Microsoft Robotics Developer Studio 2008 Express Edition *Send feedback on this topic*

Service Tutorial 2 (C#) - Updating State

Writing an application using the DSS is a simple matter of orchestrating 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\Tutorial2\CSharp
```

This tutorial teaches you how to:

- Add a New Message to a Service.
- Implement Message Handler.
- Post a Message Internally, Using Basic CCR Activation and Timers.
- See the Service in Action.

The service written in Service Tutorial 1 will be modified to become a simple monotonic clock, incrementing an internal counter approximately once per second.

Prerequisites

This tutorial uses the service written in Service Tutorial 1 (C#) - Creating a Service. The service you will create is referred to as **ServiceTutorial2**. Feel free to start a new project or keep working in the previous project, keeping in mind that some names may differ.

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: Add a New Message to a Service

The service needs to store a counter in its state and a message needs to be defined that will cause the counter to be incremented. All these changes happen in **ServiceTutorial2Types.cs**.

The first change you will make is to add an integer property, **Ticks**, to the service state type.

```
private int _ticks;

[DataMember]
public int Ticks
{
    get { return _ticks; }
    set { _ticks = value; }
}
```

Next, you need to define a message to increment the **Tick** property.

The code snippet below defines the **IncrementTick** message. A message is derived from **DsspOperation** and has three elements: an **action**, a **body**, and a **response port**. The **IncrementTick** message declared below is derived from the **Update<TBody, TResponse>** generic class which sets the action verb to **Update** (specifically, **UpdateRequest**). The body type is declared as **IncrementTickRequest**. The response port is defined as **PortSet<DefaultUpdateResponseType, Fault>**.

The **Update** verb is used for a message that will modify a subset of the service state, in this case it will only affect the **Tick** property.

The body is used for any information that the service requires to perform this message action. In this case, the body doesn't need to contain fields because the **IncrementTick** message causes the **Tick** property in the state to increment by one. A constructor is defined for the **IncrementTick** message that creates and sets the body.

The response port allows the service to communicate success or failure to the caller. In this message, in common with most **Update** messages, the **DefaultUpdateResponseType** is sufficient to signal that the update was successful. While the **IncrementTick** message does not have a failure path, the **Fault** type is used by the DSS infrastructure to tell a caller when there is a problem sending the message to the service.

Add the following message and request types in **ServiceTutorial2Types.cs**.

```
public class IncrementTick : Update<IncrementTickRequest, PortSet<DefaultUpdateResponseType, Fault>>
{
    public IncrementTick()
        : base(new IncrementTickRequest())
    {
    }
}

[DataContract]
public class IncrementTickRequest
{
}
```

Now add the **IncrementTick** message to the list of messages that the service supports. If an **IncrementTick** message is sent to this service right now, the message would sit on the port until the service was shutdown, causing a memory leak. This is because we haven't yet defined a message handler.

```
/// <summary>
/// ServiceTutorial2 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial2Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, HttpGet, Replace, IncrementTick>
{
}
```

Step 2: Implement Message Handler

The next step is to implement the handler for the **IncrementTick** message. In **ServiceTutorial2.cs**, add a handler to the **ServiceTutorial2** class.

This handler increments the **Tick** property of the service state, logs the change to that property, and sends a success response to the caller.

As described in Service Tutorial 1 (C#) - Creating a Service, the **ServiceHandler** attribute declares that this method handles a message type sent to the main service port. In this case, it needs exclusive access to the state because this handler modifies that service state. Therefore, the handler is declared with **ServiceHandlerBehavior.Exclusive**.

```
/// <summary>
/// Increment Tick Handler
/// </summary>
/// <param name="incrementTick">Not used</param>
/// <returns></returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> IncrementTickHandler(IncrementTick incrementTick)
{
    // Only update the state here because this is an Exclusive handler
    _state.Ticks++;
    LogInfo("Tick: " + _state.Ticks);
    incrementTick.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}
```

The service now has a new field in the service state, a message that tells the service to increment that field, and a handler that actually changes the service state. The next step is to cause the state to change.

Step 3: Post a Message Internally

The service increments the **Tick** property approximately once per second. To do this, it needs some kind of timer. The .NET framework provides a timer mechanism that could be used. However, the CCR provides a mechanism that allows you to use timers directly within the CCR system of ports and receivers.

The first thing you have to do is declare a port to which a message is sent each time the timer fires. The following line declares a port to which a **DateTime** can be posted. Add this member field in the service class after the **_mainPort** declaration.

```
private Port<DateTime> _timerPort = new Port<DateTime>();
```

Add two lines to the **Start** method:

- Post a **DateTime** to the port you have just declared. The value of **DateTime** posted is unimportant to the execution of the service although it can be useful for debugging.
- Activate a handler for the port **_timerPort**.

Because the **_timerPort** field is not part of the main service port, you cannot activate a handler by adding a **ServiceBehavior** declaration to the hander method. The **Activate** method takes a list of tasks to activate. The task that is declared here, **Arbiter.Receive(...)**, is a simple receiver on a port. In this case, specifying that the **TimerHandler** method will be called when a message arrives on the port **_timerPort**. The first (boolean **true**) parameter indicates that the receiver is persistent, and so all messages to the port are received by the handler.

CCR manages all tasks passed to a call to **Activate()**. CCR is a concurrent task dispatch infrastructure on which DSS is built.

The **Start()** method should look like this:

```
/// <summary>
/// Service Start
/// </summary>
protected override void Start()
{
    base.Start();

    // Kick off the timer (with no delay) and start a receiver for it
    _timerPort.Post(DateTime.Now);
    Activate(Arbiter.Receive(true, _timerPort, TimerHandler));
}
```

The **TimerHandler** is called when a message arrives on **_timerPort**. It does two things:

- It posts an **IncrementTick** message to the main service port. This causes the execution of **IncrementTickHandler**.
- It activates on a 1000 millisecond timeout interval. Since we have not declared a method as the handler for this receiver, an anonymous delegate (a new feature in .NET 2.0) is used. This allows us to write the handler inline in the call to **Arbiter.Receive**. In that anonymous delegate, we post the **DateTime** value to the port, **_timerPort**. Note that this receiver is not persistent--the first parameter is **false**. This is because the port created by the **TimeoutPort()** method receives one message after the specified interval (in this case 1000 milliseconds) expires.

When the **Activate()**method is called, it adds the task defined by calling **Arbiter.Receive** to the list of active tasks for this service. The operations defined within the **Activate** call will be scheduled independently of the current thread. In this case the **TimerHandler** method will return immediately after the call to **Activate**; it does not wait until the timer is fired.

The **Arbiter.Receive()** method defines a one-time receiver that will take the message sent when the timer interval specified in the call to **TimeoutPort** expires (in this case after 1000 milliseconds) and passes it as the parameter to the anonymous delegate specified as the third parameter to **Arbiter.Receive()**.

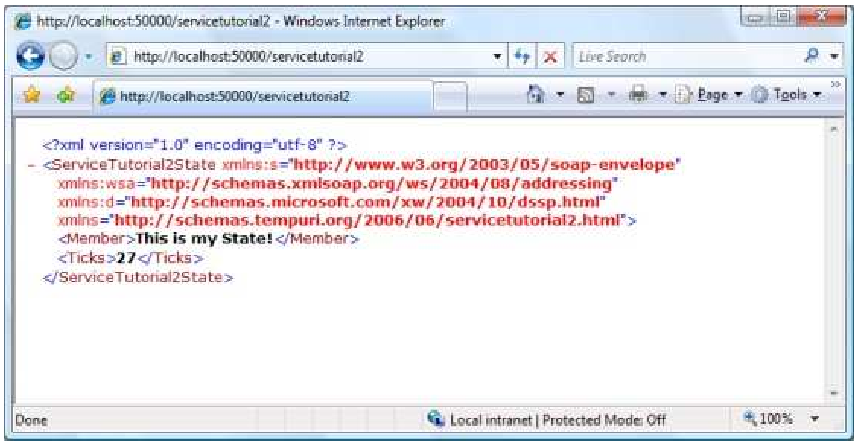
```
/// <summary>
/// Timer Handler
/// </summary>
/// <param name="signal">Not used</param>
void TimerHandler(DateTime signal)
{
    // Post a message to ourselves.
    // Do not modify the state here because this handler is
    // not part of the main interleave and therefore does not
    // run exclusively.
    _mainPort.Post(new IncrementTick());

    // Set the timer for the next tick
    Activate(
        Arbiter.Receive(false, TimeoutPort(1000),
            delegate(DateTime time)
            {
                _timerPort.Post(time);
            }
        )
    );
}
```

Step 4: See the Service in Action

If you look carefully at **ServiceTutorial2.cs** you will see that there are no handlers for the **Get** or **HttpGet** operations. However, you are going to use a web browser to examine the state. The reason that this works is because the state is tagged with the **ServiceState** attribute and **DsspServiceBase** uses default handlers when there are none declared in the service itself.

```
/// <summary>
/// Service State
/// </summary>
[ServiceState]
private ServiceTutorial2State _state = new ServiceTutorial2State();
```
Build and run the service in the usual way.

Now if you navigate to http://localhost:50000/servicetutorial2 , you should see the service state looking something like:


http://localhost:50000/servicetutorial2 - Windows Internet Explorer
http://localhost:50000/servicetutorial2
Live Search
http://localhost:50000/servicetutorial2
Page
Tools
Done
Local intranet | Protected Mode: Off
100%

```
  <?xml version="1.0" encoding="utf-8" ?>
- <ServiceTutorial2State xmlns:s="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns="http://schemas.tempuri.org/2006/06/servicetutorial2.html">
    <Member>This is my State!</Member>
    <Ticks>27</Ticks>
  </ServiceTutorial2State>
```

Figure 1 - Viewing the state of ServiceTutorial2 in browser.

Note the **Ticks** property. If you refresh the browser you will see this number increase over time.

Now open a browser window to http://localhost:50000/console/output. The logging messages sent by the service should appear in your browser:

Row	Time	Level	Description (mouse-over for details)
0	14:50:55	**	Validating Contract Directory Cache
1	14:50:57	**	Contract Directory is initialized
2	14:50:57	**	Attempting to load manifest: file:///C:/MSRS/samples/config/ServiceTutorial2.manifest.xml
3	14:50:58	**	Service uri: http://localhost:50000/servicetutorial2
4	14:50:57	**	Initial manifest loaded successfully.
5	14:50:59	**	Tick: 1
6	14:50:59	**	Tick: 2
7	14:51:00	**	Tick: 3
8	14:51:01	**	Tick: 4
9	14:51:02	**	Tick: 5
10	14:51:03	**	Tick: 6
11	14:51:04	**	Tick: 7
12	14:51:05	**	Tick: 8
13	14:51:06	**	Tick: 9
14	14:51:07	**	Tick: 10
15	14:51:08	**	Tick: 11
16	14:51:09	**	Tick: 12
17	14:51:10	**	Tick: 13
18	14:51:11	**	Tick: 14
19	14:51:12	**	Tick: 15
20	14:51:13	**	Tick: 16

Clicking on each row will open details about that message:

16	14:51:09	**	**Category**	StdOut
			Level	Info
			Time	2006-06-08T14:51:09.8709299-07:00
			Subject	Tick: 12
			Source	http://localhost:50000/servicetutorial2
			CodeSite	Boolean MoveNext()() at line:92, fileC:\MSRS \Samples\ServiceTutorials\ServiceTutorial2 \ServiceTutorial2.cs

The **DsspServiceBase** class defines a variety of logging methods using various overloads of the methods allows you to change the **Category**, **Level** and **Subject** fields of the Log. These methods include: **LogInfo**, **LogWarning** and **LogError**. The **Time**, **Source** and **CodeSite** entries are automatically populated.

Summary

In this tutorial, you took the service that you wrote in Service Tutorial 1 and turned it into a simple clock that updates a state member approximately once per second.

To do this, you completed the following:

- Add a New Message to a Service.
- Implement Message Handler.
- Post a Message Internally, Using Basic CCR Activation and Timers.

- See the Service in Action.