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

Service Tutorial 3 (C#) - Persisting State

Writing an application using the Decentralized Software Services (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\Tutorial3\CSharp`

This tutorial teaches you how to:

- Read an Initial State from a File.
- Persist a State.
- Manage Asynchronous Responses.

The service written in Service Tutorial 2 will be modified to persist its state.

Prerequisites

This tutorial uses the service written in Service Tutorial 2 (C#) - Updating State. The service you will create is referred to as **ServiceTutorial3**. 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: Read an Initial State from a File

To read the initial state of the service from a file, you need to add a declaration to the state member of the service implementation class and handle the case where there is no initial state file.

In the file, **ServiceTutorial3.cs**, make the following changes:

1. Add the **InitialStatePartner** attribute to the declaration of the service state. This causes the DSS constructor to set the initial value of the state. The **InitialStatePartner** is a special use of a service partner (partners are described in more detail in Service Tutorial 4 (C#) - Supporting Subscriptions and Service Tutorial 6 (C#) - Retrieving State and Displaying it Using an XML Transform).
2. The **ServiceUri** parameter specifies a location for the initial state document. This is expressed as a Uniform Resource Identifier (URI) relative to **/mountpoint** which maps to the DSS installation folder. Here we want to specify the initial state document in the **store** directory, so it should be **"store/ServiceTutorial3.xml"**. We can also point to that file using this combination: **ServicePaths.Store + "/ServiceTutorial3.xml"**.

The service manifest can provide a runtime override of any partner declaration. This can allow the service to load and persist state from a different location.

Modify the state declaration to look like the following:

```
/// <summary>
/// Service State
/// </summary>
[ServiceState]
[InitialStatePartner(Optional = true, ServiceUri = ServicePaths.Store + "/ServiceTutorial3.xml")]
private ServiceTutorial3State _state = new ServiceTutorial3State();
```

Because you specified **Optional = true** when you declared the initial state partner, the service will start even if no initial state document is found. In that case the **_state** field will be null. Add this check in the **Start** method, just before the call to **base.Start()**, to provide default initialization.

```
// The initial state is optional, so we must be prepared to
// create a new state if there is none
if (_state == null)
{
    _state = new ServiceTutorial3State();
}
```

Step 2: Persist a State

Currently the service attempts to load a document on startup and never finds one. Add a capability to persist state to the same location.

The exact logic for when to persist state is dependent on the nature of the service in question. In this service, the state is persisted every tenth tick. To persist the state, call the **SaveState** method of the **DsspServiceBase** class and provide the current state value. This call wraps an asynchronous process which posts a **Replace** message to the **Mount** service (that provides file-system capabilities to a service). When the state object is posted to the Mount service, that object is cloned. This prevents later modification of the service state from corrupting the serialization process and allows modification of the service state to happen concurrently with the operation of the service.

Add the following code to the **IncrementTickHandler** method after the line that logs the tick count.

```
if (_state.Ticks % 10 == 0)
{
    LogInfo("Store State");
    base.SaveState(_state);
}
```

You can now cause the state to be persisted by running the service for more than 10 seconds. Stopping the DSS node and restarting the service causes the service to resume the last saved state.

As mentioned above the state is now saved in the file, **store\ServiceTutorial3.xml**. Run the service for a while and then take a look at the contents of that file.

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceTutorial3State xmlns="http://schemas.tempuri.org/2006/06/servicetutorial3.html">
  <Member>This is my State!</Member>
  <Ticks>530</Ticks>
</ServiceTutorial3State>
```

Now open your browser to http://localhost:50000/servicetutorial3 and compare.

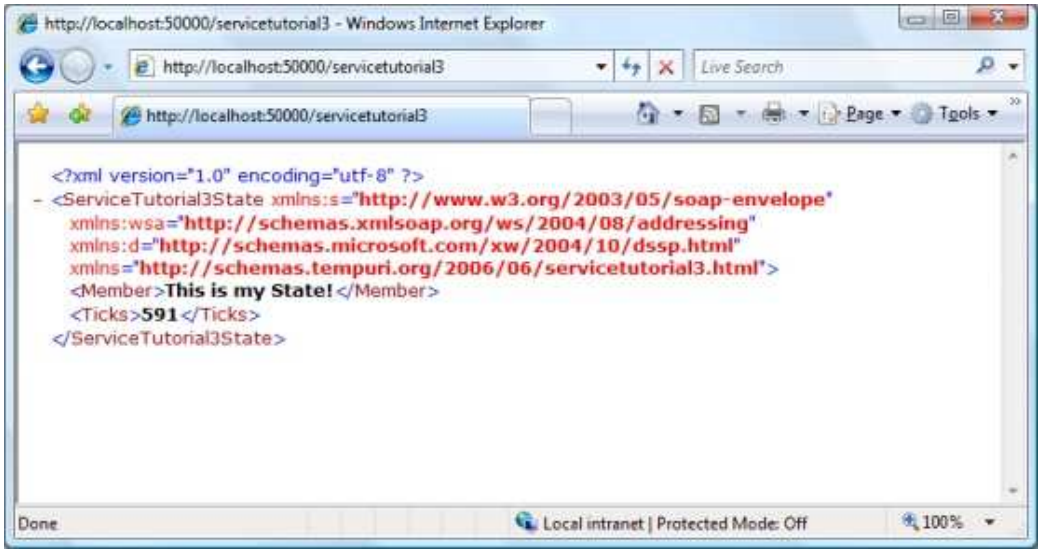

Figure 1 - Viewing the state of ServiceTutorial3 in browser.

The state, as serialized to file, is identical in form to the state as serialized to HTTP. Now stop the service and restart it. Notice that the **Tick** counter resumes from the last saved state.

Step 3: Manage Asynchronous Responses

While persisting the state above, we ignored the possibility of an error. You can make a simple change to the service to check whether **SaveState** succeeded or failed.

Create a task, in this case a **Choice**, and return that task with the **yield return** C# keyword.

Returning a task with **yield return** causes the handler to enter a wait state until the task completes. This allows you to create sequences of asynchronous operations.

The **SaveState** method posts a **Replace** message to the **Mount** service. The **SaveState** method returns the **Response** port for that message.

The **Choice** task executes the handler for the first message posted to a **PortSet** with more than one message type. In this case the **ResponsePort** defines two message types:

- **DefaultReplaceResponseType** - used for success
- **Fault** - used for failure conditions

In the **IncrementTickHandler**, replace the line that calls **base.SaveState(_state)** with the code snippet below. Here the handlers are two anonymous delegates, one for each message type. If a fault is reported, log an error.

```
// SaveState() sends a message so the process is not complete
// until a response is received. It is good practise to check
// for a Fault whenever you send a message.
yield return Arbiter.Choice(
    base.SaveState(_state),
    delegate(DefaultReplaceResponseType success) { },
    delegate(W3C.Soap.Fault fault)
    {
        LogError(null, "Unable to store state", fault);
    }
);
```

To see the effects of this change, change the permissions of the file, **store\ServiceTutorial3.xml**, to be read only. This causes the **SaveState** message to fail and your error message to be logged. Take a look at the **ConsoleOutput**:

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/ServiceTutorial3.manifest.xml
3	14:50:58	**	Service uri: http://localhost:50000/servicetutorial3
4	14:50:57	**	Initial manifest loaded successfully.
5	14:50:59	**	Tick: 371
6	14:50:59	**	Tick: 372
7	14:51:00	**	Tick: 373
8	14:51:01	**	Tick: 374
9	14:51:02	**	Tick: 375
10	14:51:03	**	Tick: 376
11	14:51:04	**	Tick: 377
12	14:51:05	**	Tick: 378
13	14:51:06	**	Tick: 379
14	14:51:07	**	Tick: 380
15	14:51:07	**	Store State
16	14:51:07	***	Unable to store state: http://www.w3.org/2003/05/soap-envelope:Receiver -> http://schemas.microsoft.com/xw/2004/10/dssp.html:OperationFailed
17	14:51:08	**	Tick: 381
18	14:51:09	**	Tick: 382
19	14:51:10	**	Tick: 383
20	14:51:11	**	Tick: 384
21	14:51:12	**	Tick: 385
22	14:51:13	**	Tick: 386

Summary

In this tutorial, you took the simple clock service written in Service Tutorial #2 and modified it to persist and load state using an Initial State Partner. You have also learned one way of waiting for the response from an asynchronous operation.

To do this, you completed the following:

- Read an Initial State from a File.
- Persist a State.
- Manage Asynchronous Responses.