

# **WSRF.NET Developer Tutorial**

**Mark Morgan**

**Glenn Wasson**

# Table of Contents

|   |    |
|---|----|
| WSRF.NET Developer Tutorial .....                                       | 1  |
| Table of Contents .....   | 2  |
| Table of Code Examples .....  | 3  |
| Chapter 1: Abstract .....   | 4  |
| Chapter 2: A Simple WSRF Web Service – The Auction .....                | 5  |
| Setting It All Up .....   | 5  |
| Preparing the Auction Service for Endpoint Creation .....               | 9  |
| Creating the Command-line Client .....                                  | 10 |
| Chapter 3: Manipulating WS-Resources and User-Defined Web Methods ..... | 12 |
| Adding Persistent State to a Web Service .....                          | 12 |
| Adding new Web Methods .....  | 13 |
| Defining the Bidder Information .....                                   | 17 |
| Chapter 4: WSRF Resource Properties .....                               | 21 |
| Manipulating Resource Properties on a Service .....                     | 21 |
| Modifying the Client to Pass Construction Parameters .....              | 26 |
| Accessing a Service’s Resource Properties .....                         | 26 |
| Chapter 5: WS-Notification and WSRF.NET .....                           | 29 |
| Preparing for a Notification Scenario .....                             | 29 |
| Modifying the Client to Subscribe to the Producer .....                 | 33 |
| Using a Service as a Consumer .....                                     | 38 |
| Chapter 6: Using Service Groups .....                                   | 43 |
| Creating the Service Group and Adding Members .....                     | 43 |
| Modifying the Client .....  | 46 |
| Appendix A: Final Code Listings .....                                   | 50 |
| AuctionConstructionParameters.cs .....                                  | 50 |
| BidderInformation.cs .....  | 51 |
| Auction.asmx.cs .....   | 52 |
| SubscriptionManager.asmx.cs .....                                       | 55 |
| Consumer.asmx.cs .....  | 56 |
| AuctionManager.asmx.cs .....  | 58 |
| AuctionManagerEntry.asmx.cs .....                                       | 59 |
| Class1.cs .....   | 60 |
| Appendix B: References .....  | 66 |

# Table of Code Examples

|  |    |
|--|----|
| Code Example 1: Initial Code for AuctionService.asmx.....                        | 6  |
| Code Example 2: Auction Service with Minimal WSRF.NET Attributes .....           | 7  |
| Code Example 3: Modifying the web.config File.....                               | 8  |
| Code Example 4: AuctionService with Lifetime Management Port Types.....          | 10 |
| Code Example 5: Initial Client Implementation .....                              | 11 |
| Code Example 6: Adding Persistent State Elements to WSRF.NET Services .....      | 13 |
| Code Example 7: Adding a New Web Method to the AuctionService.....               | 14 |
| Code Example 8: Calling the "makeBid" Method.....                                | 16 |
| Code Example 9: BidderInformation Class .....                                    | 18 |
| Code Example 10: Adding the Current Bidder to the Auction Service.....           | 19 |
| Code Example 11: Adding the Bidder Information to the Client .....               | 20 |
| Code Example 12: Turning _currentBid into the CurrentBid Resource Property ..... | 22 |
| Code Example 13: Adding the AuctionDescription Resource Property.....            | 22 |
| Code Example 14: Creating the AuctionConstructionParameters Class .....          | 24 |
| Code Example 15: Constructing a New Endpoint with Construction Parameters.....   | 25 |
| Code Example 16: Passing Construction Parameters from the Client.....            | 26 |
| Code Example 17: Retrieving Resource Properties from a Service .....             | 28 |
| Code Example 18: Raising Notifications from a Service .....                      | 30 |
| Code Example 19: Creating the Subscription Manager.....                          | 32 |
| Code Example 20: Adding Configuration Elements for Notification Producers .....  | 33 |
| Code Example 21: Receiving Notifications.....                                    | 35 |
| Code Example 22: Subscribing the Consumer to the Producer .....                  | 37 |
| Code Example 23: Creating the Consumer Service.....                              | 38 |
| Code Example 24: Using Notifications from a Consumer Service .....               | 40 |
| Code Example 25: Subscribing the Consumer Service .....                          | 42 |
| Code Example 26: The AuctionManager Service .....                                | 44 |
| Code Example 27: AuctionManagerEntry Service.....                                | 45 |
| Code Example 28: Configuring the ServiceGroup Port Type .....                    | 45 |
| Code Example 29: Creating the AuctionManager Service .....                       | 47 |
| Code Example 30: Setting Up to Add to the Manager .....                          | 48 |
| Code Example 31: Adding Entries to a Service Group .....                         | 49 |

# Chapter 1: Abstract

WSRF (the Web Services Resource Framework) is a new set of specifications (<http://www.globus.org/wsrf>) that provide a common framework on which future grid technology can be developed. This framework defines the concept of “stateful resources” which can be discovered, queried, and manipulated via web services. Close on the heels of the release of these documents, numerous software releases are expected to leverage the WSRF specifications. WSRF.NET is the first public release of such a software product.

This tutorial is meant to provide a gentle introduction to the WSRF.NET library and programming model by giving readers a simple, hands-on example that they can follow. It will guide readers through the various stages of developing a WSRF.NET web service and will provide introductory examples of various components and routines available to a WSRF.NET programmer.

To motivate this tutorial, a single software development task is described and maintained throughout the chapters. This example -- to implement a set of web services and client routines for an auctioning system -- will be a common theme in all included topics.

In order to correctly implement the examples in this tutorial, some assumptions about the state of the reader’s machine are made. These assumptions about the development environment are essential should you wish to follow the examples verbatim. In particular, I will assume:

- A) That you have a valid installation of Visual Studio .NET and that you are able to create and manipulate Web Service projects inside that environment.
- B) That you have installed the WSRF.NET system on your machine
- C) That you have correctly configured WSRF.NET and that you meet all the minimum requirements for running it (see <http://www.cs.virginia.edu/~gsw2c/WSRFdotNet/install.html>).

Throughout this tutorial, numerous concepts will be introduced and expounded on. These concepts will include the basic mechanisms for turning a C# .NET web service into a WSRF.NET web service, how to manipulate your web service so that it can create and manage *WS-Resources*, how to reflect those resources to the outside world, and how to use additional services and service mechanisms to achieve even more functionality. In Chapter 2 we will create a simple .NET web service and then modify it to make the web service a WSRF.NET compliant web service. This will involve adding attributes to the class and marking it as a WSRF web service in its project properties. Chapter 3 will continue by looking at the topic of creating your own *WS-Resource* types and how WSRF.NET allows you to persist or retrieve those resources from a database. It will also take a brief look at how WSRF.NET allows you to add your own web methods to your web service. In Chapter 4, we’ll take time to examine *WS-ResourceProperties* and how your service can export or “project” its own properties to the outside world. Chapters 5 and 6 will finish the tutorial by describing respectively how *WS-Notification* and *WS-ServiceGroups* can be used by your new web service to facilitate asynchronous notification of events and grouping of *WS-Resources* together.

## Chapter 2: A Simple WSRF Web Service – The Auction

The first WSRF service that we are going to write in this tutorial is an **Auction** service. Initially this service will be very basic but writing it will provide invaluable exposure to some of the lower level details involved in using WSRF.NET. By the end of this chapter you will have implemented a WSRF.NET web service that can create and destroy its own *WS-Resources* via mechanisms provided by WSRF.NET. In following chapters, we will add state to this service and begin implementing some of our own web methods.

### Setting It All Up

To begin, we need to set up a Visual Studio .NET solution suitable for developing the software in this tutorial. Start Visual Studio .NET and create a new blank solution. Once the solution is open, add a new C# based Web Service project to the solution called **AuctionServices** (see Figure 1).

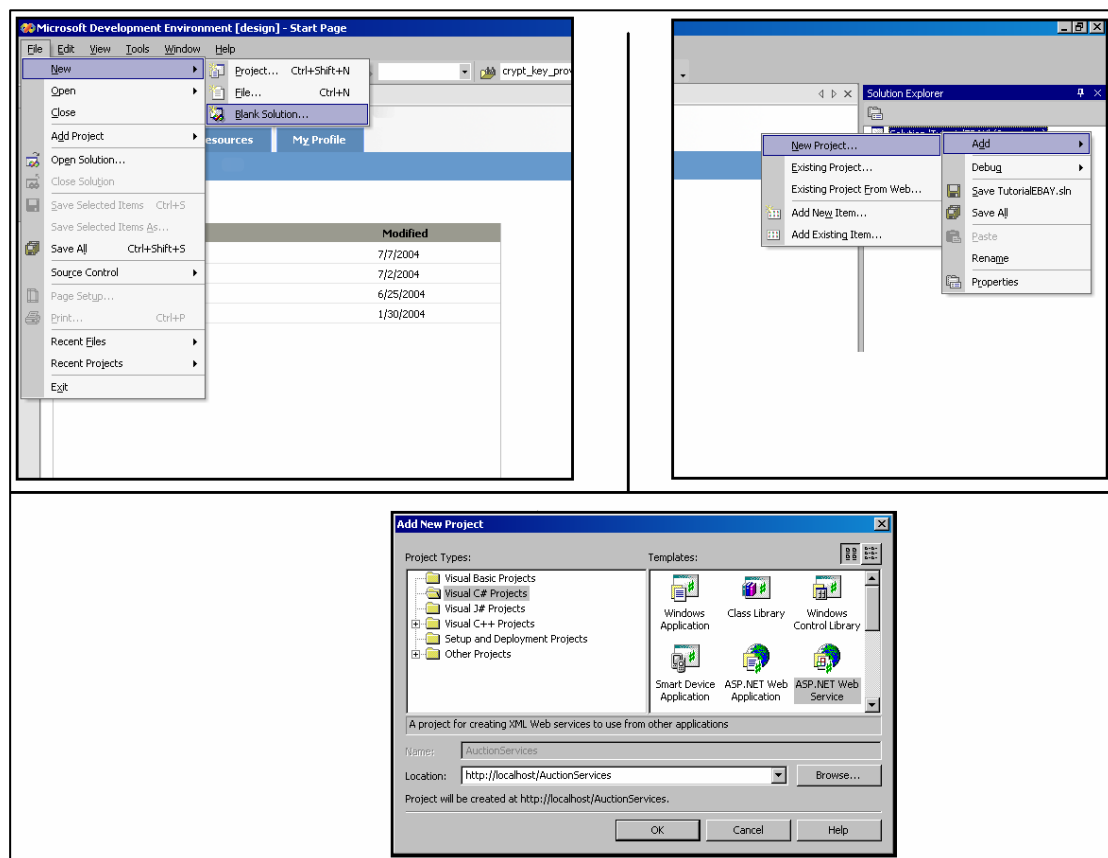


Figure 1: Visual Studio .NET Solution and Project Creation

Visual Studio will automatically create a new service in the Web Service project called “Service1.asmx”, but for the sake of clarity, go ahead and rename this

service to “Auction.asmx”<sup>1</sup>. Also, to save on space in this tutorial, all “auto-generated” C# comments will be removed (though of course you are welcome to leave them in your own code). I will also make a habit of “collapsing”, or simply omitting, any generated regions of code (such as the *Component Designer generated code* section which is created for each new Visual Studio .NET C# Web Service<sup>2</sup>).

Once the web service project has been successfully created and the Service1.asmx file renamed, you should be able to view the source in that file and see something similar to the code block shown in Code Example 1.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace AuctionServices
{
    public class Service1 : System.Web.Services.WebService
    {
        public Service1()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        ...
        #endregion
    }
}
```

**Code Example 1: Initial Code for AuctionService.asmx**

At this point, you have a fully-functional Windows ASP.NET web service though the service itself does not do anything particularly interesting right now. The next steps will modify this web service so that it is a WSRF.NET compliant web service.

Before we can write any code for the new Auction service, we need to add assembly references to the WSRF.NET library dlls (**WSRFCommonLib** and **WSRFServiceLib**). We also need to add an assembly reference to the Microsoft WSE 3.0 library (**Microsoft.Web.Services3**). Also, we can greatly improve the readability of our code by adding *using* statements for many of the namespaces from WSRF.NET that we will be using (one for **UVa.GCG.WSRF.Common.Attributes** and another for **UVa.GCG.WSRF.Service.BaseTypes**).

To a large degree, most of the programmer’s interaction with the WSRF.NET system is via .NET attributes. These attributes allow WSRF.NET to control various aspects of the component web services. Everything from WSDL generation, to naming, to persistent state manipulation, to port type aggregation is controlled via these attributes. The first thing that we will do with the **Auction** service is to add

---

<sup>1</sup> Renaming the .asmx file does not rename the C# class contained inside. This is unfortunate, but we won’t change the name of the C# class in this tutorial because doing so correctly is unfortunately non-trivial.

<sup>2</sup> This auto-generated code must not be removed from a working example as it is a required component for Visual Studio .NET.

some attributes to the **Service1** class that indicate to WSRF.NET how to generate appropriate WSDL. We also change the base class off of which the service class derives. The new code (shown in Code Example 2) includes a **[WsdLBaseName]** attribute which is of particular importance to WSRF.NET. This attribute describes to WSRF.NET the base name to use when generating WSDL names as well as the namespace in which to place the WSDL elements.

In order to deploy a WSRF.NET service, it is necessary to run the WSRF.NET port type aggregator tool over the assembly generated by compiling your web service project. While this can be done by hand, the easiest way to use the port type aggregator is by making use of the WSRF.NET service add-in tool. To use this tool, open the properties for your .asmx file (by right-clicking on the file in the solution explorer and selecting the Properties menu item) and make sure that the WSRF property is set to true (see Figure 2). Doing so marks that service as a WSRF.NET service and makes sure that the port type aggregator tool is run every time your service code is compiled.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;

namespace AuctionServices
{
    [WsdLBaseName("Auction",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    public class Service1 : ServiceSkeleton
    {
        ...
    }
}
```

**Code Example 2: Auction Service with Minimal WSRF.NET Attributes**

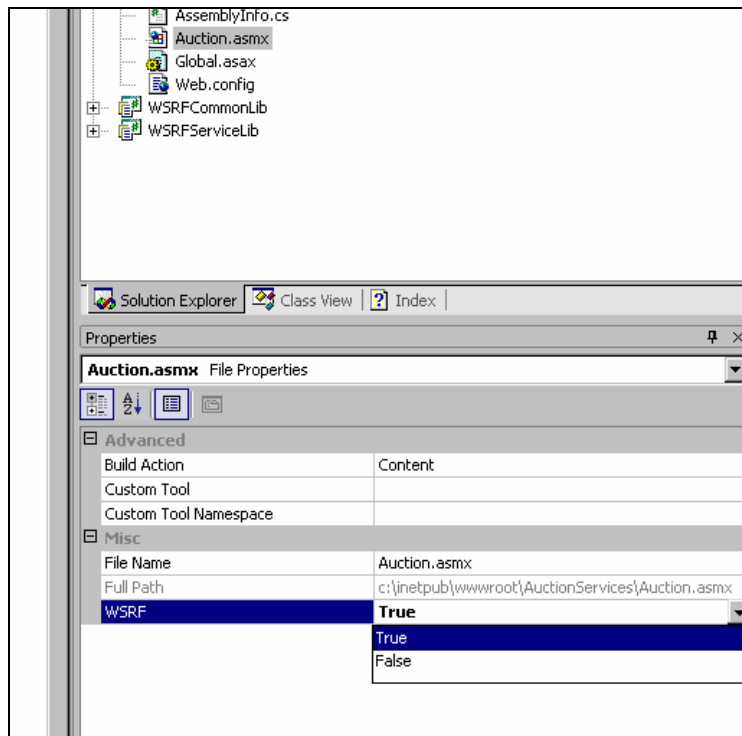


Figure 2: Selecting the WSRF Property

Finally, you need to configure your service to use the Web Services Extensions (WSE 3.0). This procedure (described by the WSE 3.0 documentation) involves modifying the web.config file that was created when you made the web service project. Open that configuration file and make the modifications indicated by Code Example 3 below<sup>3</sup>.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="microsoft.web.services3"
type="Microsoft.Web.Services3.Configuration.WebServicesConfiguration,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
  </configSections>
  <system.web>
    <webServices>
      <soapExtensionImporterTypes>
        <add
type="Microsoft.Web.Services3.Description.WseExtensionImporter,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
      </soapExtensionImporterTypes>
      <soapServerProtocolFactory
type="Microsoft.Web.Services3.WseProtocolFactory,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
    </webServices>
  </system.web>
</configuration>
```

Code Example 3: Modifying the web.config File.

<sup>3</sup> When adding entries to the web.config file, it is important to keep all string attribute values on a single line. Due to space limitations in this document, they are shown with carriage returns that should not in fact be reflected in the configuration file itself. For any string inside this file which appears inside of double quotes, do not put carriage returns or spaces unless indicated explicitly.

## Preparing the Auction Service for Endpoint Creation

Part of the point of the WSRF specifications is to provide a standard way of interacting with stateful web services. We have not yet talked about how to persist and restore Resource information for your WSRF.NET services and I'll hold off doing so until a later chapter, but in the mean time, it's still useful to create new *WS-Resources* for your service.

Why we are creating these Resources is also an interesting question that deserves a little discussion as well. We have thus far been creating a service that will handle auctions for an online bidding system. However, this isn't the end of the story. The vision for this project is to create a WSRF.NET service whose Resources will each individually represent a single ongoing auction. Each *WS-Resource* will contain information about the current bid for that auction, information about the bidder who currently has the highest bid, information about the individual offering the item up for auction, and finally a textual description of the auction Resource. In this sense, the creation of a new Auction service Resource represents the creation of a single auction for a single auction-able item. Conversely, the destruction of a *WS-Resource* represents the closing of, or removal of an individual auction. The auction service is merely a "portal" to the *WS-Resource* that is made visible to the outside world.

The WSRF specification does not explicitly define the process of *WS-Resource* creation (though *WS-ResourceLifetime* [3] specifies how *WS-Resources* are destroyed). WSRF.NET however provides some common ways of instantiating or creating Resources. In order to do this you have to "aggregate" in a new port type, provided by WSRF.NET, called the *GCGResourceFactory* port type.

Port type aggregation is a concept hinted at by the WSRF documents and implemented for you by WSRF.NET. Put simply, it is a means of aggregating or "pasting" together a number of different port types into a single web service. Most of the work for port type aggregation is handled automatically for you merely by setting the WSRF property on a service in Visual Studio .NET. The service author simply indicates, via .NET attributes, which port types should be included in the service. For the purposes of lifetime management of our new endpoints, we need to aggregate in two different port types. One is the *WS-ResourceLifetime* port type, *ImmediateResourceTermination*. Including this port type in the **Auction** service will give clients the ability to ask for a specific endpoint to terminate, or exit, immediately. The other port type that we will aggregate in is the *GCGResourceFactory* port type. This port type is not a standard port type in the WSRF specification, but is provided by WSRF.NET for the programmer's convenience. Including these port types into your service is shown in Code Example 4 below.

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;

namespace AuctionServices
{
    [WsdlBaseName("Auction",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    public class Service1 : ServiceSkeleton
    ...
}

```

#### Code Example 4: AuctionService with Lifetime Management Port Types

Once the changes indicated in Code Example 4 have been made, you should be able to build your new service. The next step in this tutorial will be to create a new client side tool that you can use to start testing your WSRF web service.

## Creating the Command-line Client

Now that we have a functional WSRF.NET web service with interesting lifetime management functionality, it's time to create a client application capable of exercising this service. To begin, create a new C# Console Application under your solution and call it **AuctionDriver**. As before, you will need to add a number of assembly references to this project, namely:

- 1) WSRFCommonLib
- 2) Microsoft.Web.Services3
- 3) System.Web.Services

Once these references have been added, open and modify the client code as shown in Code Example 5. Notice that a binding to the GCGResourceFactory port type is instantiated and that an out call is made through it to "create" a new endpoint. This web method call is being handled by the GCGResourceFactory port type that we added to our **Auction** service in the last section. Also note that an instance of the ImmediateResourceTerminationProxy is later used to destroy this endpoint. Similarly, this action is being handled on the service side by the ImmediateResourceTermination port type.

Individuals familiar with developing web service applications will undoubtedly recognize that we are using proxy clients generated from the WSDL for our target service. These proxies are built in such a way that they can make web method calls on the target service with correctly formatted soap and so that they can subsequently receive a properly formatted response. Unlike most web service development in Visual Studio .NET, WSRF.NET provides proxies for most of the WSRF.NET included port types. While it's still possible to generate your own proxies (by selecting "Add Web Reference" for your project), we recommend that, when possible, you use the proxies provided by WSRF.NET as they have been

designed to work with data types included in the WSRF.NET library. As you may be aware, when “Add Web Reference” generates a new proxy, it creates data types for all of the parameters and result values within the proxy’s namespace. Unfortunately, this method of code generation causes the same data types to have different names (because they have different namespaces) for each proxy<sup>4</sup> and this makes development of a client-side programming library difficult.

```
using System;
using System.Xml;
using UVa.GCG.WSRF.Common.WS.Addressing;
using UVa.GCG.WSRF.Common.WS.Grid;
using UVa.GCG.WSRF.Common.WS.ResourceLifetime;

namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            EndpointReferenceType auction = createAuction(
                "http://localhost/AuctionServices/Auction.asmx");
            destroyEndpoint(auction);
        }

        static private EndpointReferenceType createAuction(
            string serviceURL)
        {
            GCGResourceFactoryBinding proxy =
                new GCGResourceFactoryBinding(serviceURL);
            Create creationParms = new Create();

            CreateResponse response =
                proxy.Create(creationParms);

            return response.ResourceEndpoint;
        }

        static private void destroyEndpoint(EndpointReferenceType epr)
        {
            ImmediateResourceTerminationProxy proxy =
                new ImmediateResourceTerminationProxy(epr);
            proxy.Destroy(new Destroy());
        }
    }
}
```

#### Code Example 5: Initial Client Implementation

This concludes this chapter on basic WSRF.NET functionality. So far we have learned how to correctly mark a web service as a WSRF.NET web service and we have learned how to aggregate multiple port types together into a single service. In the next chapter we will cover various ways in which you can manipulate your own private state. We will also cover how to create your own WSRF compliant web methods.

---

<sup>4</sup> In other words, suppose that you have one WSRF.NET web service which returns an EndpointReferenceType as a result. Later, you may want to use that endpoint value as a parameter to another web service. If you use “Add Web Reference” to generate proxies for both of these services, then each auto-generated file will contain its own definition for EndpointReferenceType and you will be required to manually convert between the two.

## Chapter 3: Manipulating WS-Resources and User-Defined Web Methods

In this chapter we will begin to look at how you can write your own port types/services which can manipulate *WS-Resources*. We will also write our own web method to access our web service.

### Adding Persistent State to a Web Service

Persistent state is one of the cornerstones of WSRF and presents itself in the form of *WS-Resources*. At its most basic level, the WSRF specification is a specification for making web services stateful and for accessing those stateful Resources. WSRF.NET recognizes the importance of this fact and the programming model is designed to make dealing with *WS-Resources* easy. WSRF.NET stores *WS-Resources* in a database and the relevant resources are loaded automatically by a WSRF.NET web service whenever a new web method invocation arrives. Upon completion of that method, the *WS-Resource* is once again persisted back to a database. Adding data to your web service's *WS-Resource* is as easy as adding a new .NET attribute to a class data member. Specifically, any data member of a WSRF.NET web service class can be made into part of that service's *WS-Resource* by adding the **[Resource]** attribute to that member. Further, you can also add this attribute to a .NET property provided that both a set and get accessor are available.

First let's add some new data components to the *WS-Resource* type for our existing Auction service. In particular, we will add state to maintain the current bid that has been placed on the auction. Code Example 6 shows the simple modification to our class. Notice in the code example that we also overrode a method from our base class called *InitResource*. This method allows programmers to control the initialization of new *WS-Resources* and is similar in function to a class constructor. However, *InitResource* only gets called once for each new *WS-Resource* that is created rather than every time the class is instantiated (which for .NET web services happens every time a method invocation comes in to the service). For the moment, ignore the *parms* parameter (a WSRF.NET parameter which provides a way of passing initialization parameters to the *InitResource* method) -- we will hold off discussing this for a few pages.

```

public class Service1 : ServiceSkeleton
{
    [Resource]
    private int _currentBid;

    public Service1()
    {
        InitializeComponent();
    }

    #region Component Designer generated code
    ...
    #endregion

    public override void InitResource(Hashtable parms)
    {
        _currentBid = 0;
    }
}

```

**Code Example 6: Adding Persistent State Elements to WSRF.NET Services**

## **Adding new Web Methods**

Now that we can correctly store the current bid on an auction, its time to provide a method that would allow a client to place a new bid. Adding a new web method to a WSRF.NET web service is just like adding a web method to any ASP.NET web service. You simply need to add the correct public C# method to your class and annotate that method with appropriate web method attributes. In Code Example 7 below I have done just that.

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;

namespace AuctionServices
{
    [WsdBaseName("Auction",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    public class Service1 : ServiceSkeleton
    {
        ...

        [WebMethod]
        [SoapDocumentMethod(
            "http://wsrfnet.cs.virginia.edu/auction-tutorial/bid")]
        [return: XmlElement("bidSuccessful",
            Namespace="http://wsrfnet.cs.virginia.edu/auction-tutorial")]
        public bool bid(int newBid)
        {
            if (newBid <= _currentBid)
                return false;

            _currentBid = newBid;
            return true;
        }
    }
}

```

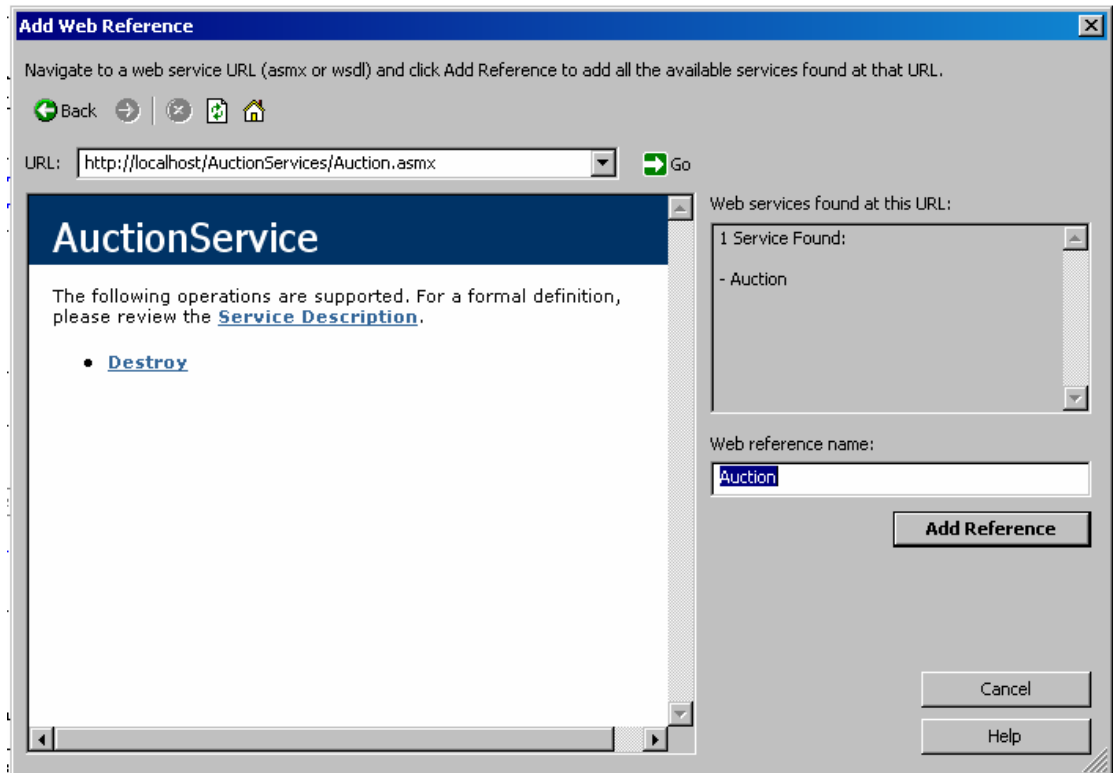
**Code Example 7: Adding a New Web Method to the AuctionService**

Now that we have added the new code to the service, it is time to modify the client application to test our new functionality. The process of doing this is nearly identical to that for using any other ASP.NET web service method.

First, add a new web reference to the client console application. The web reference should refer to your Auction service (e.g. <http://localhost/AuctionServices/Auction.asmx>). For the purposes of this tutorial, call that web reference “Auction” (see Figure 3<sup>5</sup>).

---

<sup>5</sup> Developers familiar with the *Add Web Reference* dialog in Visual Studio .NET may notice that a number of web methods which you might expect to see are not listed in the preview pane for the Auction service. This is due to a bug in Visual Studio .NET and will not affect the generated stubs. The methods are in fact there and are merely being omitted from the preview.



**Figure 3: Adding the Auction Web Reference**

The step of adding the web reference in Visual Studio will have the effect of creating a new C# proxy class that can be used to communicate with the WSRF.NET web service. We must next create an instance of this class, use a utility method provided by WSRF.NET to set the EPR data for this proxy (in other words, take the endpoint reference for our *WS-Resource* and set the information in the proxy headers to correctly indicate this Resource in outgoing method calls), and finally invoke the bid method on our web service. The code for the new client is shown in Code Example 8.

```

using System;
using System.Xml;
using UVa.GCG.WSRF.Common.WS;
using UVa.GCG.WSRF.Common.WS.Addressing;
using UVa.GCG.WSRF.Common.WS.Grid;
using UVa.GCG.WSRF.Common.WS.ResourceLifetime;

namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            EndpointReferenceType auction = createAuction(
                "http://localhost/AuctionServices/Auction.asmx");

            AuctionDriver.Auction.AuctionServiceWse bidProxy =
                new AuctionDriver.Auction.AuctionServiceWse();
            WSUtilities.setEPR(bidProxy, auction);
            makeBid(bidProxy, 5);
            makeBid(bidProxy, 1);
            makeBid(bidProxy, 10);

            destroyEndpoint(auction);
        }

        static private void makeBid(
            AuctionDriver.Auction.AuctionServiceWse bidProxy,
            int newBid)
        {
            Auction.bid bid = new Auction.bid();
            bid.bidder = info;
            bid.newBid = newBid;
            if (bidProxy.bid(bid).bidSuccessful)

                Console.WriteLine("Bid of {0} succeeded.", newBid);
            else
                Console.WriteLine("Bid of {0} failed!", newBid);
        }

        static private EndpointReferenceType createAuction(
            string serviceURL)
        {
            GCGResourceFactoryBinding proxy =
                new GCGResourceFactoryBinding(serviceURL);
            Create creationParms = new Create();

            CreateResponse response =
                proxy.Create(creationParms);

            return response.ResourceEndpoint;
        }

        static private void destroyEndpoint(EndpointReferenceType epr)
        {
            ImmediateResourceTerminationProxy proxy =
                new ImmediateResourceTerminationProxy(epr);
            proxy.Destroy(new Destroy());
        }
    }
}

```

**Code Example 8: Calling the "makeBid" Method**

## Defining the Bidder Information

Now that we have finally gotten to the point where we can write our own methods for our web service, its time to consider what happens when we store state or pass parameters for classes that aren't provided by either .NET or by WSRF.NET. When passing parameters for a web method call, the rules are exactly the same as they are for any other ASP.NET web method – the parameters must be XML serializable. However, any state that WSRF.NET stores in its database, must be binary serializable (i.e. processable by the **BinaryFormatter** class). This will sometimes mean that when you write new data types for your applications, they will have to be written as both binary and XML serializable types. While this restriction may be relaxed in the future (by making everything XML serializable), for now it's necessary.

Now let's make the web service keep track, not only of the current highest bid that an auction has received, but also of the bidder who placed that bid. In order to do this, we'll create a new data type (called **BidderInformation**) which will store relevant pieces of information about the current highest bidder. We'll then pass that information along with each *bid* method call and store it on the server side as a part of one of the auction service's *WS-Resource*.

To begin, create a new project item under your **AuctionServices** project called BidderInformation.cs and write a class that can store both the bidder's name as well as his/her email address. The class must be both binary and XML serializable. My version of this class appears in Code Example 9.

```

using System;

namespace AuctionServices
{
    [Serializable]
    public class BidderInformation
    {
        private string _name;
        private string _email;

        public string Name
        {
            get { return _name; }

            set { _name = value; }
        }

        public string Email
        {
            get { return _email; }

            set { _email = value; }
        }

        public BidderInformation(string name, string email)
        {
            _name = name;
            _email = email;
        }

        public BidderInformation() : this(null, null)
        {
        }
    }
}

```

#### Code Example 9: BidderInformation Class

Next we need to modify the existing Auction service to store and manipulate the current bidder information based on values passed to the service via the *bid* method. In general, this is very straight forward – you simply create a private data member for the class (annotated with the **[Resource]** attribute), just like we did for the current bid. You also need to initialize the new data member in the *InitResource* method just like we did before with the current bid data item. Finally, you must modify the bid method to take this additional information as a parameter when called. These changes can be seen below in Code Example 10. Make these changes and compile the service.

```

[WsdLBaseName("Auction",
    "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
[WebService]
[WebServiceBinding]
[WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
[WSRFPortType(typeof(GCGResourceFactoryPortType))]
public class Service1 : ServiceSkeleton
{
    [Resource]
    private int _currentBid;

    [Resource]
private BidderInformation _currentBidder;

    public Service1()
    {
        InitializeComponent();
    }

    #region Component Designer generated code
    ...
    #endregion

    public override void InitResource(Hashtable parms)
    {
        _currentBid = 0;
        _currentBidder = null;
    }

    [WebMethod]
    [SoapDocumentMethod(
        "http://wsrfnet.cs.virginia.edu/auction-tutorial/bid")]
    [return: XmlElement("bidSuccessful",
        Namespace="http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    public bool bid(BidderInformation bidder, int newBid)
    {
        if (newBid <= _currentBid)
            return false;

        _currentBid = newBid;
        _currentBidder = bidder;

        return true;
    }
}

```

**Code Example 10: Adding the Current Bidder to the Auction Service**

Finally, we need to modify the client code used to test the new bid method. Right-click in the client project on the web reference that you created for the service. From the drop down menu, select the “Update Web Reference” option at the top. Your generated proxy should now reflect the changes made in the service. Change your client code to correctly create and pass a **BidderInformation** instance to the bid method<sup>6</sup>. Once this is complete, your code should look similar to the example shown below in Code Example 11.

---

<sup>6</sup> The Auction *service* uses the BidderInformation class you created to pass and store the values for the current bidder. However, the Auction *client* uses a new class that was generated for the web service proxy. This class will have identical data members to the server-side class from Code Example 9 but won’t have the same constructors and methods. This is because the proxy classes are generated from the service’s WSDL and not the original C# code.

```

static private void makeBid(
    AuctionDriver.Auction.AuctionServiceWse bidProxy,
    int newBid)
{
    AuctionDriver.Auction.BidderInformation info =
        new AuctionDriver.Auction.BidderInformation();
    info.Name = "Mark Morgan";
    info.Email = "mmm2a@cs.virginia.edu";
    Auction.bid bid = new Auction.bid();
    bid.bidder = info;
    bid.newBid = newBid;

    if (bidProxy.bid(bid).bidSuccessful)
        Console.WriteLine("Bid of {0} succeeded.", newBid);
    else
        Console.WriteLine("Bid of {0} failed!", newBid);
}

```

**Code Example 11: Adding the Bidder Information to the Client**

At this point, you should once again have a working application that you can run and test. This concludes this chapter on resource elements and custom, user-defined web methods. In the next chapter we'll begin discussing the various ways that *WS-ResourceProperties* can be used in your applications.

## Chapter 4: WSRF Resource Properties

In this chapter, we will begin discussing Resource Properties in WSRF.NET. The WSRF specification describes Resource Properties as “projections” of a service’s *WS-Resource*. This means that Resource Properties expose the state of a *WS-Resource* via some transformation, which could be a null. Resource Properties can be discovered, queried and modified by clients through interaction with the service that owns the *WS-Resource*. In WSRF.NET, you can add the **[ResourceProperty]** attribute to any C# properties or data fields of a class to make them Resource Properties. .NET properties labeled as Resource Properties must have a get accessor. A set accessor is required only if you wish for your Resource Property to be modifiable by a client. Finally, WSRF.NET also provides an alternate syntax for you to define Resource Properties at the class level using .NET attributes. These class-level Resource Properties are stored internally by the WSRF.NET system and can be accessed and manipulated via a provided API. However, this mechanism for defining Resource Properties is not covered in this tutorial, see [13] for more information. In this chapter we’ll discuss how to define Resource Properties for your service and how to add various port types from the *WS-ResourceProperties* specification that allow clients to interact with them.

### Manipulating Resource Properties on a Service

To begin we will focus on manipulating Resource Properties in the Auction service. There are a number of Resource Properties that might make sense for an Auction service to have. For our example, we’ll create Resource Properties that describe the auction and reflect the current bid that an auction has. First we’ll focus on the last of these resource properties.

Our `currentBid` Resource holds the information we want to expose with the current bid Resource Property. WSRF.NET provides a short-hand for directly exposing part of a *WS-Resource*’s state, i.e. having a null transform between Resource and Resource Property, by placing both a **[Resource]** and **[ResourceProperty]** attribute on the data member. I’ve shown this below in Code Example 12. Note the parameter that we pass to the **[ResourceProperty]** attribute. This parameter identifies the name of the Resource Property that clients will use. If you do not specify this name, WSRF.NET will use the name of the data member as the name of the Resource Property.

```

[WsdBaseName("Auction",
    "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
[WebService]
[WebServiceBinding]
[WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
[WSRFPortType(typeof(GCGResourceFactoryPortType))]
public class Service1 : ServiceSkeleton
{
    [Resource]
    [ResourceProperty(Name="CurrentBid")]
    private int _currentBid;

    [Resource]
    private BidderInformation _currentBidder;

    ...
}

```

**Code Example 12: Turning `_currentBid` into the `CurrentBid` Resource Property**

The other Resource Property that we'd like to add to our service is the *AuctionDescription* property. In this case, we will do something a little trickier than before as far as creating the property. Let's assume this time that we want to store two pieces of information that describe the auction – the auction owner, and a textual description of the auction. However, instead of reflecting both of these items publicly to the outside world, we want to transform the data in some way and project that transformation to potential clients. To do this, we will create Resource elements for the two pieces of information (just like we did for `_currentBid` and `_currentBidder`). However, rather than put a **[ResourceProperty]** attribute on one or both of those state items, we will instead create a .NET property which transforms the data in some appropriate way and then label this property as a Resource Property. This code change is shown below in Code Example 13.

```

public class Service1 : ServiceSkeleton
{
    [Resource]
    [ResourceProperty(Name="CurrentBid")]
    private int _currentBid;

    [Resource]
    private BidderInformation _currentBidder;

    [Resource]
    private string _auctionDescription;

    [Resource]
    private string _auctionOwner;

    [ResourceProperty]
    public string AuctionDescription
    {
        get
        {
            return string.format(
                "{0} is offering \"{1}\" for auction.",
                _auctionOwner, _auctionDescription);
        }
    }

    ...
}

```

**Code Example 13: Adding the `AuctionDescription` Resource Property**

## Initializing Resources and Resource Properties

So far we have assumed that all state for a new *WS-Resource* can be created with reasonable default values (e.g. 0 for the current bid, null for the current bidder), but now that we have state that represents the auction description and the auction owner, this is no longer true. We could initialize these values to null and then later set them via some out of bounds means (a new “*set*” web method call, or by making the corresponding Resource Properties set-able), however, doing so leaves open the possibility that someone may try to access the service before these values are set and it is an in-elegant hack to say the least. The answer to our problem lies in the *GCGResourceFactory* port type that we talked about earlier. Recall that this port type is used to create new *WS-Resources* for a service. In order to initialize new Resources properly, we must pass construction parameters with our create calls to this port type.

In order to pass these parameters we will create a data type that can be used to store those values and pass them on the wire. Since this new type must be shared by both the client and the service projects, I recommend creating a new C# class library project in your solution. We will then include the assembly it generates in both the client and the service projects. Go ahead and create this project and call it **AuctionCommon**. Inside of this project, create a new file called [AuctionConstructionParameters.cs](#) and create a class inside of this file with the same name. This class should be capable of storing both the auction description, and the auction owner and it must be XML serializable. My copy of this class is shown in Code Example 14. You will also need to add assembly references to your new project or assembly in the existing **AuctionDriver** and **AuctionServices** projects.

```

using System;
using System.Xml.Serialization;

namespace AuctionCommon
{
    [XmlRoot("Auction",
        Namespace="http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [XmlType("Auction",
        Namespace="http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    public class AuctionConstructionParameters
    {
        private string _description;
        private string _owner;

        public string Description
        {
            get { return _description; }
            set { _description = value; }
        }

        public string Owner
        {
            get { return _owner; }
            set { _owner = value; }
        }

        public AuctionConstructionParameters(
            string description, string owner)
        {
            _description = description;
            _owner = owner;
        }

        public AuctionConstructionParameters() : this (null, null)
        {
        }
    }
}

```

**Code Example 14: Creating the AuctionConstructionParameters Class**

Next, we need to indicate to the GCGResourceFactory port type that this class is used for passing construction parameters. This is done by placing the **[ResourceInitializerType]** attribute on the service class. This attribute denotes a C# type that contains all the information needed to initialize the **[Resource]** annotated members of a port type. Each port type in a service may use an object of a different type to initialize its **[Resource]** members. When a service creates a new *WS-Resource*, a hash table is passed to the *WS-Resource* initialization routine. This hash table has keys that are the full type names of the port types in the service and values that are instances of the objects that each port type specifies with its **[ResourceInitializerType]** attribute. The *InitResource* method can use the objects in this hash table to place appropriate initial values in the members of a new *WS-Resource*. Finally, we need to change our code in our service class to make use of these new parameter values. I've shown the code for all of this in Code Example 15 below.

```

using AuctionCommon;
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml;
using System.Xml.Serialization;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;
using UVa.GCG.WSRF.Service.ResourceProperties;

namespace AuctionServices
{
    [WsdlBaseName("Auction",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [ResourceInitializerType(typeof(AuctionConstructionParameters))]
    public class Service1 : ServiceSkeleton
    {
        ...

        public override void InitResource(Hashtable parms)
        {
            AuctionConstructionParameters cons =
            parms[this.GetType().FullName] as
            AuctionConstructionParameters;

            _currentBid = 0;
            _currentBidder = null;

            _auctionDescription = cons.Description;
            _auctionOwner = cons.Owner;
        }

        ...
    }
}

```

**Code Example 15: Constructing a New Endpoint with Construction Parameters**

## Modifying the Client to Pass Construction Parameters

Now that we have a web service that can accept construction parameters during *WS-Resource* creation, it's time to modify the client so that it may pass appropriate values for those parameters. To do this, we modify the **creationParms** instance that, up until now, we created and left empty. I've shown this step in Code Example 16. Notice that we use the *WSUtilities.Serialize* method in our code. This is a utility method provided by WSRF.NET to make it easier to XML serialize data types.

```
using AuctionCommon;
using System;
using System.Xml;
using UVa.GCG.WSRF.Common.WS;
using UVa.GCG.WSRF.Common.WS.Addressing;
using UVa.GCG.WSRF.Common.WS.Grid;
using UVa.GCG.WSRF.Common.WS.ResourceLifetime;

namespace AuctionDriver
{
    class Class1
    {
        ...

        static private EndpointReferenceType createAuction(
            string serviceURL)
        {
            AuctionConstructionParameters cons =
                new AuctionConstructionParameters(
                    "2004 Kawasaki Ninja", "Mark Morgan");

            GCGResourceFactoryBinding proxy =
                new GCGResourceFactoryBinding(serviceURL);
            Create creationParms = new Create();

            creationParms.PortTypeInitializers = new XmlElement[]
            {
                WSUtilities.Serialize(cons)
            };

            CreateResponse response =
                proxy.Create(creationParms);

            return response.ResourceEndpoint;
        }
        ...
    }
}
```

Code Example 16: Passing Construction Parameters from the Client

Now build your entire solution and run your test client. In the next part of this chapter, we will see how to access a service's Resource Properties.

## Accessing a Service's Resource Properties

The *WS-ResourceProperties* specification includes a number of port types that can be used by clients to read and manipulate Resource Properties on a service. In particular, it is possible given these port types to retrieve values, update/add/delete values, and query a service for any properties that match certain patterns. Covering

all of these port types is beyond the scope of this document; however we will cover the first of these port types here.

In order for a service to allow clients to access their Resource Properties, the service must include the appropriate port type using the **WSRFPortType** attribute. In particular, you will need to add the `GetResourceProperty` port type from the `UVa.GCG.WSRF.Service.ResourceProperties` namespace. Rather than show the modified code, since the change is so trivial, I will merely state here that you need to add the following attribute to your Service class definition:

**[WSRFPortType(typeof(GetResourcePropertyPortType))]**

Once the above line has been included, you should be able to build your new service. The last step in this chapter is to change your command line client to access the service's new Resource Properties. As before, we will use one of the proxy classes included in `WSRF.NET` to communicate with the WSRF-defined `GetResourceProperty` port type. This code is shown below in Code Example 17.

```

using AuctionCommon;
using System;
using System.Xml;
using UVa.GCG.WSRF.Common.WS;
using UVa.GCG.WSRF.Common.WS.Addressing;
using UVa.GCG.WSRF.Common.WS.Grid;
using UVa.GCG.WSRF.Common.WS.ResourceLifetime;
using UVa.GCG.WSRF.Common.WS.ResourceProperties;

namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            EndpointReferenceType auction = createAuction(
                "http://localhost/AuctionServices/Auction.asmx");

            AuctionDriver.Auction.AuctionServiceWse bidProxy =
                new AuctionDriver.Auction.AuctionServiceWse();
            WSUtilities.setEPR(bidProxy, auction);

            getProperties(auction);

            makeBid(bidProxy, 5);
            makeBid(bidProxy, 1);
            makeBid(bidProxy, 10);

            destroyEndpoint(auction);
        }

        static private void getProperties(
            EndpointReferenceType auction)
        {
            GetResourcePropertyBinding proxy =
                new GetResourcePropertyBinding(auction);

            XmlElement val = proxy.GetResourceProperty(
                new XmlQualifiedName(
                    "AuctionDescription",
                    "http://wsrfnet.cs.virginia.edu/auction-tutorial")).Any[0]
                as XmlElement;
            Console.WriteLine("The auction description is \"{0}\".",
                val.InnerText);
        }

        ...
    }
}

```

#### Code Example 17: Retrieving Resource Properties from a Service

This chapter has focused on Resource Properties and how they can be used and manipulated by both clients and services. After running the console application and verifying that all of the new code works as expected, continue on to the next chapter where we will begin discussing one of the more advanced features that WSRF.NET provides – asynchronous notification.

## Chapter 5: WS-Notification and WSRF.NET

Among the more advanced WSRF port types that WSRF.NET supports are the Notification port types. Notification is the means by which consumers or “sinks” can register interest in receiving asynchronous messages relating to desired topics. This notification can occur either via direct communication from a Notification Producer, or through a Notification Broker which acts as an intermediary. Additionally, while the general case is that both producers and consumers will use the WSRF-defined port types, WSRF.NET also supports the notion of non-WSRF producers as well as consumers which are not themselves web services (i.e. client-side programs that are not running behind a web server).

This chapter of the tutorial will explore one of the forms of notification, in which the service implements the WSRF-defined NotificationProducer port type. Other types of notification (Brokered, Demand Based, etc. are discussed in [13]). We will, however, explore both forms of consumers (those that are web services, and those that are not).

### Preparing for a Notification Scenario

Unlike the examples shown so far, using notification in WSRF.NET requires additional configuration. In particular, the *WS-Notification* specification defines an additional port type, the SubscriptionManager port type, to manage the subscription process. WSRF.NET requires that you provide further configuration information in the service project’s `web.config` file for any service that implements the NotificationProducer port type. The Notification Producer service will use this information to locate its Subscription Manager service (which could be itself -- i.e. the same service could be both the Notification Producer and Subscription Manager).

For the first part of this chapter, I will discuss the case where we want the **Auction** service to notify the console client application when a bid is made on the auction. In order to accomplish this task, we will make the Auction service a Notification Producer (by adding the correct port type to the service) and we will add a new web service to the project to act as the Subscription Manager.

Adding the new port type to the Notification Producer is as easy as adding any of the other port types that we have thus far used -- we merely add an attribute indicating the name of the class to use. However, we must also add code to the service that, upon receipt of a new bid, will raise a notification event. This is shown in Code Example 18.

```

using AuctionCommon;
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml;
using System.Xml.Serialization;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;
using UVa.GCG.WSRF.Service.ResourceProperties;
using UVa.GCG.WSRF.Service.WS.Notification;

namespace AuctionServices
{
    [WsdlBaseName("Auction",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(GetResourcePropertyPortType))]
    [WSRFPortType(typeof(NotificationProducerPortType))]
    [ResourceInitializerType(typeof(AuctionConstructionParameters))]
    public class Service1 : ServiceSkeleton
    {
        ...

        protected override void portInit()
        {
            ServiceBase.Topics.addTopicHierarchy(
                new XmlQualifiedName("new-bid",
                    "http://wsrfnet.cs.virginia.edu/auction-tutorial"),
                true);
        }

        ...

        [WebMethod]
        [SoapDocumentMethod(
            "http://wsrfnet.cs.virginia.edu/auction-tutorial/bid",
            ParameterStyle=SoapParameterStyle.Bare)]
        [return: XmlElement("bidSuccessful",
            Namespace="http://wsrfnet.cs.virginia.edu/auction-tutorial")]
        public bool bid(BidderInformation bidder, int newBid)
        {
            if (newBid <= _currentBid)
                return false;

            _currentBid = newBid;
            _currentBidder = bidder;

            XmlQualifiedName topicDesc = new XmlQualifiedName("new-bid",
                "http://wsrfnet.cs.virginia.edu/auction-tutorial");
            ServiceBase.Topics[topicDesc].notify(newBid);

            return true;
        }
    }
}

```

**Code Example 18: Raising Notifications from a Service**

Notice in the code example above that a new member of the class, **ServiceBase**, is used. This member is a part of the base class (**ServiceSkeleton**) that your service class derives from and it gives you access to a number of useful methods and data members. In this case, we are using **ServiceBase** to access the topic set for the notification producer. This data member is used by producers first to register which topics the producer might produce events on (as was done in the *portInit* method) and then later to actually fire off events on those given topics when appropriate (as in the *makeBid* method). Next, we will create a new WSRF.NET web service to act as the Subscription Manager for the auction service. The process of doing this is surprisingly easy because WSRF.NET already defines all the needed functionality.

Also shown in Code Example 18 above that we are overriding a method from our base class called *portInit*. This method exists to allow new port type designers to write code that is called every time the service class is constructed. When a new instance of a WSRF.NET service class is constructed by WSRF.NET, all the port type constructors are run and then all the *portInit* methods of the port types (if any) are run. The *portInit* method is useful when the setup of one port type relies on another port type already having been constructed. In other words, it is sometimes useful to do additional initialization after all the constructors for all a service's port types have been run. In this case, we need the **ServiceBase** member to be valid before we can modify its event queue. This requires all the service's port type constructors to have finished running, and so the *portInit* method is used.

To create this new service, add a C# Web Service project item to your **AuctionServices** project and call the new item **SubscriptionManager.asmx**. Open the source code for this new web service and modify it, similar to the way you did before for the **Auction** service, by adding the appropriate namespaces, port types, and C# attributes to the class as shown below in Code Example 19. Finally, you once again need to set the WSRF property to true on the **SubscriptionManager.asmx** item inside of the VS.NET (see Figure 2).

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.Notification;

namespace AuctionServices
{
    [WsdlBaseName("SubscriptionManager",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(SubscriptionManagerPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    public class SubscriptionManager : ServiceSkeleton
    {
        public SubscriptionManager()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        ...
        #endregion
    }
}

```

#### Code Example 19: Creating the Subscription Manager

Next, you need to configure the **Auction** service (which is now a Notification Producer in addition to the other port types you have added) to use the newly created Subscription Manager service. This is done by adding configuration elements to the `web.config` file for the **AuctionServices** project. Specifically you must add a new configuration section element for the WSRF.NET specific configurations and within that, a new configuration block for the Notification Producer. These changes are shown in Code Example 20 below. Note once again that any strings enclosed in double quotes should not contain carriage returns, though for readability, we have shown them that way in this document.

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <configSections>
    <section name="microsoft.web.services3"
type="Microsoft.Web.Services3.Configuration.WebServicesConfiguration,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>

    <section name="wsrf-config"
type="UVA.GCG.WSRF.Service.Configuration.WSRFServiceConfigurationHand
ler, UVA.GCG.WSRF.Service"/>
  </configSections>

  <wsrf-config>
    <services>
      <service name="Auction.asmx">
        <parameter name="SubscriptionManagerURL"
value="%transport-protocol%://%machine-name%/%service-base-
path%/SubscriptionManager.asmx"/>
      </service>
    </services>
  </wsrf-config>

  <system.web>

    <webServices>
      <soapExtensionImporterTypes>
        <add
type="Microsoft.Web.Services3.Description.WseExtensionImporter,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
      </soapExtensionImporterTypes>
      <soapServerProtocolFactory
type="Microsoft.Web.Services3.WseProtocolFactory,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
    </webServices>

    ...

```

**Code Example 20: Adding Configuration Elements for Notification Producers**

For the Non-Brokered form of notification in WSRF.NET, only the SubscriptionManagerURL parameter is needed. This configuration component tells the Notification Producer the URL of a Subscription Manager that it should use to handle all subscriptions for that producer. Notice that the value of the SubscriptionManagerURL parameter in the configuration file above contains a number of strings surrounded by percent (%) characters. These are WSRF.NET macros that make it easier for users to specify configuration parameters in the `web.config` file. In particular, this line of text is indicating that the URL of the subscription manager is given by re-using all of the components of the URL which make up the Notification Producer's address. This is with the exception that the last component of the address is replaced with the string `SubscriptionManager.asmx`. In other words, if the URL for the **Auction** service were <http://localhost/some-path/Auction.asmx>, then the URL of the associated Subscription Manager would be <http://localhost/some-path/SubscriptionManager.asmx>. This method of specifying the Subscription Manager's URL allows for a large degree of freedom in how the Notification Producer is configured.

## Modifying the Client to Subscribe to the Producer

Now that we have the service-side of our example finished, it's time to concentrate on the client-side changes. In this section of the tutorial, we will cover the necessary steps to subscribe the client to the Notification Producer on the *new-bid* topic. Since the console application is not currently running any kind of listening thread, we will need to add a server thread which will then receive the asynchronous notification messages. Fortunately, doing so is easy thanks to WSRF.NET's **AsynchronousNotificationListener** class.

Code Example 21 shows the modifications that I made to the console application which prepare it to receive notifications. This new code does not yet register the client application with the producer to receive notifications (we will add the subscription code shortly), but merely starts a listener thread so that when we subscribe, there is a server thread ready to receive notification messages.

```

using AuctionCommon;
using System;
using System.Xml;
using UVa.GCG.WSRF.Common.HttpServer;
using UVa.GCG.WSRF.Common.WS;
using UVa.GCG.WSRF.Common.WS.Addressing;
using UVa.GCG.WSRF.Common.WS.Grid;
using UVa.GCG.WSRF.Common.WS.ResourceLifetime;
using UVa.GCG.WSRF.Common.WS.ResourceProperties;
using UVa.GCG.WSRF.Common.WS.Notification;
using UVa.GCG.WSRF.Common.WS.Notification.Topics;

namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            TopicExpression bidTopic =
                WellknownDialects.SIMPLE.createExpression(
                    new XmlQualifiedName("new-bid",
                        "http://wsrfnet.cs.virginia.edu/auction-tutorial"));
            bidTopic.addHandler(new TopicExpressionListener(
                notifyCallback));

            AsynchronousNotificationListener listener =
                new AsynchronousNotificationListener(5432);
            listener.registerExpression(bidTopic);
            listener.start();

            EndpointReferenceType auction = createAuction(
                "http://localhost/AuctionServices/Auction.asmx");

            AuctionDriver.Auction.AuctionServiceWse bidProxy =
                new AuctionDriver.Auction.AuctionServiceWse();
            WSUtilities.setEPR(bidProxy, auction);

            getProperties(auction);

            makeBid(bidProxy, 5);
            makeBid(bidProxy, 1);
            makeBid(bidProxy, 10);

            // Wait for notifications to come in
            System.Threading.Thread.Sleep(1000 * 5);

            destroyEndpoint(auction);

            listener.stop();
        }

        static private void notifyCallback(
            Topic topic,
            NotificationMessageHolderType message)
        {
            Console.WriteLine(
                "Received notification that the bid has changed to {0}.",
                message.Message.InnerText);
        }
    }
}

```

**Code Example 21: Receiving Notifications**

The final step in this section is to add code to the client which performs the act of subscribing it to the **Auction** service on the *new-bid* topic. Notice that as per the *WS-Notification* specification, the act of subscribing creates a subscription Resource whose *EndpointReference* is returned by the *createSubscription* function. We then clean up this *WS-Resource* at the end of our application execution. The changes are shown in Code Example 22.

```

...
namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            TopicExpression bidTopic =
                WellknownDialects.SIMPLE.createExpression(
                    new XmlQualifiedName("new-bid",
                        "http://wsrfnet.cs.virginia.edu/auction-tutorial"));
            bidTopic.addHandler(new TopicExpressionListener(
                notifyCallback));

            AsynchronousNotificationListener listener =
                new AsynchronousNotificationListener(5432);
            listener.registerExpression(bidTopic);
            listener.start();

            EndpointReferenceType auction = createAuction(
                "http://localhost/AuctionServices/Auction.asmx");
            EndpointReferenceType consoleSubscription =
                createSubscription(
                    bidTopic, auction, new EndpointReferenceType(
                        new AttributedURI(null, "http://localhost:5432"),
                        WSUtilities.createReferencePropertiesType(
                            Guid.NewGuid().ToString(),
                            null, null, null, null));

            AuctionDriver.Auction.AuctionServiceWse bidProxy =
                new AuctionDriver.Auction.AuctionServiceWse();
            WSUtilities.setEPR(bidProxy, auction);

            getProperties(auction);

            makeBid(bidProxy, 5);
            makeBid(bidProxy, 1);
            makeBid(bidProxy, 10);

            // Wait for notifications to come in
            System.Threading.Thread.Sleep(1000 * 5);

            destroyEndpoint(auction);
            destroyEndpoint(consoleSubscription);

            listener.stop();
        }

        static private EndpointReferenceType createSubscription(
            TopicExpression bidTopic,
            EndpointReferenceType producer,
            EndpointReferenceType consumer)
        {
            TopicExpressionType topic = bidTopic.TopicExpressionType;

            NotificationProducerProxy proxy =
                new NotificationProducerProxy(producer);
            return proxy.Subscribe(
                new SubscribeRequest(consumer,
                    topic)).SubscriptionReference;
        }

        ...
    }
}

```

Code Example 22: Subscribing the Consumer to the Producer

## Using a Service as a Consumer

Using a web service as a Notification Consumer is also a relatively painless procedure in WSRF.NET. Provided in the library is a port type that already takes care of most of the hard parts of setting up a consumer. The only thing left to the programmer is to plug his or her code to process incoming notifications into a .NET event that is raised by the consumer port type whenever a notification message is received. In this section we will do just that.

First, we need to create a new web service capable of accepting incoming notification messages. Once more, add a new project item (of type C# Web Service) to your **AuctionServices** project. Call this one **Consumer.asmx**. Next open the code for viewing and set the service up as a new WSRF.NET web service with an aggregated port type of **NotificationConsumer**<sup>7</sup>. This is shown in Code Example 23. Once again, you will need to change the properties for this project item so that it is considered a WSRF service (again, by setting the WSRF property on the project item to **true**). Once that is done, build the project and the new service will have the functionality of the added port types.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;
using UVa.GCG.WSRF.Service.WS.Notification;

namespace AuctionServices
{
    [Wsd1BaseName("Consumer",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(NotificationConsumerPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    public class Consumer : ServiceSkeleton
    {
        public Consumer()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        ...
        #endregion
    }
}
```

**Code Example 23: Creating the Consumer Service**

---

<sup>7</sup> We actually add two additional port types (GCGResourceFactoryPortType and ImmediateResourceTerminationPortType) as a convenience to give us better control of this consumer's Resource Lifetime management.

Now that the consumer service has been created, we will set it up to process notification messages it receives. In general, a service author can implement arbitrary processing of these messages. For the sake of simplicity, this tutorial logs the message to a file.

Again we use the **ServiceBase** member of **ServiceSkeleton** and we set up its .NET event queue. The service author registers a .NET delegate to be called when the notification event is raised. When the service receives a notification message, it will fire that event and our delegate will be called with the message as one of its parameters. This can all be seen in Code Example 24.

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.IO;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;
using UVa.GCG.WSRF.Common.WS.Notification;
using UVa.GCG.WSRF.Common.WS.Notification.Topics;

namespace AuctionServices
{
    [WsdBaseName("Consumer",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(NotificationConsumerPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    public class Consumer : ServiceSkeleton
    {
        public Consumer()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        ...
        #endregion

        protected override void portInit()
        {
            TopicExpression bidTopic =
                WellknownDialects.SIMPLE.createExpression(
                    new XmlQualifiedName("new-bid",
                        "http://wsrfnet.cs.virginia.edu/auction-tutorial"));
            bidTopic.addHandler(new TopicExpressionListener(
                NotificationQueue_OnNotification));

            ServiceBase.TopicExpressions.registerExpression(bidTopic);
        }

        private void NotificationQueue_OnNotification(Topic topic,
            NotificationMessageHolderType msg)
        {
            StreamWriter writer = null;
            try
            {
                writer = new StreamWriter(@"C:\ASPNET\log.txt", true);
                writer.WriteLine("Received a new bid of {0}",
                    msg.Message.InnerText);
            }
            finally
            {
                if (writer != null)
                    writer.Close();
            }
        }
    }
}

```

Code Example 24: Using Notifications from a Consumer Service

In the above example we write our output file to the path `C:\ASPNET\log.txt`. In order for this to correctly work, the user account under which IIS runs must have write permission on the given directory (and the directory must exist). On most windows boxes, this account by default will be the *machine-name*\ASPNET user account. However, on Windows 2003 Server machines, the account name is instead **NT AUTHORITY\NETWORK SERVICE\ASPNET**. In either case, you must explicitly change the security settings for that directory so that the appropriate user has write permission.

Finally, we need to modify the client code so that it creates and subscribes the new consumer service to the producer on the correct topic. The code for doing this is identical in concept to the code we have written before (for creating the Auction, and for subscribing the client to it). It is shown in Code Example 25 below.

```

...
[STAThread]
static void Main(string[] args)
{
    ...

    listener.start();

    EndpointReferenceType auction = createAuction(
        "http://localhost/AuctionServices/Auction.asmx");
    EndpointReferenceType consumer = createConsumer(
        "http://localhost/AuctionServices/Consumer.asmx");
    EndpointReferenceType consoleSubscription = createSubscription(
        bidTopic, auction, new EndpointReferenceType(
            new AttributedURI(null, "http://localhost:5432"),
            WSUtilities.createReferencePropertiesType(
                Guid.NewGuid().ToString()),
            null, null, null));
    EndpointReferenceType serviceSubscription = createSubscription(
        bidTopic, auction, consumer);

    AuctionDriver.Auction.AuctionServiceWse bidProxy =
        new AuctionDriver.Auction.AuctionServiceWse();
    WSUtilities.setEPR(bidProxy, auction);

    getProperties(auction);

    makeBid(bidProxy, 5);
    makeBid(bidProxy, 1);
    makeBid(bidProxy, 10);

    // Wait for notifications to come in
    System.Threading.Thread.Sleep(1000 * 5);

    destroyEndpoint(auction);
    destroyEndpoint(consoleSubscription);
    destroyEndpoint(serviceSubscription);
    destroyEndpoint(consumer);

    listener.stop();
}

static private EndpointReferenceType createConsumer(
    string serviceURL)
{
    GCGResourceFactoryBinding proxy =
        new GCGResourceFactoryBinding(serviceURL);
    Create creationParms = new Create();

    CreateResponse response =
        proxy.Create(creationParms);

    return response.ResourceEndpoint;
}
...

```

**Code Example 25: Subscribing the Consumer Service**

This concludes this section on using notification in WSRF.NET applications. At this point, you should be able to completely run your example and find the notifications being received at the correct locations.

## Chapter 6: Using Service Groups

In this last chapter of the tutorial, we will briefly touch on another of the more advanced topics that WSRF.NET has to offer – that of Service Groups. Service Groups in WSRF are a mechanism by which multiple *WS-Resources* can be grouped together into a logical collection. Those groups can then be queried with various XPath statements to discover members (or information about members) of the group.

Now that the **Auction** service is up and running, we step back and consider a broader view of our auction grid application. It makes sense to allow users to search on the auctions currently ongoing and to then select from among those auctions ones in which they are interested in bidding. One way to do this is by using Service Groups to make a collection of all the active auctions and then allow users to search those auctions. In the following sections of this tutorial, we'll do just that.

### Creating the Service Group and Adding Members

The first step in creating our Service Group is to create a new auction manager service and add the **ServiceGroup** port type using the [**WSRFPortType**] attribute. We will also add the **GCGResourceFactory** port type as well as the **ImmediateResourceTermination** port type to make the job of managing the lifetime of this Resource easier. Finally, in order to query the Service Group for its members, we need to have access to its member's Resource Properties so we add the **QueryResourceProperties** port type as well. Create a new C# Web Service item in your **AuctionServices** project and call it [AuctionManager.asmx](#) (the code for [AuctionManager.asmx](#) is shown in Code Example 26). As before, you must make sure that the WSRF property on the new project item is set to true.

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVA.GCG.WSRF.Common.Attributes;
using UVA.GCG.WSRF.Service.BaseTypes;
using UVA.GCG.WSRF.Service.Grid;
using UVA.GCG.WSRF.Service.ResourceLifetime;
using UVA.GCG.WSRF.Service.ResourceProperties;
using UVA.GCG.WSRF.Service.ServiceGroup;

namespace AuctionServices
{
    [WsdBaseName("AuctionManager",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ServiceGroupRegistrationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(QueryResourcePropertiesPortType))]
    public class AuctionManager : ServiceSkeleton
    {
        public AuctionManager()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        ...
        #endregion
    }
}

```

**Code Example 26: The AuctionManager Service**

Just as the Notification Producer service used a Subscription Manager service to manage information it needed, services implementing the Service Group port type use an additional service to manage the references to the Service Group members. This service must have a ServiceGroupEntry port type and for WSRF.NET it is also required to have both GCGResourceFactory port type as well as ImmediateResourceTermination port type. Once again, create a new project item which is a C# Web Service (don't forget to mark the WSRF property for this item as true) and call the new item AuctionManagerEntry.asmx. See Code Example 27 below for the code.

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;
using UVa.GCG.WSRF.Service.ServiceGroup;

namespace AuctionServices
{
    [WsdlBaseName("AuctionManagerEntry",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ServiceGroupEntryPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    public class AuctionManagerEntry : ServiceSkeleton
    {
        public AuctionManagerEntry()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        ...
        #endregion
    }
}

```

#### Code Example 27: AuctionManagerEntry Service

The last service-side change that you have to make in order to start adding auctions to your new manager is to configure the manager so that it can locate the Service Group Entry port type. This is very similar to the configuration changes that you made to the [web.config](#) file before in order to set the Subscription Manager URL for the Notification Producer. Once again, open the [web.config](#) file and edit the file as shown in Code Example 28.

```

...
<wsrf-config>
  <services>
    <service name="Auction.asmx">
      <parameter name="SubscriptionManagerURL"
value="%transport-protocol://%machine-name%/%service-base-
path%/SubscriptionManager.asmx"/>
    </service>
    <service name="AuctionManager.asmx">
      <parameter name="ServiceGroupEntryURL" value="%transport-
protocol://%machine-name%/%service-base-
path%/AuctionManagerEntry.asmx"/>
    </service>
  </services>
</wsrf-config>
</wsrf-config>
...

```

#### Code Example 28: Configuring the ServiceGroup Port Type

## Modifying the Client

Modifying the client to use the new **AuctionManager** service is somewhat more involved than setting up the services. While the code is not complex, there is a lot of it and so we will take things in small pieces and build up the example slowly.

The first step is to have our client create the new **AuctionManager** *WS-Resource*. In practice, there would be a pre-existing AuctionManager Resource that clients could query, but for the propose of this tutorial, we will create one ourselves. The creation of this *WS-Resource* is handled exactly as the creations of the other Resources have been – via the GCGResourceFactory port type *create* method. This time however when we create the *WS-Resource*, we will pass in initialization parameters of a type that is part of the WSRF.NET library. These initialization values tell the Service Group what the “rules” are for being a member of that group. These rules take the form of XmlQualifiedNames that identify the set of XmlElements which must accompany the addition of each new entry in the group. Each XmlElement must have the name of one of the rules given and its value will be the content for that membership rule. Later, we will use these rules to search the auctions that are available. Code Example 29 below shows this first change to our client-side code.

```

...
using Uva.GCG.WSRF.Common.WS.ServiceGroup;

namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            ...

            listener.start();

            EndpointReferenceType manager = createManager(
                "http://localhost/AuctionServices/AuctionManager.asmx");

            EndpointReferenceType auction = createAuction(
                "http://localhost/AuctionServices/Auction.asmx");

            ...

            destroyEndpoint(auction);
            destroyEndpoint(consoleSubscription);
            destroyEndpoint(serviceSubscription);
            destroyEndpoint(consumer);
            destroyEndpoint(manager);

            listener.stop();
        }

        static private EndpointReferenceType createManager(
            string url)
        {
            GCGResourceFactoryBinding proxy =
                new GCGResourceFactoryBinding(url);
            ServiceGroupPortTypeInit sgInit =
                new ServiceGroupPortTypeInit();
            MembershipContentRule rule = new MembershipContentRule();
            rule.ContentElements = new XmlQualifiedName[]
            {
                new XmlQualifiedName("AuctionDescription",
                    "http://wsrfnet.cs.virginia.edu/auction-tutorial")
            };
            sgInit.Rules.Add(rule);
            Create create = new Create();
            create.PortTypeInitializers = new XmlElement[]
            {
                WSUtilities.Serialize(sgInit)
            };

            return proxy.Create(create).ResourceEndpoint;
        }
    }
}
...

```

**Code Example 29: Creating the AuctionManager Service**

Now that we have created the **AuctionManager** service Resource, we need to start adding WS-Resources to its Service Group. Again, in practice, the AuctionManager would already contain Resources that the client could query for, but here we simplify matters by having the client create the Resource and add some WS-Resources to it. Adding Resources to the Service Group is easy and simply requires that our client add EPRs for auctions to the manager along with the membership

content elements that we choose to associate with each auction. Each membership content element is an arbitrary Xml element that the Service Group will associate with the entry. These membership content elements can later be used to search for or query specific elements. In the example below, we add a membership content element which contains the description of the auction. Later, we will write client code that can look through these content elements and one could assume in a real application, a human could then pick the auction that he or she wished to participate in.

```

...
static void Main(string[] args)
{
    ...

    listener.start();

    EndpointReferenceType manager = createManager(
        "http://localhost/AuctionServices/AuctionManager.asmx");

    EndpointReferenceType auction = createAuction(
        manager,
        "http://localhost/AuctionServices/Auction.asmx",
        "2004 Kawasaki Ninja", "Mark Morgan");
    EndpointReferenceType auction2 = createAuction(
        manager,
        "http://localhost/AuctionServices/Auction.asmx",
        "Slightly used BIC Pen", "Don Joe");
    EndpointReferenceType auction3 = createAuction(
        manager,
        "http://localhost/AuctionServices/Auction.asmx",
        "Large Mansion in Beverley Hills", "Jane Doe");

    ...

static private EndpointReferenceType createAuction(
    EndpointReferenceType manager,
    string serviceURL, string description, string seller)
{
    AuctionConstructionParameters cons =
        new AuctionConstructionParameters(
            description, seller);

    GCGResourceFactoryBinding proxy =
        new GCGResourceFactoryBinding(serviceURL);
    Create creationParms = new Create();

    creationParms.PortTypeInitializers = new XmlElement[]
    {
        { WSUtilities.Serialize(cons)
        };

    CreateResponse response =
        proxy.Create(creationParms);

    return response.ResourceEndpoint;
}

```

### Code Example 30: Setting Up to Add to the Manager

At this point, we have modified the client to create the auction manager and then to create a number of auction instances. Next we will see how to add those instances to the manager as Service Group entries. This step is shown in Code Example 31 below.

```

...
static private EndpointReferenceType createAuction(
    EndpointReferenceType manager,
    string serviceURL, string description, string seller)
{
    AuctionConstructionParameters cons =
        new AuctionConstructionParameters(
            description, seller);

    GCGResourceFactoryBinding proxy =
        new GCGResourceFactoryBinding(serviceURL);
    Create creationParms = new Create();

    creationParms.PortTypeInitializers = new XmlElement[]
    {
        WSUtilities.Serialize(cons)
    };

    CreateResponse response =
        proxy.Create(creationParms);

    AddType add = new AddType();
    XmlDocument doc = new XmlDocument();
    doc.AppendChild(doc.CreateElement("AuctionDescription",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial"));
    doc.DocumentElement.InnerText = description;
    add.Content = new XmlElement[]
    {
        doc.DocumentElement
    };
    add.MemberEPR = response.ResourceEndpoint;
    add.InitialTerminationTimeSpecified = false;
    ServiceGroupRegistrationBinding sg =
        new ServiceGroupRegistrationBinding(manager);
    sg.Add(add);

    return response.ResourceEndpoint;
}
...

```

**Code Example 31: Adding Entries to a Service Group**

The final step in modifying our Auction client is to change the client so that it picks the auction to make bids on by asking the Auction manager for a list of current auctions and then selecting one. For this tutorial, we'll select one randomly. We will also need to change the way that the auctions are destroyed at the end. These changes and the entire code for all services and clients created in this tutorial is included in the tutorial appendix. By now you will have had a good introduction to building services and clients using WSRF.NET. This document was not able to cover every nuance and detail of the WSRF.NET system or WSRF specifications and so interested readers should see the WSRF.NET Programmer's Reference manual for more details. Hopefully, this tutorial has provided you with a solid foundation on which to grow.

## Appendix A: Final Code Listings

### AuctionConstructionParameters.cs

```
using System;
using System.Xml.Serialization;

namespace AuctionCommon
{
    [XmlRoot("Auction",
        Namespace="http://wsrfnet.cs.virginia.edu/auction-
tutorial")]
    [XmlType("Auction",
        Namespace="http://wsrfnet.cs.virginia.edu/auction-
tutorial")]
    public class AuctionConstructionParameters
    {
        private string _description;
        private string _owner;

        public string Description
        {
            get { return _description; }
            set { _description = value; }
        }

        public string Owner
        {
            get { return _owner; }
            set { _owner = value; }
        }

        public AuctionConstructionParameters(
            string description, string owner)
        {
            _description = description;
            _owner = owner;
        }

        public AuctionConstructionParameters() : this (null,
null)
        {
        }
    }
}
```

## BidderInformation.cs

```
using System;

namespace AuctionServices
{
    [Serializable]
    public class BidderInformation
    {
        private string _name;
        private string _email;

        public string Name
        {
            get { return _name; }

            set { _name = value; }
        }

        public string Email
        {
            get { return _email; }

            set { _email = value; }
        }

        public BidderInformation(string name, string email)
        {
            _name = name;
            _email = email;
        }

        public BidderInformation() : this(null, null)
        {
        }
    }
}
```

## Auction.asmx.cs

```
using AuctionCommon;
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml;
using System.Xml.Serialization;
using UVA.GCG.WSRF.Common.Attributes;
using UVA.GCG.WSRF.Service.BaseTypes;
using UVA.GCG.WSRF.Service.Grid;
using UVA.GCG.WSRF.Service.Notification;
using UVA.GCG.WSRF.Service.ResourceLifetime;
using UVA.GCG.WSRF.Service.ResourceProperties;

namespace AuctionServices
{
    [WsdlBaseName("Auction",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(GetResourcePropertyPortType))]
    [WSRFPortType(typeof(NotificationProducerPortType))]
    [ResourceInitializerType(typeof(AuctionConstructionParameters))]
}

public class Service1 : ServiceSkeleton
{
    [Resource]
    [ResourceProperty(Name="CurrentBid")]
    private int _currentBid;

    [Resource]
    private BidderInformation _currentBidder;

    [Resource]
    [ResourceProperty]
    private string AuctionDescription;

    [Resource]
    [ResourceProperty]
    private string AuctionOwner;

    public Service1()
    {
        InitializeComponent();
    }

    public override void InitResource(Hashtable parms)
    {
        AuctionConstructionParameters cons =
            parms[this.GetType().FullName] as
            AuctionConstructionParameters;
    }
}
```

```

        _currentBid = 0;
        _currentBidder = null;

        AuctionDescription = cons.Description;
        AuctionOwner = cons.Owner;
    }

    protected override void portInit()
    {
        ServiceBase.Topics.addTopicHierarchy(
            new XmlQualifiedName("new-bid",
"http://wsrfnet.cs.virginia.edu/auction-tutorial"),
            true);
    }

    [WebMethod]

    [SoapDocumentMethod("http://wsrfnet.cs.virginia.edu/auction-
tutorial/bid")]
    [return: XmlElement("bidSuccessful",
        Namespace="http://wsrfnet.cs.virginia.edu/auction-
tutorial")]
    public bool bid(BidderInformation bidder, int newBid)
    {
        if (newBid <= _currentBid)
            return false;

        _currentBid = newBid;
        _currentBidder = bidder;

        ServiceBase.Topics[
            new XmlQualifiedName("new-bid",
"http://wsrfnet.cs.virginia.edu/auction-
tutorial")].notify(newBid);

        return true;
    }

    #region Component Designer generated code

    //Required by the Web Services Designer
    private IContainer components = null;

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if(disposing && components != null)
        {

```

```
        components.Dispose();
    }
    base.Dispose(disposing);
}
#endregion
}
```

## SubscriptionManager.asmx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.Notification;

namespace AuctionServices
{
    [WsdName("SubscriptionManager",
"http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(SubscriptionManagerPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    public class SubscriptionManager : ServiceSkeleton
    {
        public SubscriptionManager()
        {
            //CODEGEN: This call is required by the ASP.NET Web
Services Designer
            InitializeComponent();
        }

        #region Component Designer generated code

        //Required by the Web Services Designer
        private IContainer components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if(disposing && components != null)
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #endregion
    }
}
```

## Consumer.asmx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.IO;
using System.Web;
using System.Web.Services;
using System.Xml;
using UVa.GCG.WSRF.Common.Attributes;
using UVa.GCG.WSRF.Common.WS.Notification;
using UVa.GCG.WSRF.Common.WS.Notification.Topics;
using UVa.GCG.WSRF.Service.BaseTypes;
using UVa.GCG.WSRF.Service.Grid;
using UVa.GCG.WSRF.Service.ResourceLifetime;
using UVa.GCG.WSRF.Service.Notification;
using UVa.GCG.WSRF.Service.Notification.Topics;

namespace AuctionServices
{
    /// <summary>
    /// Summary description for Consumer.
    /// </summary>
    [WsdlBaseName("Consumer",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(NotificationConsumerPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    public class Consumer : ServiceSkeleton
    {
        public Consumer()
        {
            //CODEGEN: This call is required by the ASP.NET Web
            Services Designer
            InitializeComponent();
        }

        protected override void portInit()
        {
            TopicExpression bidTopic =
                WellknownDialects.SIMPLE.createExpression(
                    new XmlQualifiedName("new-bid",
                        "http://wsrfnet.cs.virginia.edu/auction-tutorial"));
            bidTopic.addHandler(new TopicExpressionListener(
                NotificationQueue_OnNotification));

            ServiceBase.TopicExpressions.registerExpression(bidTopic);
        }

        private void NotificationQueue_OnNotification(Topic
topic,
            NotificationMessageHolderType msg)
        {
            StreamWriter writer = null;

```

```

        try
        {
            writer = new
StreamWriter(@"C:\ASPNET\log.txt", true);
            writer.WriteLine("Received a new bid of {0}",
                msg.Message.InnerText);
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
    }

    #region Component Designer generated code

    //Required by the Web Services Designer
    private IContainer components = null;

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if(disposing && components != null)
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #endregion
}
}
}

```

## AuctionManager.asmx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVA.GCG.WSRF.Common.Attributes;
using UVA.GCG.WSRF.Service.BaseTypes;
using UVA.GCG.WSRF.Service.Grid;
using UVA.GCG.WSRF.Service.ResourceLifetime;
using UVA.GCG.WSRF.Service.ResourceProperties;
using UVA.GCG.WSRF.Service.ServiceGroup;

namespace AuctionServices
{
    [WsdlBaseName("AuctionManager",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ServiceGroupRegistrationPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    [WSRFPortType(typeof(QueryResourcePropertiesPortType))]
    public class AuctionManager : ServiceSkeleton
    {
        public AuctionManager()
        {
            InitializeComponent();
        }

        #region Component Designer generated code
        //Required by the Web Services Designer
        private IContainer components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if(disposing && components != null)
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
        #endregion
    }
}
```

## AuctionManagerEntry.asmx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using UVA.GCG.WSRF.Common.Attributes;
using UVA.GCG.WSRF.Service.BaseTypes;
using UVA.GCG.WSRF.Service.Grid;
using UVA.GCG.WSRF.Service.ResourceLifetime;
using UVA.GCG.WSRF.Service.ServiceGroup;

namespace AuctionServices
{
    [WsdllBaseName("AuctionManagerEntry",
        "http://wsrfnet.cs.virginia.edu/auction-tutorial")]
    [WebService]
    [WebServiceBinding]
    [WSRFPortType(typeof(ServiceGroupEntryPortType))]
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
    public class AuctionManagerEntry : ServiceSkeleton
    {
        public AuctionManagerEntry()
        {
            InitializeComponent();
        }

        #region Component Designer generated code

        //Required by the Web Services Designer
        private IContainer components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if(disposing && components != null)
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #endregion
    }
}
```

## Class1.cs

```
using AuctionCommon;
using System;
using System.Xml;
using System.Xml.Serialization;
using UVa.GCG.WSRF.Common.HttpServer;
using UVa.GCG.WSRF.Common.WS;
using UVa.GCG.WSRF.Common.WS.Addressing;
using UVa.GCG.WSRF.Common.WS.Grid;
using UVa.GCG.WSRF.Common.WS.ResourceLifetime;
using UVa.GCG.WSRF.Common.WS.ResourceProperties;
using UVa.GCG.WSRF.Common.WS.Notification;
using UVa.GCG.WSRF.Common.WS.Notification.Topics;
using UVa.GCG.WSRF.Common.WS.ServiceGroup;

namespace AuctionDriver
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            TopicExpression bidTopic =
                WellknownDialects.SIMPLE.createExpression(
                    new XmlQualifiedName("new-bid",
"http://wsrfnet.cs.virginia.edu/auction-tutorial"));
            bidTopic.addHandler(new TopicExpressionListener(
                notifyCallback));

            AsynchronousNotificationListener listener =
                new AsynchronousNotificationListener(5432);

            listener.registerExpression(bidTopic);

            listener.start();

            EndpointReferenceType manager = createManager(
"http://localhost/AuctionServices/AuctionManager.asmx");

            createAuction(
                manager,
"http://localhost/AuctionServices/Auction.asmx",
                "2004 Kawasaki Ninja", "Mark Morgan");
            createAuction(
                manager,
"http://localhost/AuctionServices/Auction.asmx",
                "Slightly used BIC Pen", "Don Joe");
            createAuction(
                manager,
"http://localhost/AuctionServices/Auction.asmx",
                "Large Mansion in Beverley Hills", "Jane
Doe");
        }
    }
}
```

```

        EndpointReferenceType auction =
chooseAuction(manager);

        EndpointReferenceType consumer = createConsumer(
            "http://localhost/AuctionServices/Consumer.asmx");
        EndpointReferenceType consoleSubscription =
            createSubscription(bidTopic,
                auction, new EndpointReferenceType(
                    new AttributedURI(null,
"http://localhost:5432"),
                    WSUtilities.createReferencePropertiesType(
                        Guid.NewGuid().ToString()),
                    null, null, null, null));
        EndpointReferenceType serviceSubscription =
createSubscription(
            bidTopic, auction, consumer);

        AuctionDriver.Auction.AuctionServiceWse bidProxy =
            new
AuctionDriver.Auction.AuctionServiceWse();
        WSUtilities.setEPR(bidProxy, auction);

        getProperties(auction);

        makeBid(bidProxy, 5);
        makeBid(bidProxy, 1);
        makeBid(bidProxy, 10);

        // Wait for notifications to come in
        System.Threading.Thread.Sleep(1000 * 5);

        destroyAuctions(manager);
        destroyEndpoint(consoleSubscription);
        destroyEndpoint(serviceSubscription);
        destroyEndpoint(consumer);
        destroyEndpoint(manager);

        listener.stop();
    }

    static private EndpointReferenceType createManager(
        string url)
    {
        GCGResourceFactoryBinding proxy =
            new GCGResourceFactoryBinding(url);
        ServiceGroupPortTypeInit sgInit = new
ServiceGroupPortTypeInit();
        MembershipContentRule rule = new
MembershipContentRule();
        rule.ContentElements = new XmlQualifiedName[]
        {
            new
XmlQualifiedName("AuctionDescription",
            "http://wsrfnet.cs.virginia.edu/auction-tutorial")
        };
        sgInit.Rules.Add(rule);
        Create create = new Create();
        create.PortTypeInitializers = new XmlElement[]
        {

```

```

        WSUtilities.Serialize(sgInit)
    };

    return proxy.Create(create).ResourceEndpoint;
}

static private EndpointReferenceType createConsumer(
    string serviceURL)
{
    GCGResourceFactoryBinding proxy =
        new GCGResourceFactoryBinding(serviceURL);
    Create creationParms = new Create();

    CreateResponse response =
        proxy.Create(creationParms);

    return response.ResourceEndpoint;
}

static private EndpointReferenceType createSubscription(
    TopicExpression bidTopic,
    EndpointReferenceType producer,
    EndpointReferenceType consumer)
{
    TopicExpressionType topic =
bidTopic.TopicExpressionType;

    NotificationProducerProxy proxy =
        new NotificationProducerProxy(producer);
    return proxy.Subscribe(
        new SubscribeRequest(consumer,
            topic)).SubscriptionReference;
}

static private void notifyCallback(
    Topic topic, NotificationMessageHolderType message)
{
    Console.WriteLine(
changed to {0}.",
        message.Message.InnerText);
}

static private void getProperties(
    EndpointReferenceType auction)
{
    GetResourcePropertyBinding proxy =
        new GetResourcePropertyBinding(auction);

    XmlElement val = proxy.GetResourceProperty(
        new XmlQualifiedName(
            "AuctionDescription",
            "http://wsrfnet.cs.virginia.edu/auction-
tutorial")).Any[0]
        as XmlElement;
    Console.WriteLine("The auction description is
\"{0}\".",
        val.InnerText);
}

static private void makeBid(

```

```

        AuctionDriver.Auction.AuctionServiceWse bidProxy,
        int newBid)
    {
        AuctionDriver.Auction.BidderInformation info =
            new
AuctionDriver.Auction.BidderInformation();
        info.Name = "Mark Morgan";
        info.Email = "mmm2a@cs.virginia.edu";

        Auction.bid bid = new Auction.bid();
        bid.bidder = info;
        bid.newBid = newBid;
        if (bidProxy.bid(bid).bidSuccessful)
            Console.WriteLine("Bid of {0} succeeded.",
newBid);
        else
            Console.WriteLine("Bid of {0} failed!",
newBid);
    }

    static private EndpointReferenceType createAuction(
        EndpointReferenceType manager,
        string serviceURL, string description, string
seller)
    {
        AuctionConstructionParameters cons =
            new AuctionConstructionParameters(
                description, seller);

        GCResourceFactoryBinding proxy =
            new GCResourceFactoryBinding(serviceURL);
        Create creationParms = new Create();

        creationParms.PortTypeInitializers = new
XmlElement[]
        {
            {
                WSUtilities.Serialize(cons)
            }
        };

        CreateResponse response =
            proxy.Create(creationParms);

        AddType add = new AddType();
        XmlDocument doc = new XmlDocument();

        doc.AppendChild(doc.CreateElement("AuctionDescription",
            "http://wsrfnet.cs.virginia.edu/auction-
tutorial"));

        doc.DocumentElement.InnerText = description;
        add.Content = new XmlElement[]
        {
            {
                doc.DocumentElement
            }
        };
        add.MemberEPR = response.ResourceEndpoint;
        add.InitialTerminationTimeSpecified = false;
        ServiceGroupRegistrationBinding sg = new
ServiceGroupRegistrationBinding(manager);
        sg.Add(add);

        return response.ResourceEndpoint;
    }
}

```

```

        static private void destroyEndpoint(EndpointReferenceType
epr)
    {
        ImmediateResourceTerminationProxy proxy =
            new ImmediateResourceTerminationProxy(epr);
        proxy.Destroy(new Destroy());
    }

    static private EntryType[]
retrieveEntries(EndpointReferenceType manager)
    {
        int lcv;
        XmlRootAttribute rootAttr = new
XmlRootAttribute("Entry");
        rootAttr.Namespace =
WSConstants.WS_SERVICE_GROUPS_XSD_NS;

        QueryResourcePropertiesRequest query = new
QueryResourcePropertiesRequest();
        query.QueryExpression = new QueryExpressionType();
        query.QueryExpression.Dialect =
"http://www.w3.org/TR/1999/REC-xpath-19991116";

        string xpathQuery = string.Format("/*/*[namespace-
uri()='{0}' and local-name()='Entry' "
            + "and *[namespace-uri()='{0}' and " +
            "local-name()='Content' and *[namespace-
uri()='{1}' and local-name()='AuctionDescription']]]",
            WSConstants.WS_SERVICE_GROUPS_XSD_NS,
            "http://wsrfnet.cs.virginia.edu/auction-tutorial");
        XmlDocument doc = new XmlDocument();
        query.QueryExpression.Any = new XmlNode[] {
doc.CreateTextNode(xpathQuery) };

        QueryResourcePropertiesBinding tsgService = new
QueryResourcePropertiesBinding(manager);
        QueryResourcePropertiesResponse response =
tsgService.QueryResourceProperties(query);
        EntryType []entries = new
EntryType[response.Any.Length];
        for (lcv = 0; lcv < response.Any.Length; lcv++)
        {
            entries[lcv] =
(EntryType)WSUtilities.Deserialize(response.Any[lcv],
                typeof(EntryType), rootAttr);
            Console.WriteLine("Found an Entry: {0}",
entries[lcv].Content.InnerText);
        }

        return entries;
    }

    static private EndpointReferenceType
chooseAuction(EndpointReferenceType manager)
    {
        int index;
        EntryType []entries = retrieveEntries(manager);

        Random r = new Random();
        index = r.Next(entries.Length - 1);
    }

```

```

        Console.WriteLine("Choosing \"{0}\".",
entries[index].Content.InnerText);

        return entries[index].MemberServiceEPR;
    }

    static private void destroyAuctions(EndpointReferenceType
manager)
    {
        EntryType []entries = retrieveEntries(manager);

        foreach (EntryType entry in entries)
        {
            destroyEndpoint(entry.MemberServiceEPR);
        }
    }
}

```

## Appendix B: References

- [1] Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W. 2004. The WS-Resource Framework. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf>
- [2] Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Snelling, D., and Tuecke, S. 2004. From Open Grid Services Infrastructure to Web Services Resource Framework: Refactoring and Evolution. <http://www-106.ibm.com/developerworks/webservices/library/ws-resource/grogsitowsrf.html>.
- [3] Frey, J., Graham, S., Czajkowski, C., Ferguson, D., Foster, I., Leymann, F., Maguire, T., Nagaratnam, N., Nally, M., Storey, T., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., and Weerawarana, S. 2004. WS-ResourceLifetime. <http://www-106.ibm.com/developerworks/library/ws-resource/wsresourcelifetime.pdf>.
- [4] Graham, S., Czajkowski, C., Ferguson, D., Foster, I., Frey, J., Leymann, F., Maguire, T., Nagaratnam, N., Nally, M., Storey, T., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., and Weerawarana, S. 2004. WS-ResourceProperties. <http://www-106.ibm.com/developerworks/library/ws-resource/wsresourceproperties.pdf>.
- [5] Graham, S., Maguire, T., Frey, J., Nagaratnam, N., Sedukhin, I., Snelling, D., Czajkowski, K., Tuecke, S., and Vambenepe, W. 2004. WS-ServiceGroups. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-servicegroup.pdf>.
- [6] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W. and Weihl, B. 2004. Web Services Based Notification (WS-Base Notification). <ftp://www6.software.ibm.com/software/developer/library/wsnotification/WS-BaseN.pdf>.
- [7] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W. and Weihl, B. 2004. Web Services Brokered Notification (WS-BrokeredNotification). <ftp://www6.software.ibm.com/software/developer/library/wsnotification/WS-BrokeredN.pdf>.
- [8] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W. and Weihl, B. 2004. Web Services Topics (WSTopics). <ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-Topics.pdf>.
- [9] Microsoft. Web Services Enhancements (WSE). <http://msdn.microsoft.com/webservices/building/wse/default.aspx>
- [10] Tuecke, S., Czajkowski, K., Frey, J., Foster, I., Graham, S., Maguire, T., Sedukhin, I., Snelling, D., Vambenepe, W. 2004. WS-BaseFaults. <http://www-106.ibm.com/developerworks/library/wsresource/ws-basefaults.pdf>.
- [11] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T., Sandholm, T., Snelling, D. and Vanderbilt, P. 2003. Open Grid Services Infrastructure (OGSI) version 1.0. [https://forge.gridforum.org/docman2/ViewProperties.php?group\\_id=43&category\\_id=392&document\\_content\\_id=347](https://forge.gridforum.org/docman2/ViewProperties.php?group_id=43&category_id=392&document_content_id=347)
- [12] WS-Addressing. 2004. IBM/BEA/Microsoft Corporation. <http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/wsaddressing.asp>
- [13] Wasson, G. 16 August 2004. WSRF.NET Programmer's Reference. [http://www.cs.virginia.edu/~gsw2c/WSRFdotNet/WSRFdotNet\\_programmers\\_reference.pdf](http://www.cs.virginia.edu/~gsw2c/WSRFdotNet/WSRFdotNet_programmers_reference.pdf)