a **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();
        ConfigureContainer(sessionContainer, "session");

        this.SessionContainer = sessionContainer;
    }
}
```

5. Add a **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.

```
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** as a child of the **containers** element.

```
<containers>
  <container name="application">
    ...
  </container>

  <container name="session">
    <types>
    </types>
  </container>
</containers>
```

3. Add a **type** 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">
  <types>
    <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
      <lifetime type="singleton"/>
      <typeConfig>
        <property name="Logger" propertyType="ILogger"/>
      </typeConfig>
    </type>
  </types>
</container>
```

4. Remove the type element mapping the **IStockQuoteService** interface from the **types** collection for the **application** container.

```
<container name="application">
  <types>
    <type type="ILogger" mapTo="TraceSourceLogger">
      <lifetime type="singleton"/>
      <typeConfig>
        <constructor>
          <param name="sourceName" parameterType="string">
            <value value="default"/>
          </param>
        </constructor>
      </typeConfig>
    </type>
    <type type="IStockQuoteService" mapTo="MoneyCentralStockQuoteService">
      <typeConfig>
```

```
        <property name="Logger" propertyType="ILogger"/>
      </typeConfig>
    </type>
  </types>
</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 Labs\Lab05\end\StocksTicker folder.

Copyright

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, Windows Vista, MSDN, Visual C#, Visual Basic, and Visual Studio are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.