Service Tutorial 1 (C#) - Creating a Service

Writing an application using Decentralized Software Services (DSS) is a simple matter of orchestrating the input and output between a set of services. Services represent the interface to software or hardware and allow you to communicate between processes that perform specific functions.

This tutorial is provided in the C# language. You can find the project files for this tutorial at the following location under the installation folder:

Sample location
```
Samples\ServiceTutorials\Tutorial1\CSharp
```

This tutorial teaches you how to:

- Create a new service.
- Start a service.
- Support an HTTP request.
- Use Control Panel.
- Stop the Service.
- Support Replace.

See Also:

- Appendix A: The Code

Prerequisites

Hardware

This tutorial requires no special hardware.

Software

This tutorial is designed for use with Microsoft Visual C#. You can use:

- Microsoft Visual C# Express Edition
- Microsoft Visual Studio Standard, Professional, or Team Edition.

You will also need Microsoft Internet Explorer or another conventional web browser.

Step 1: Create a Service

Begin by creating a new service.

Open the **Start** menu and choose the **DSS Command Prompt** command. If the command does not appear in the **Start** menu, choose **All Programs**, **Microsoft Robotics Developer Studio**, and then **DSS Command Prompt**. This opens a special Command Prompt window in the root directory of the installation path.

Change to the **Samples** directory and run the **DssNewService** tool using the parameters shown in the sample below to create your first service. Then change to the **ServiceTutorial1** (**ServiceTutorial<Number One>**) directory. This procedure automatically creates a template to help you get started.

```
cd Samples
dssnewservice /namespace:Robotics /service:ServiceTutorial1
cd ServiceTutorial1
```

At this time, a Microsoft Visual Studio solution named **ServiceTutorial1.sln** is created in **ServiceTutorial1** directory. Load this solution using your C# editor.

```
start ServiceTutorial1.sln
```

Next, build the solution. In Visual Studio you can build the solution by clicking **Build** menu and then choosing **Build Solution** (or pressing **F6**). You can also compile from the **DSS Command Prompt**:

```
msbuild ServiceTutorial1.sln
```

Step 2: Start a Service

Go back to the Microsoft Robotics Developer Studio (RDS) installation folder.

```
cd ..\..
```

You should now see the following files in the **bin** directory that were created when you built your project.

```
.
.
.
ServiceTutorial1.Y2007.M07.dll
ServiceTutorial1.Y2007.M07.pdb
ServiceTutorial1.Y2007.M07.Proxy.dll
ServiceTutorial1.Y2007.M07.Proxy.pdb
ServiceTutorial1.Y2007.M07.proxy.xml
ServiceTutorial1.Y2007.M07.transform.dll
ServiceTutorial1.Y2007.M07.transform.pdb
```

> By default, **DssNewService** appends the current year and month to the name of the assembly of the generated service, such as **.Y2007.M07**. So depending on the current date, the build names for the **ServiceTutorial1** files that you just created may be different from the ones you see in the above sample. You may also see other **ServiceTutorial1** files with different dates that are coming from the **ServiceTutorial1** project installed with Robotics Developer Studio.

To run a service, you must first run a DSS node by running the DSS hosting application **DssHost.exe**. **DssHost** can start services for you. There are 3 ways to specify which service or services **DssHost** should start:

- By manifest, using the command line flag **/manifest**
- By assembly name, using the command line flag **/dll**
- By contract, using the command line flag **/contract**

A **manifest** file is an XML file that contains the information needed to start a service. **DssNewService** automatically creates a file called **ServiceTutorial1.manifest.xml** which contains the information required by **DssHost** to start the service.

Start **DssHost** using the manifest already created with the following command.

```
dsshost /port:50000 /manifest:"samples\ServiceTutorial1\ServiceTutorial1.manifest.xml"
```

> The manifest files for the installed tutorial projects are stored in the **samples\Config** folder. If you are using the **ServiceTutorial1** project in **samples\ServiceTutorials\ServiceTutorial1** folder you should run **DssHost** with the right manifest from **samples\Config**.
>
> **bin\dsshost /port:50000 /manifest:"samples\Config\ServiceTutorial1.manifest.xml"**
>
> However, the installed tutorial projects contain the completed code for each tutorial and thus running them at this point has the behavior of the service when all the steps in the tutorial are completed.

You should see service load outputs for the specified manifest:

```
.
.
.
*   Starting manifest load: .../ServiceTutorial1.manifest.xml
*   Service uri:  ...[http://localhost:50000/servicetutorial1]
*   Manifest load complete ...[http://localhost:50000/manifestloaderclient]
```

Now open your web browser and navigate to the address **http://localhost:50000/servicetutorial1**

An XML serialization (representation) of the newly created service, **ServiceTutorial1State** encapsulated in a SOAP envelope appears in the browser window.

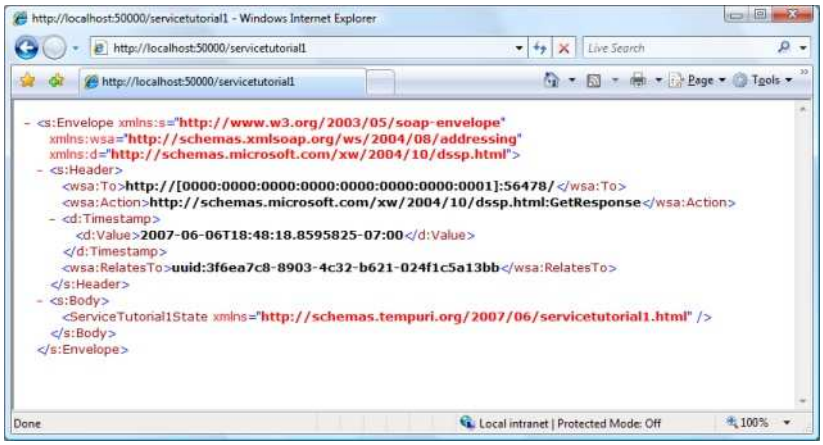

Figure 1 - http://localhost:50000/servicetutorial1 in browser shows the state of the service as a SOAP envelope.

Exit DSS host by pressing **CTRL+C** in the command window. "Shutdown complete" message appears and then returns to the command prompt.

Step 3: Support HTTP GET

When sending state to a web browser, the service can optionally avoid sending a SOAP envelope and instead send an XML serialization of its state as follows.

The **ServiceTutorial1Types.cs** file defines the *Contract* of the service. It includes types for contract identifier, state, operations port, operation messages, and request/response types for this service. You will learn more about the components of services as you go through these tutorials.

In the file **ServiceTutorial1Types.cs**, make the following changes:

1. First add a using directive to the top of the file to include the namespace **Microsoft.Dss.Core.DsspHttp**. This namespace contains the message definitions that allow a service to respond directly to requests from a standard HTTP client, such as a standard web browser.

   ```
   using Microsoft.Dss.Core.DsspHttp;
   ```

2. Next, add a property to the class **ServiceTutorial1State**, this will allow us to see the serialized data more clearly. In Service Tutorial 6 (C#) - Retrieving State and Displaying it Using an XML Transform you will use the **ServiceTutorial1State** class to carry information between services.

   ```
   private string _member = "This is my State!";

   [DataMember]
   public string Member
   {
       get { return _member; }
       set { _member = value; }
   }
   ```

   The **DataContract** attribute specifies that the **ServiceTutorial1State** class is XML serializable. Within a type marked with the **DataContract** attribute, you must explicitly mark individual properties and fields as XML serializable using the **DataMember** attribute. Only public properties and fields declared with this attribute will be serialized. Also, in order for the property members to serialize, they will need to have both the **set** and **get** methods implemented.

   > *Attributes* is a feature in .NET that allows adding keyword-like annotations to programming elements such as types, fields, methods, and properties. To learn more about .NET attributes see Attributes Overview in the .NET Framework Developer's Guide.

3. Now, further down in the same file, add the **HttpGet** message to the list of messages supported by the service's port. A port is a mechanism through which messages are communicated between services. A **PortSet** is just a collection of ports.

   ```
   [ServicePort]
   public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, HttpGet>
   {
   }
   ```

4. In the file **ServiceTutorial1.cs**, add support for the **HttpGet** message. **ServiceTutorial1.cs** defines the *behavior* of the service.

   Again add a using statement for **DsspHttp**.

   ```
   using Microsoft.Dss.Core.DsspHttp;
   ```

5. Then in the **ServiceTutorial1** class, add a message handler for the **HttpGet** message.

   ```
   /// <summary>
   /// Http Get Handler
   /// </summary>
   /// <remarks>This is a standard HttpGet handler. It is not required because
   ///  DsspServiceBase will handle it if the state is marked with [ServiceState].
   ///  It is included here for illustration only.</remarks>
   [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
   public IEnumerator<ITask> HttpGetHandler(HttpGet httpGet)
   {
       httpGet.ResponsePort.Post(new HttpResponseType(_state));
       yield break;
   }
   ```

   This handler sends an **HttpResponseType** message to the response port of the **HttpGet** message. The HTTP infrastructure within the DSS node will serialize the provided state to XML and set that as the body of response to the HTTP request.

   IMPORTANT NOTE: This is the default behavior of a **HttpGet** handler. The resulting output will be an XML serialized version of the service state. If this is the behavior that you want, then there is no need to write the **HttpGet** code into your service because **DsspServiceBase** will take care of it for you if you tag your state with the **ServiceState** attribute.

   > The **HttpResponseType** constructor has other overloads than the one used here. One of these overloads allows the service author to specify the path of an XSLT file that can be used by a web client to transform the serialized XML. (See Service Tutorial 6 (C#) - Retrieving State and Displaying it Using an XML Transform). An alternative, which is especially useful because you do not have to declare a **HttpGet** handler, is to specify the transform using the attribute **[ServiceState (StateTransform="...")]** on the service state class.

6. Build and run the service (press **F5** or select the **Debug** > **Start Debugging** menu command) and, while it is running, use a web browser to navigate to **http://localhost:50000/servicetutorial1** to view the following:

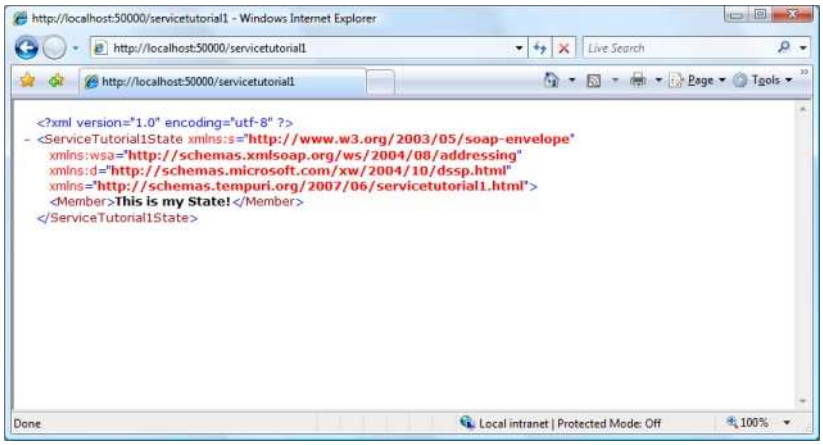

Figure 2 - State of the service in browser shows **ServiceTutorial1State** and its members.

Step 4: Using Control Panel

To start your service you could also use DSS Control Panel, which is itself a DSS service. To try this, first terminate the current DSS node by pressing CTRL+C in the command prompt. Then run **DssHost** again without supplying a manifest.

```
dsshost /port:50000
```

Now open http://localhost:50000 in the browser. When the page is loaded, in the left navigation pane click on **Control Panel**. A table of services recognized by the current node appears in your browser.

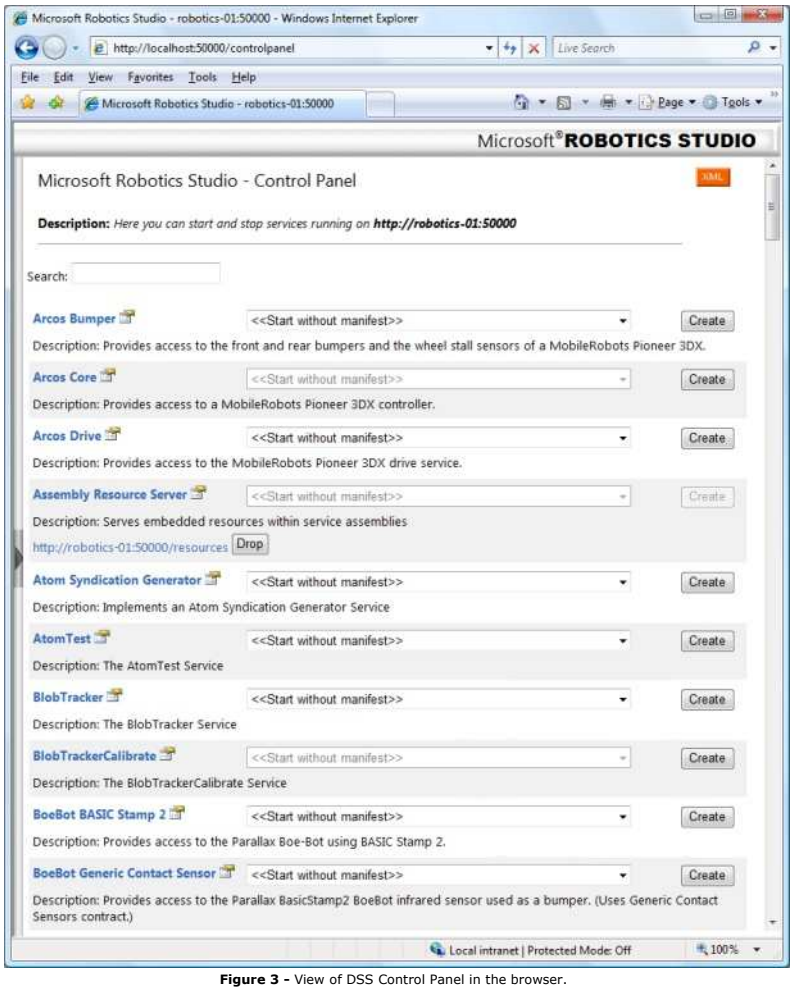

Figure 3 - View of DSS Control Panel in the browser.

Each row of the table begins with the **Name** of a service. Following that there is a dropdown list with the list of manifests that can run the service. If there are any instances of that service which are currently running, you should see a URL for that instance under the service's **Description**. Each instance of a running service also includes a button in the right-hand side of the URL which allows you to **Drop** that service. Clicking this button sends a **Drop** message to the service. This message stops the service.

Scroll down the page and find the entry for **ServiceTutorial1** or type the name of your service in the Search box: **servicetutorial1**

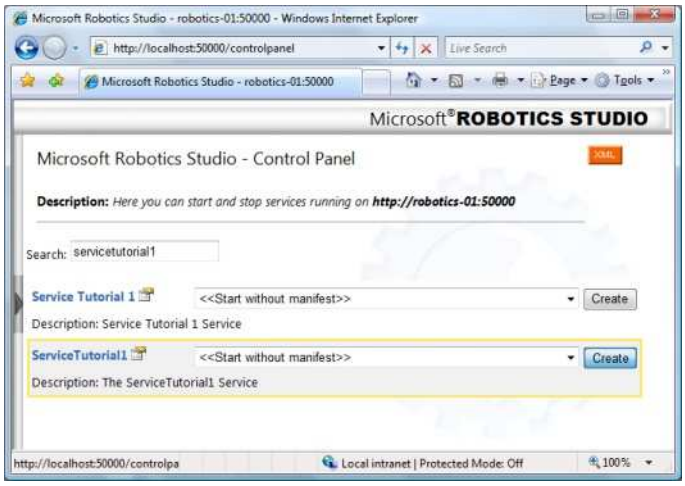

Figure 4 - Entry for **ServiceTutorial1** in Control Panel

You will probably see two results. One of them is the **Service Tutorial 1** from the completed project that comes with a DSS installation. You can distinguish them by looking at the location of manifests in the dropdown lists, however, for running this service you don't need to select a manifest and can do it by directly loading the assembly (dll file) by selecting **<<Start without manifest>>**. In cases where you need to run your service together with a group of partner services that are listed in the manifest, you will need to run the manifest instead.

Run the service by clicking on the **Create** button.

Now from the left navigation pane select **Service Directory**. You should see **/servicetutorial1** in the list of services that are running. Notice the other services running, including **/controlpanel** are actually different components of the DSS runtime and are started by default when DSS environment is initialized. You can inspect the state of each service by clicking on its link or browsing directly to the service's URL.

> The control panel service is always started when a node starts. You can use this service to start and stop your service without having to restart the node. However, because the node loads the service assembly you will need to stop the node when you rebuild your service.

Step 5: Stop the Service

While the service is running, open your web browser to **http://localhost:50000/controlpanel**

> You may need to refresh the Control Panel service to get an up-to-date representation of the services running on the node.

Find the entry for **servicetutorial1** as described in the previous section.

Figure 5 - Instance of **ServiceTutorial1** service in Control Panel with a Drop button

Click the URL to inspect the state of the service, create a new instance by clicking the **Create** button, or stop a running iteration of the service by clicking the **Drop** button.

Step 6: Support Replace

A Replace message is used to replace the current state of a service. When a Replace message is sent the entire state of the service is replaced with the state object specified in the body of the Replace message. This allows initializing a service with a new state or restoring the service with a state that was previously saved, at any time during the lifetime of the service.

For our service to support replace, define the **Replace** type in **ServiceTutorial1Types.cs**.

```
/// <summary>
/// ServiceTutorial1 Replace Operation
/// </summary>
/// <remarks>The Replace class is specific to a service because it uses
/// the service state.</remarks>
public class Replace : Replace<ServiceTutorial1State, PortSet<DefaultReplaceResponseType, Fault>>
{
}
```

Then add **Replace** to the port set.

```
/// <summary>
/// ServiceTutorial1 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, HttpGet, Replace>
{
}
```

In **ServiceTutorial1.cs** add the Replace handler.

```
/// <summary>
/// Replace Handler
/// </summary>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> ReplaceHandler(Replace replace)
{
    _state = replace.Body;
    replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
    yield break;
}
```

In the above code the *body* of the replace message which is of type **ServiceTutorial1State** is assigned to **_state** of this service. Then, a success response of type **DefaultReplaceResponseType** is posted to the **ResponsePort** of the Replace message. This signals back to the sender that the state was successfully replaced.

We will use the **Replace** operation later to exchange values between services.

Summary

In this tutorial, you learned how to:

- Create a new service.
- Start a service.
- Support an HTTP request.
- Use Control Panel.
- Stop the Service.
- Support Replace.

Appendix A: The Code

ServiceTutorial1Types.cs

The **ServiceTutuorial1Types.cs** file defines the service Contract. The Contract is identified by a unique text string, which is associated with a .NET CLR namespace, and a set of messages that the service supports.

This establishes the namespaces that are used in this file.

```
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.Core.DsspHttp;
using Microsoft.Dss.ServiceModel.Dssp;

using System;
using System.Collections.Generic;
using W3C.Soap;

using servicetutorial1 = RoboticsServiceTutorial1;
```

The Contract Class

```
/// <summary>
/// ServiceTutorial1 Contract class
/// </summary>
public sealed class Contract
{
    /// <summary>
    /// The Dss Service contract
    /// </summary>
    public const String Identifier = "http://schemas.tempuri.org/2006/06/servicetutorial1.html";
}
```

This **Contract** class defines the unique string, **Identifier**, that is used to identify this Contract and, in general, service. We follow a convention used in XML documents of using a URI (Unique Resource Identifier) to specify a unique name. The default mechanism used is to compose the URI from a host name (provided as a parameter to **DssNewService**), a path (in this example empty), the year, the month and the name of the service. If the **Namespace** is composed from a host name and path that the service author has some level of control over (for example, the address of an account at **http://spaces.live.com** could be used) then the composition of date elements and service name gives the user a reasonable expectation of uniqueness, with the benefit of containing a small amount of information about the service. While there is no requirement that the URI have a page behind it, there is also no reason why a service author shouldn't create a matching page.

If an opaque unique identifier is required it is possible, but not recommended, to use a GUID (generated using the tool **GuidGen.exe**) in the form "**urn:uuid:4de060f3-f665-11da-95e7-00e08161165f**"

The ServiceTutorial1State Class

```
/// <summary>
/// The ServiceTutorial1 State
/// </summary>
[DataContract]
public class ServiceTutorial1State
{
    private string _member = "This is my State!";

    [DataMember]
    public string Member
    {
        get { return _member; }
        set { _member = value; }
    }
}
```

The **ServiceTutorial1State** class has one public property, **Member**, that is itself declared with the **DataMember** attribute that (as discussed above) is used to indicate that this member should be serialized. By default, a property or field that is **null** will not be serialized.

The ServiceTutorial1Operations Class

```
/// <summary>
/// ServiceTutorial1 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, HttpGet, Replace>
{
}
```

This class defines the public messages that the service supports.

Message	Description
DsspDefaultLookup	Every service **must** support a message derived from **Lookup<TBody,TResponse>**. The infrastructure defines a **DsspDefaultLookup**. The **DsspServiceBase** class (from which service implementation classes derive) defines a default message handler for **Lookup**. So in practice, adding **DsspDefaultLookup** to the operations **PortSet** is all that a service author needs to do. A service responds to the **Lookup** message with basic information about itself. The default implementation returns the **ServiceInfo** property from the **DsspServiceBase**.
DsspDefaultDrop	The **Drop** message, when sent to a service, stops that service. A service doesn't have to implement this. A default implementation is provided in **DsspServiceBase**. In general, a service can add **DsspDefaultDrop** to the operations **PortSet** to support **Drop**.
Get	A service should respond to the **Get** message with its current state. The service should not modify its state in response to the **Get** message. Although the **Get** message is optional, nearly all services implement it. Even though most services have a message type called **Get**, every service has a different state type. This means that the **Get** messages are different for every service because the response must include a copy of the service-specific state. Hence the requirement to create a new **Get** class in every service.
HttpGet	As discussed above, this is equivalent to the **Get** operation but allows the service to communicate more directly with a Web browser client. The **HttpGet** message defined in the DSS Core can be used and there is no need to declare a service-specific message type.
Replace	A service should replace its entire state with the body of a replace message. The replace message is optional and not every service needs nor should implement this message.

Get and **Replace** are the two messages that do *not* have appropriate default declarations but are supported by this service.

- In the case of **Get**, this is because the primary response to a **Get** message should be the **State** of the service. The state type of this service, **ServiceTutorial1State**, is specific to this service.
- For **Replace**, the **Body** of the message is the **ServiceTutorial1State** of this service.

```
/// <summary>
/// ServiceTutorial1 Get Operation
/// </summary>
/// <remarks>All services require their own specific Get class because
/// the service state is different for every service.</remarks>
public class Get : Get<GetRequestType, PortSet<ServiceTutorial1State, Fault>>
{
    /// <summary>
    /// ServiceTutorial1 Get Operation
    /// </summary>
    public Get()
    {
    }

    /// <summary>
    /// ServiceTutorial1 Get Operation
    /// </summary>
    public Get(Microsoft.Dss.ServiceModel.Dssp.GetRequestType body) :
            base(body)
    {
    }

    /// <summary>
    /// ServiceTutorial1 Get Operation
    /// </summary>
    public Get(Microsoft.Dss.ServiceModel.Dssp.GetRequestType body, Microsoft.Ccr.Core.PortSet<ServiceTutorial1State,W3C.Soap.Fault> responsePort) :
            base(body, responsePort)
    {
    }
}

/// <summary>
/// ServiceTutorial1 Replace Operation
```

```
/// </summary>
/// <remarks>The Replace class is specific to a service because it uses
/// the service state.</remarks>
public class Replace : Replace<ServiceTutorial1State, PortSet<DefaultReplaceResponseType, Fault>>
{
}
```

ServiceTutorial1.cs

The **ServiceTutorial1.cs** file contains the service implementation class.

```
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.Core.DsspHttp;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Xml;
```

These are the namespaces that will be used in this class.

Service Implementation Class

This section of the code declares a class, **ServiceTutorial1**, derived from **DsspServiceBase**. All service implementation classes derive from **DsspServiceBase**.

```
/// <summary>
/// Implementation class for ServiceTutorial1
/// </summary>
[DisplayName("Service Tutorial 1: Creating a Service")]
[Description("Demonstrates how to write a basic service.")]
[Contract(Contract.Identifier)]
[DssServiceDescription("http://msdn.microsoft.com/library/bb483064.aspx")]
public class ServiceTutorial1Service : DsspServiceBase
{
```

The **Contract** attribute declares a direct association between this class and the **Contract Identifier** discussed in the previous section. The **DisplayName** and **Description** are attributes that describe the service. The **DisplayName** is the shorter description similar to a title. The **Description** is a more detailed description.

ServiceTutorial1State holds the state of the service. Every service provides messages through its operations port that allow other services to read and modify its state.

```
/// <summary>
/// Service State
/// </summary>
[ServiceState]
private ServiceTutorial1State _state = new ServiceTutorial1State();
```

In this sample, the service only supports:

- Reading the entire state (**Get** and **HttpGet**), and
- Replacing the entire state (**Replace**)

The **ServicePort** attribute declares that the **_mainPort** field is the primary service port of this service.

```
/// <summary>
/// Main Port
/// </summary>
[ServicePort("/servicetutorial1", AllowMultipleInstances=false)]
private ServiceTutorial1Operations _mainPort = new ServiceTutorial1Operations();
```

This also specifies the default path used to address this service, **/servicetutorial1**. It also stipulates that only one instance of this service to run at a time. If **AllowMultipleInstances = true** is specified, a unique suffix is appended to the path when an instance of the service is created.

Initialization

When services are created, there is a two stage creation process during which partner services are created as discussed in Service Tutorial 4 (C#) - Supporting Subscriptions and Service Tutorial 5 (C#) - Subscribing.

```
/// <summary>
/// Default Service Constructor
/// </summary>
public ServiceTutorial1Service(DsspServiceCreationPort creationPort) :
        base(creationPort)
{
}
```

This constructor is used in the first part of the creation process and **must** have the form shown for the service to be created correctly.

The **Start** method is called as the last action of the two stage creation process.

```
/// <summary>
/// Service Start
/// </summary>
protected override void Start()
{
    base.Start();
    // Add service specific initialization here.
}
```

Calling **base.Start()** does three things for the service. (These steps could be done explicitly, but using **base.Start()** is easier).

1. Calls **ActivateDsspOperationHandlers** which causes **DsspServiceBase** to attach handlers to each message supported on the main service port. (This relies on **ServiceHandler** attributes and method signatures.)

   See below for how to declare a handler.

2. Calls **DirectoryInsert** to insert the service record for this service into the directory. The directory is itself a service and this method sends an **Insert** message to that service.

   When the **DssHost** is running, you can inspect the directory service by pointing your browser to http://localhost:50000/directory.

3. Calls **LogInfo** to send an **Insert** message to the **/console/output** service (**http://localhost:50000/console/output**). The category of the message is **Console**, which causes the message to be printed to the command console. The URI of the service is automatically appended to the output.

The following code summarizes the three tasks done by **base.Start()** as described above. However, in most cases it is recommended to use **base.Start()** instead of manually initializing the service start.

```
// Listen on the main port for requests and call the appropriate handler.
ActivateDsspOperationHandlers();

// Publish the service to the local node Directory
DirectoryInsert();

// display HTTP service Uri
LogInfo(LogGroups.Console, "Service uri: ");
```

Message Handlers

The following is the handler for the **Get** message. This handler simply posts the state of the service to the response port of the message.

```
/// <summary>
/// Get Handler
/// </summary>
/// <param name="get"></param>
/// <returns></returns>
/// <remarks>This is a standard Get handler. It is not required because
///  DsspServiceBase will handle it if the state is marked with [ServiceState].
///  It is included here for illustration only.</remarks>
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
```

```
public virtual IEnumerator<ITask> GetHandler(Get get)
{
    get.ResponsePort.Post(_state);
    yield break;
}
```

The **ServiceHandler** attribute is used by the method **ActivateDsspOperationHandlers** called in the **base.Start()** method to identify member functions as message handlers for messages posted to the main service port, itself identified with the **ServicePort** attribute. **ServiceHandlerBehavior.Concurrent** is used to specify that the message handler only needs Read-Only access to the state of the service. This allows message handlers that do not modify state to run concurrently.

> ⚠ A message handler that needs **write access** to the state of the service should use **ServiceHandlerBehavior.Exclusive**. This makes sure only one handler at a time can modify the state.

Message handlers generally have the signature, **Microsoft.Ccr.Core.IteratorHandler<T>**

```
public delegate IEnumerator<ITask> IteratorHandler<T>(T parameter);
```

Using *Iterators* (a new feature in .NET 2.0) allows the handler to contain a sequence of asynchronous actions without blocking a thread. This is demonstrated in Service Tutorial 3 (C#) - Persisting State.

The **HttpGetHandler** is discussed previously in this tutorial (Step 3: Supporting HTTP GET).

```
/// <summary>
/// Http Get Handler
/// </summary>
/// <remarks>This is a standard HttpGet handler. It is not required because
///  DsspServiceBase will handle it if the state is marked with [ServiceState].
///  It is included here for illustration only.</remarks>
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> HttpGetHandler(HttpGet httpGet)
{
    httpGet.ResponsePort.Post(new HttpResponseType(_state));
    yield break;
}
```

Note that this is the default behavior for a **HttpGet** handler. Therefore it is not necessary to include this code in your service. It is included here so that you can see what it does. However, the DsspServiceBase class will take care of it for you and it is only necessary to declare your own handler if you want to do some service-specific processing before sending the result.

The following is the handler for the **Replace** message. Note that this handler is declared with **ServiceHandlerBehavior.Exclusive**, indicating that it will modify state. Only one **Exclusive** handler will execute at a time and no **Concurrent** handler can execute while an **Exclusive** handler is running.

```
/// <summary>
/// Replace Handler
/// </summary>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> ReplaceHandler(Replace replace)
{
    _state = replace.Body;
    replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
    yield break;
}
```

See Also

.NET Framework Developer's Guide: Attributes Overview
