

WSRF.NET 3.0 Programmer's Reference

Glenn Wasson

Updated: January 13, 2006

This manual describes how to author and deploy WSRF-compliant web services on the Microsoft .NET platform. The reader is assumed to have the WSRF.NET software already installed on their Windows machine and to have a working knowledge of web services.

1. The Web Services Resource Framework (WSRF)

The WS-Resource Framework (WSRF) [3][4][5][11] is a set of specifications that describe the relationship between “stateful resources” and web services. This relationship is defined in terms of WS-Resources, an abstraction for modeling/discovering the state manipulated by web services. A WS-Resource is a “composition of a web service and a stateful resource” [1] described by an XML document (with known schema) that is associated with the web service’s port type and addressed by a WS-Addressing EndpointReference [13]. The WSRF defines functions that allow interactions with WS-Resources such as querying, lifetime management and grouping. WSRF is based on the OGSi specification [12] and can be thought of as expressing the OGSi concepts in terms that are compatible with today’s web service standards [2]. Arguably and simplistically, it is sometimes convenient when contrasting OGSi and WSRF to think of OGSi as “distributed objects that conform to many Web Services concepts (XML, SOAP, a modified version of WSDL)”, while WSRF is fundamentally “vanilla” Web Services with more explicit handling of state. One artifact of this is that OGSi did not really support interacting with these “vanilla” Web Services and instead only interacted with “Grid Services” (by definition these were OGSi-compliant); WSRF fully supports interacting with these “vanilla” Web Services (although the argument is made that client interactions with WSRF-compliant are richer).

Currently, there are 4 specifications in the WS-ResourceFramework with one more to be officially released. WS-ResourceProperties defines how WS-Resources are described by ResourceProperty (XML) documents that can be queried and modified. WS-ResourceLifetime defines mechanisms for destroying WS-Resources (there is no defined creation mechanism). WS-ServiceGroups describe how collections of services can be represented and managed. WS-BaseFaults defines a standard exception reporting format. WS-RenewableReference (unreleased) will define how a WS-Resource’s EndpointReference, which has become invalid, can be refreshed. There are also 3 WS-Notification specifications (WS-BaseNotification, WS-Topics and WS-BrokeredNotification) that, although not part of WSRF, build on it to describe asynchronous notification.

2. The WS-ResourceFramework on .NET (WSRF.NET)

WSRF.NET is an implementation of the WSRF specifications on the Microsoft .NET platform. WSRF.NET consists of a set of libraries and tools that allows web services to be transformed into WSRF-compliant web services. WSRF.NET uses the IIS/ASP.NET architecture for web services; that is WSRF.NET services are web services. To use WSRF.NET, a service author first creates a web services using VS.NET just as they would any other web service. Then the service logic is annotated with attributes that the WSRF.NET tools recognize. WSRF.NET tools transform the author’s compiled web service into a WSRF-compliant web service consistent with the meta-data given by the

service author in the attributes. The following sections describe the process of programming in WSRF.NET, how to deploy a service and other topics related to the use of WSRF.NET.

3. Programming WSRF.NET

Authoring a service in WSRF.NET consists of three steps, 1) creating the service logic, 2) annotating the logic with meta-data through attributes, and 3) using the WSRF.NET tools to create and deploy the WSRF-compliant web service.

3.1. Create Service Logic

First, the service author should create a web service using an ASP.NET web service project in Visual Studio.NET. The service must derive off `UVa.GCG.WSRF.Service.BaseTypes.ServicesSkeleton1`. All the usual .NET attributes can be used to annotate the service logic (e.g. `[WebMethod]`, `[SoapDocumentMethod]`, `[return]` etc.). In general, a WSRF.NET service will need to reference the following dlls via VS.NET's "Add Reference" (located under the "Project" menu):

- `Microsoft.Web.Services2`
- `System`
- `System.Data`
- `System.Web.Services`
- `System.XML`
- `UVa.GCG.WSRF.Common`
- `UVa.GCG.WSRF.Service`

The last two dlls can be found in the WSRF.NET bin directory.

Figure 1 shows an example service, the package tracking service, which will be used throughout this manual to illustrate concepts in WSRF.NET programming. The service logic of Figure 1 was designed with WSRF.NET in mind and we use it as a starting point to create a WSRF compliant web service. The package tracking service is used by a delivery company and its customers. When a customer drops off a package with the company, the company clerk calls `SchedulePackageDelivery()` which creates a route for the package and returns a tracking number to be given to the customer for monitoring package status. As the package moves along its route, company employees call `CheckPkgIn()` to note that the package has reached a certain waypoint. Customers use the tracking number they got when dropping the package off to find out information about the package and its current (dynamically computed) estimated delivery date. Normally, security and policy would be used to allow only authorized individuals to access these functions. WSRF.NET leverages the rich security support of WSE [9], but this is omitted here for brevity.

¹ A service is composed of one or more port types (each of which contains one or more functions) and `ServiceSkeleton` is the base class for both port types and services in WSRF.NET. While section 3.4 provides a details of how to compose port types into a service, at this point is it sufficient to know that any discussion of programming services in the document applies equally well to programming port types.

```

using UVA.GCG.WSRF.Service.BaseTypes;
public class PackageService : ServiceSkeleton
{
    [WebMethod]
    public string Sender (TrackingNum t) {
        // return the sender of the package identified by t
    }

    [WebMethod]
    public DateTime EstimatedDeliveryDate (TrackingNum t) {
        // compute location and route from t
        return ComputeDeliveryDate(location, route);
    }

    [WebMethod]
    public TrackingNum SchedulePackageDelivery (Package p,
                                                Destination d)
    {
        // create a new Package record and delivery route
        // - return TrackingNum that customer can use
    }

    [WebMethod]
    public bool CheckPkgIn(TrackingNum t, string location)
    {
        // adjust package's location in system
    }

    private DateTime ComputeDeliveryDate(PkgLocation l, PkgRoute r)
    {
        // estimate delivery date given location on route
    }
}

```

Figure 1. Original Service Logic

3.2. Attributes for Stateful Resources

A WS-Resource is a “composition of a web service and a stateful resource” [1]. In effect, a WS-Resource is an abstraction for a collection of (possibly dynamically varying) state that is manipulated by a particular web service. A WS-Resource is addressed by name using a WS-Addressing EndpointReference (EPR) in the <To> SOAP headers of a message to a web service.

WSRF.NET allows class-level data members to be declared as part of the stateful resource manipulated by a web service via the [Resource] attribute. A WSRF.NET service has one “type” of resource which is made up of all the members annotated with the [Resource] attribute on all the service’s port types (see section 3.4 for a discussion on creating and adding port types). In the remainder of this manual, we use the word Resource (with a capital ‘R’) to describe the collection of values of the data members annotated with the [Resource] attribute. A service can have many Resources, each containing its own set of values and each being named by a unique identifier. Each Resource is stored in a database under a key made up of the URL of the web service and the Resource’s unique identifier. The EPR present in the <To> header of a SOAP

message to the web service will contain both <Address> and <ReferenceProperties> components. If the <Address> matches the service's URL and the <ReferenceProperties> matches the Resource's unique identifier, the associated Resource will be loaded from the database and its values placed in the web service's [Resource]-annotated data members. When the invocation is complete, any changes made to the [Resource]-annotated members are saved back to the database under the same EPR. This allows the service author to write code that manipulates stateful resources as if they were class-level data members. The WSRF.NET architecture takes care of the database interaction.

The `InitResource()` method must be implemented by every class that derives off of `ServiceSkeleton`. `InitResource()` should initialize any data members annotated as [Resource]. When a new Resource is created, this method will be called on each of the service's port types and it should initialize the portion of the Resource declared in that port type class (see Appendix for details on creating new Resources).

In our example service, we want a Resource to consist of information about a particular package. The unique identifier for the Resource is the package's tracking number. By placing the tracking number in the SOAP headers of messages to the package service, the data about a package can be loaded into appropriate class-level data members. Figure 2 shows the package service with [Resource] attributes. Three class-level data members have been added to contain the data about the package that will be loaded from the database, `pkg`, `route` and `location`. Each of these members has the [Resource] attribute. Note that *not all* class-level data members must be part of the Resource, but only those annotated with [Resource] will be automatically persisted and restored from the database.

```

public class PackageService : ServiceSkeleton
{
    [Resource]
    Package pkg;

    [Resource]
    PkgRoute route;

    [Resource]
    PkgLocation location;

    public string Sender {
        get { return pkg.sender; }
    }

    public DateTime EstimatedDeliveryDate {
        get { return ComputeDeliveryDate(location, route); }
    }

    public void InitResource(Hashtable parameters) {
        // initialize the [Resource] members when creating
        // a new package record
    }

    [WebMethod]
    public TrackingNum SchedulePackageDelivery (Destination d)
    {
        // create a new Package record and delivery route
        // - return TrackingNum that customer can use
    }

    [WebMethod]
    public bool CheckPkgIn(string location)
    {
        // adjust package's location in system
    }

    ...
}

```

Figure 2. Package Service with [Resource] Attributes

Since any invocation on the package web service will use the “implied resource pattern” [1], we no longer need to pass the tracking number as an explicit parameter to the [WebMethod]s. In addition, we have converted the Sender() and EstimatedDeliveryDate() functions to be C# Properties on the web service class that will use the data automatically loaded as part of the Resource. We have also added the InitResource() method that is required because the service derives off ServiceSkeleton. Here we would initialize the [Resource] annotated data members when creating a new package resource – these values would then be saved in the database with the corresponding tracking number as the key.

Though not shown in Figure 2, the [Resource] attribute can also be placed on a C# Property. In this case, the C# Property must have both a “getter” and a “setter” defined. When WSRF.NET loads a Resource, it will set the Property’s value by calling the

Property's setter with the value loaded from the database. When the Property's value is to be saved into the database, the value returned by the Property's getter is used.

```
[Resource]
public int ServiceResource {
    get { // retrieve value from your own custom resource }
    set { // set value of your custom resource }
}
```

A final note: The examples above demonstrate how Resources can be collections of state variables. However, WSRF.NET also supports the Resources that are running processes. See the Appendix for a description of how WSRF.NET's process launcher operates and how services can interact with it.

3.2.1. Implementing the IResource Interface

WSRF.NET allows a service author to customize how Resources are loaded, saved and manipulated by WSRF.NET. This capability is important when designing web services to “wrap” existing resources. In other words, the IResource interface allows existing Resources to be accessed by a service without having to move those Resources into the WSRF.NET database. Any type that implements the IResource interface and is annotated with the [Resource] attribute will not be loaded/stored by the standard WSRF.NET mechanisms. Instead, functions defined by IResource will be called. The IResource interface is defined in Figure 3. An example of an IResource declaration is also shown.

```
public interface IResource {
    public void store();
    public void load();
    public object create();
    public void destroy();
    public object key { get; set; }
}

public class MyLegacyResource : IResource {}

[Resource]
MyLegacyObject lo; // WSRF.NET will treat this as an IResource
```

Figure 3. IResource Definition

NOTE: A service author implementing a class that implements IResource must also implement a default constructor for that class.

The load() and store() functions are meant to bring the state of the Resource into the data member implementing IResource and save the state of the data member back to the service author's persistent store. The load() function will be called on each web method invocation for each [Resource] annotated member implementing IResource (before the actual code of the web method is invoked). The store() function is called after the code for each web method call is completed (but before any results are returned to the client). The create() function is meant to initialize a new Resource and destroy() cleans up that Resource's state. The create() function must return an object which is a unique “key” that can be used to load the Resource. The load() function is expected to use this key (i.e. to access it using the key property) when interacting with the resource's persistent storage.

Finally, the virtual property, key, allows the key for this Resource to be retrieved or set. Note that these unique keys are stored in WSRF.NET's database. When a Resource is being loaded from the database, any data member implementing IResource will have its key property set to the value read from the database. Then each IResource's load() method will be called (so it can use that key).

WSRF.NET services must derive off ServiceSkeleton and therefore implement the InitResource() function. WSRF.NET uses this function to initialize a new instance of a service's Resource (Section 5.1 provides more details on InitResource). If a [Resource] annotated data member implements IResource, the service author may initialize that member in this function. However, if, after InitResource() has executed, a [Resource] annotated member that implements IResource is still null, the following actions are taken by WSRF.NET. First, WSRF.NET will call the IResource's default constructor. Then the IResource's create() function is called and the returned object is "set" as the key using the key property's setter function.

The IResource interface is a powerful mechanism for incorporating existing or highly customized resources into WSRF.NET. For example, it is not necessary to load all data that might be considered part of a Resource on every web method invocation. It is up to an author's implementation of the load() to determine what data needs to be loaded. In fact, an author could implement load and store as empty functions. In this case, it would be expected that the class implementing IResource would contain a set of C# Properties for accessing its state and these Properties would load/save the state dynamically as it is requested or changed. Finally, the create() function can be used to initialize a new Resource. However, all that is required is that it return a unique key for that Resource.

3.3. Attributes for Resources Properties

WSRF describes stateful resources with an XML document called the Resource Property document (RPD). This document contains the publicly exposed information about a particular Resource and is composed of elements called ResourceProperties (RPs) that are referenced by QName. The schema of the Resource Property document is included in a WSRF service's WSDL. Clients can retrieve the values of one or more ResourceProperty elements from the document or run XPath queries against the document via the WS-ResourceProperty [4] defined functions GetResourceProperty, GetMultipleResourceProperties, GetResourcePropertyDocument and QueryResourceProperties. If a particular RP in the RPD allows, clients can add, update or remove values for an RP using the SetResourceProperties function.

In WSRF.NET, RPs are declared with the [ResourceProperty] attribute. This attribute can be used in two different ways. It can be placed on a C# property (with appropriate getters and/or setters) or on a class-level data member (usually in combination with the [Resource] attribute).

In WSRF.NET, all the Resource Properties declared within all of a service's port types define the service's ResourcePropertyDocument. The schema of this document can be thought of as containing the definitions of all the types of all the C# properties or data members annotated with [ResourceProperty] attribute. There is one instance of a

ResourcePropertyDocument for every Resource which contains the unique property values for that Resource.

Choosing whether to declare ResourceProperties using C# properties or data members depends on several factors. In general, we believe that placing the [ResourceProperty] attribute on C# properties will handle most needs. ResourceProperties typically represent some transform on the state held in a Resource. C# Properties are an excellent way of manipulating the values of [Resource]-annotated data members into a form for client's to consume. Placing the [ResourceProperty] attribute on a C# property causes information about the return type of that property to be exposed as part of the RPD. Whenever a client calls one of the WS-ResourceProperty functions to retrieve the value(s) of an RP, the appropriate C# property is run to compute them. For example, the "Sender" property in Figure 4 computes its value from the pkg element of the Resource. By writing a C# property set function (a "setter"), the service author can also process data from a SetResourceProperty invocation (presumably saving the processed data into some [Resource] annotated data member).

It is also possible to place the [ResourceProperty] attribute on a data member, causing the value of that data member to be exposed as a ResourceProperty. However, this makes sense only if the data member has both a [Resource] and [ResourceProperty] attribute because without the [Resource] attribute, the data member will not have a value loaded from the database and so will only ever have its default value. Applying both attributes to a data member is shorthand for saying that you want the value of that portion of the Resource to be directly exposed, i.e. with no transform on that value (as you might do with a C# property).

An interesting aspect of ResourceProperties as defined in the WSRF specifications is that they can have multiple values. The RPD defines a minimum and maximum number of values that each RP can have. Typically, WSRF.NET will infer the min. and max. number of values for an RP based on the type of the C# Property or data member annotated with the [ResourceProperty] attribute. For most types, the min. and max. number of values is set to "1". However, if the type of the C# property or data member is ArrayList, the minimum and maximum are set to "0" and "unbounded". These defaults have cover most of the cases we have encountered. However, if a different min and max are desired, the service author must do two things. First, they must appropriately parameterize the [ResourceProperty] attribute (see below). Second, they must write code in their [ResourceProperty] annotated C# property to enforce their restrictions on the number of values.

While WSRF.NET can automatically generate the XML schema of the ResourcePropertyDocument from the set of [ResourceProperty] attributes used within the service's port types, customization of this schema can be achieved through a set of parameters available on the [ResourceProperty] attribute. These are summarized in the following table.

Name	.C# Type	Description
Name	string	The Name of the ResourceProperty. Clients will refer to this RP using the QName consisting of Name and Namespace (see below). If this parameter is omitted, the name of the annotated Property or data member is used.
Namespace	string	An attribute that specifies the namespace of the RP's XmlQualifiedName. If none is specified, then the namespace of the port type in which this RP is declared is used.
Type	Type	Specifies the C# type of the values of the RP. Single valued RPs (those with MinOccurs=1 and MaxOccurs = 1) can omit this parameter because WSRF.NET will use the C# property's type as the ResourceProperty's type. However, if the RP can have multiple values (i.e. MaxOccurs > 1), the type of the C# property (or data member) should be set to ArrayList and the type of the values of the RP should be set using this parameter (and the C# property should return an ArrayList full of elements of that type).
Settable	bool	Specifies if this RP can be set by clients using the SetResourceProperties function. The default is false.
XsdType	string	Specifies the xsd:type to use in the XML schema for the ResourceProperty document. If this parameter is omitted, WSRF.NET uses .NET's built-in mappings between the C# type of the Property or data member (or the type given in the Type parameter above) and an xsd type. This parameter should be used in cases where there is no built-in mapping, or the mapping is ambiguous.
Nillable	bool	Determine whether or not null is a valid value for this RP. Default is false.
MinOccurs	string	The minimum number of values that this RP can have. Default is "1", except when the C# Property or data member's is an ArrayList, in which case the default is "0".
MaxOccurs	string	The maximum number of values that this RP can have. Default is "1". This parameter can be set to the string "unbounded" to indicate an infinite number of values. This is the default for C# Properties or data members that are ArrayLists.

Table 1. Parameters for the [ResourceProperty] Attribute

The [ResourceProperty] attribute has two constructors, one that takes no parameters (in which case all the defaults are used), and one that takes all the parameters in Table 1 (in the order they are shown). However, it is rare that a service author will want to specify

all the possible parameters and so the parameters can be set individually “by name” as shown with the Namespace and Type parameters in Figure 4.

```
public class PackageService : ServiceSkeleton
{
    [Resource]
    Package pkg;

    [Resource]
    PkgRoute route;

    [Resource]
    PkgLocation location;

    [ResourceProperty]
    public string Sender {
        get { return pkg.sender; }
    }

    [ResourceProperty(Namespace="http://www.pkg.com")]
    public DateTime EstimatedDeliveryDate {
        get { return ComputeDeliveryDate(location, route); }
    }

    [Resource]
    string[] comments;

    [ResourceProperty(Type=typeof(string), Settable=true, Nillable=true)]
    public ArrayList ReceiverComments
    {
        get
        {
            ArrayList rv = new ArrayList();
            if (comments != null)
                rv.AddRange(comments);
            return rv;
        }
        set
        {
            comments = new string[value.Count];
            value.CopyTo(comments);
        }
    }

    [WebMethod]
    public TrackingNum SchedulePackageDelivery (Destination d)
    {
        // create a new Package record and delivery route
        // - return TrackingNum that customer can use
    }

    [WebMethod]
    public bool CheckPkgIn(string location)
    {
        // adjust package's location in system
    }

    private DateTime ComputeDeliveryDate(PkgLocation l, PkgRoute r)
    {
        // estimate delivery date given location on route
    }
    ...
}
```

Figure 4. Using the [ResourceProperty] Attribute.

We can now modify the package service as shown in Figure 4. Each package handled by the package service has three pieces of information that we will expose in the

ResourcePropertyDocument, the package's sender, the package's estimated delivery date and any comments from the package receiver. The package service is now as follows:

The ReceiverComments RP is an ArrayList. The receiver of a package can leave comments for the deliver company by calling SetResourceProperty on this RP. The RP consists of between 0 and an unbounded number of string values, which can be null. Note how, despite the fact that the ReceiverComments RP is an ArrayList, the data is stored in the comments Resource which is a string array. This is because .NET (and hence WSRF.NET) cannot easily serialize ArrayLists since the type of each element is unknown. However, the C# property ReceiverComments shows how a service can be written to manipulate ArrayLists, which via the get and set methods on the property, are saved in WSRF.NET's database as fixed arrays. The Sender RP can be queried to get the name of the package sender. This value is stored in the pkg Resource and the Sender C# Property accesses that Resource whenever a client tries to get the value of the Sender RP. The schema of the ResourceProperty document for this web service will show that the Sender RP is of type string, and has default values for the other RP parameters. Finally, the EstimatedDeliveryDate RP is a DateTime whose value is determined by the Property's get function (which calls ComputeDeliveryDate()).

3.4. Adding Port Types to the Service

The WSRF specifications define functions that WSRF-compliant services may support. These functions are logically grouped into port types (though there is often a single function per port types in the WSRF specs). WSRF.NET allows a web service to "import" functionality defined in other port types using the [WSRFPortType] attribute. WSRF.NET comes with implementations of all the current WSRF specifications (as well as the WS-Notification family of specifications) and service authors can easily make use of that code using this attribute.

The [WSRFPortType] attribute takes a single parameter, a C# Type, that denotes the class of the port type to be imported. For example, if we want our service from Figure 3 to respond to the GetResourceProperty method [4] and the Destroy method [3], we would add attributes on the service class as shown in Figure 4.

```
[ResourceProperty("ReceiverComments", typeof(string), true, null,
                 true, "0", "unbounded")]
[WSRFPortType(typeof(GetResourcePropertyPortType))]
[WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
public class PackageService : ServiceSkeleton
{
```

Figure 4. The Package Service with [WSRFPortType] Attributes.

Service authors can create their own port types to import into services as well. This is often helpful when a certain set of functionality is used in a number of services. Any class that derives off ServiceSkeleton can be used as a port type. In fact, code that "is a service" in one service can be imported as a port type in another service. A service class must have the [WebService] attribute (see section 3.5) and it must have a .asmx file referencing it. Otherwise, a port type and a service are the same.

Creating a port type is the same as creating a service. A port type must derive off ServiceSkeleton and it can have [Resource], [ResourceProperty] and [WSRFPortType] attributes of its own. Port types are composed into services. For example, if a port type (A) uses the [WSRFPortType] attribute to import another port type (B) and a service (C) imports A, service C will have both port type A and B.

3.5. Adding Attributes for the WSDL Generator

WSRF.NET will automatically generate WSDL for web services. However, a set of attributes can be used to guide the process. These attributes are [WsdIBaseName], [WebService] and [WebServiceBinding]. The [WsdIBaseName] attribute allows the service author to control the name and targetNamespace used in the <wsdl:definitions> element of the generated WSDL files. [WsdIBaseName] takes two string parameters, a name and a namespace. The <wsdl:definitions> element in the WSDL service and port type definition files will have a targetNamespace equal to the namespace given in the [WsdIBaseName] attribute. The <wsdl:definitions> element in the WSDL bindings file will have a targetNamespace equal to the namespace given in the [WsdIBaseName] attribute with “/bindings” appended. The name attribute of the <wsdl:definitions> element will be the name parameter of the [WsdIBaseName] attribute with either “Service”, “PortType” or “Binding” appended (depending on which file is being generated).

Both .NET’s [WebService] and [WebServiceBinding] attribute can be used with or without the [WsdIBaseName] attribute. If [WebService] is specified, its name and namespace parameters override those of [WsdIBaseName] for the <wsdl:definitions> element of the service and port type WSDL files. If [WebServiceBinding] is used, its name and namespace parameters override those of [WsdIBaseName] for the <wsdl:definitions> element of the WSDL binding file.

Recall that in section 3.3, it was mentioned that RPs are referred to by QName, that is by a name and namespace. If the Namespace parameter is not used in the [ResourceProperty] attribute, the namespace for that RP will be the namespace of the port type in which it is declared. That namespace is set by the [WsdIBaseName] attribute (and/or the [WebService] attribute).

Finally, every service must at least have a [WebService] attribute and any of [WsdIBaseName], [WebService] and [WebServiceBinding] can be used on any of a service’s port types. WSDL files with appropriate names will be generated for each. However, the [WsdIBaseName] attribute does not have to be used. It is merely a convenience method used to provide default values that can otherwise be specified with the [WebService], [WebServiceBinding] and [SoapDocumentMethod] attributes.

4. Deploying a WSRF.NET Service

Deploying a service in WSRF.NET is done using the PortTypeAggregator tool. This tool generates a WSRF-compliant service from the author’s service (by importing port types, adding database manipulation code, etc.), generates the service’s WSDL (including the RP document schema) and deploys the service by modifying the service’s .asmx file.

WSRF.NET uses a VS.NET Add-in to automatically run this tool whenever a WSRF.NET web service is built. To tell VS.NET to run this tool over a web service, click on the service's .asmx file in the Solution Explorer (usually on the right hand side of the VS.NET window) and display the file properties. If the Add-in is running, the file will have a property named "WSRF" which can be set to true or false. When it is set to true, the generation of a WSRF-compliant service from that web service will happen whenever that web service is compiled.

Note: The PortTypeAggregator can generate errors even if the service compiles correctly (for example if the WSRF.NET attributes are incorrectly parameterized). However, VS.NET considers a project to have built successfully if it compiles. You should always check the build output window to ensure that the PortTypeAggregator also ran successfully.

4.1. Configuration

WSRF.NET services are ASP.NET web services and are therefore configured through the web.config file. WSRF.NET recognizes some additional web.config parameters. The following must be put in the <configSections> of the web.config file to enable WSRF.NET's additional parameters.

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

There are a number of configuration elements that can now be added to the web.config file. All of WSRF.NET's configuration elements must be placed inside a <wsrf-config> element as shown in Figure 5.

The <all-services> element allows the service to override settings in WSRF.NET's server.config (described in the installation instructions). The <machine-name> element contains a string of hostname (and optional port) that the service will use in the <Address> element of EPRs generated for the service's WS-Resources. The <transport-protocol> is a string used as the transport protocol in the <Address> element of the EPR. This value defaults to "http" if unspecified.

Normally, all the configuration information in a web.config file affects all the web services in the directory with the web.config file (i.e. you can put multiple .asmx files in the same directory if you wish). Using the <services> element, you can provide name/value pairs of configuration information for individual services. To do this, include a <service> sub-element with a name attribute that specifies the name of the web service's .asmx file (including the .asmx extension). <parameter> sub-elements of <service> should contain "name" and "value" attributes. These can be read by the service (i.e. within the service code) using the base.ServiceParameters hashtable. So, in the example below, base.ServiceParameters["DefaultResource"] returns true.

WSRF.NET defines several service configuration parameters that can be used on any service. Figure 5 shows several of these. The first is the DefaultResource parameter. If

this parameter is set to true (as shown below), then any request that comes into the service in which the EPR contained in the <To> SOAP header does not contain a <ReferenceProperties> element causes the “DefaultResource” to be loaded from the database and used as the execution context. This is useful for services that do not need multiple, unique WS-Resources, but would still like to take advantage of WSRF.NET’s ability to save/load dynamically changing service state. A second pre-defined parameter named “useCache” can have its value set to “true” or “false” (if omitted the default is false). If this parameter is set to true, the Resources for that service will be stored in an in-memory, write-through cache after being loaded from the database. This can improve performance for frequently accessed Resources. However, if an entity (service, program or human), other than the service itself, modifies the service’s Resources in the database, the service cannot detect this and its cache may become invalid. If your deployment permits one service to modify another’s Resources, or if a Resource’s state in the database can be modified by an entity other than the “owning” service, this parameter use not be used (so omit it or set it to false).

```

<wsrf-config>
  <services>
    <service name="MyService.asmx">
      <parameter name="DefaultResource" value="true" />
    </service>
    <service name="AServiceUsingServiceGroupPortType.asmx">
      <parameter name="useCache" value="true" />
      <parameter name="ServiceGroupEntryURL"
        value="%transport-protocol%://%machine-name%/WSRF/
Services/Core/ServiceGroupEntry.asmx" />
    </service>
    <service name="AServiceUsingNotificationBrokerPortType.asmx">
      <parameter name="SubscriptionManagerURL"
        value="%transport-protocol%://%machine-name%/WSRF/
Services/Core/SubscriptionManager.asmx" />
      <parameter name="PublisherRegistrationManagerURL"
        value="%transport-protocol%://%machine-name%/
%service-base-path%/PublisherRegistrationManager.asmx" />
      <parameter name="BrokerRegistrationRequired" value="false" />
    </service>
  </services>
</wsrf-config>

```

Figure 5. web.config Section for WSRF.NET Services

The other available parameters for the <service> element are used if the service implements particular port types. If the service implements the NotificationProducerPortType, NotificationBrokerPortType or ServiceGroupPortType (that is “includes” one or more of these port type via the WSRFPortType attribute), you must include the configuration elements shown in Figure 5. Service’s using the NotificationProducerPortType must include the SubscriptionManagerURL parameter. Service’s using the NotificationBrokerPortType must include the SubscriptionManagerURL parameter, the PublisherRegistrationManagerURL parameter and the BrokerRegistrationRequired parameter. Service’s using the ServiceGroupPortType must include the ServiceGroupEntryURL parameter.

The `SubscriptionManagerURL` parameter must contain the URL of a service implementing the `SubscriptionManagerPortType`. This is the service that will maintain the subscriptions for the `NotificationProducer` or `NotificationBroker`. If the service implementing the `NotificationProducerPortType` (or `NotificationBrokerPortType`) also implements the `SubscriptionManagerPortType` (that is if the service that is a notification producer also manages its own subscriptions), the macro `%my-url%` can be used for the `SubscriptionManagerURL` (see below for more information on configuration macros).

The `PublisherRegistrationManagerURL` parameter must contain the URL of a service implementing the `PublisherRegistrationManagerPortType`. This service will maintain the registrations of publishers for the `NotificationBroker`. The `%my-url%` macro can be used if the same service implements the `NotificationBroker` and the `PublisherRegistrationManager` port types.

The `ServiceGroup` parameter is used to specify the `ServiceGroupEntry` service that the `ServiceGroup` service should use. If the service implements both the `ServiceGroup` and `ServiceGroupEntry` port types, the `%my-url%` macro can be used.

4.2. Macros

Macros can make it easy to configure a service. For example, they are an easy way to configure `NotificationProducers` or `NotificationBrokers`. They allow the value of a particular configuration element to be resolved by the service at runtime. The allowable macros are shown in Table 2.

<code>%machine-name%</code>	Resolves to either the machine name specified in the service's <code>web.config</code> file or the machine name in the <code>WSRF.NET server.config</code> file (if none is specified in the <code>web.config</code>). If no machine name is specified in either file, the macro resolves to the machine name from <code>WebService.HttpServerUtil.MachineName</code> .
<code>%transport-protocol%</code>	Resolves to the transport protocol specified in the <code>web.config</code> or <code>server.config</code> file or "http" if neither config file specifies.
<code>%service-base-path%</code>	Resolves to a portion of the service's URL. For example, if the service's URL is <code>http://machine:port/one/two/three.asmx</code> , the service base path is "one/two".
<code>%service-name%</code>	Resolves to the name of the service. For the example above, the service name is "three.asmx".
<code>%my-url%</code>	Resolves to the service's URL. For the example above, <code>%my-url%</code> is the entire string <code>http://machine:port/one/two/three.asmx</code> .

Table 2. Config File Macros

5. Appendix

This section describes topics not covered in other sections of this manual. `WSRF.NET` provides a number of convenient library routines that service authors can use by deriving

their service from ServiceSkeleton. These are described here along with additional attributes that can be used. Also, we discuss how clients can make use of WSRF.NET services. This section is divided into service and client discussions and each sub-section is arranged by topic.

5.1. Service Programming Information

▪ How do I create a new stateful resource?

The WSRF specifications do not define any mechanism for creating new Resources. There are simply too many possible creation paradigms to standardize a single one. WSRF.NET provides two mechanisms for creating new Resources. First, we provide a `create()` method that a web service can use internally to create a new Resource. No matter how a service author chooses to expose the creation functionality, the `create()` method should be used to handle the interaction between the WSRF.NET service and the database where Resources are stored. This method is inherited from ServiceSkeleton.

When `create` is called, a new instance of every data member on every port type in the service that is annotated with [Resource] is created. These new instances, which collectively are the Resource, are then stored in the database under a WS-Addressing EndpointReference [13] that can be used to refer to the Resource subsequently. The EndpointReference (EPR) element has two sub-elements, `<Address>` and `<ReferenceProperties>`. `<Address>` contains the service's URL while `<ReferenceProperties>` contains a unique identifier for this Resource. The `create` method's prototype is:

```
public string create(string refProp, Hashtable parameters);
```

The first parameter, `refProp`, is a string specifying the unique identifier to use in the `<ReferenceProperties>` element. If none is specified, `create()` will automatically generate a unique string identifier (a GUID). The second parameter, `parameters`, is a hashtable collection of name/value pairs that will be passed to the `InitResource` method of each of the service's port types. These methods should set the initial values for the various data members in the Resource. Notice from section 3.2 that the `InitResource` method takes a hashtable parameter. The `InitResource` methods of the service's port types will receive the hashtable in the `create` call and they should extract needed setup parameters from it.

The `create` method returns a string which is either the identifier passed in as the `refProp` parameter or the unique string generated by the `create` method.

NOTE: Please read the Transactions information in section 5.3 for additional information on the correct use of the `create` method.

The second method for creating a new Resource is to use the WSRF.NET `GCGResourceFactory` port type. While WSRF does not define a standard port type used for Resource creation, WSRF.NET provides one which can work in a large number of circumstances. If this port type suits your application, it is easy to use. If it does not, then the `create` method can be used as part of the service author's create functionality.

There are two steps to using the `GCGResourceFactoryPortType`. First, add the attribute `[WSRFPortType(typeof(GCGResourceFactoryPortType))]` to your service class. Second, add `[ResourceInitializerType]` attributes to the service's port types. The `[ResourceInitializerType]` attribute takes a single parameter that is a C# type (i.e. `typeof(my_type)`). This type should contain all the information needed to initialize the port type's `[Resource]` annotated members. The `GCGResourceFactoryPortType` exports a single web method called `Create()`. When this web method is invoked by a client, the service will call the `create()` method described above. The `parameters` hashtable that is passed to `create()` will be set up by the `Create` web method as follows. Each key in the hashtable will be the full C# Type names of one of the port types in the service (i.e. `UVa.GCG.WSRF.ResourceLifetime.ScheduledTerminationPortType`). The corresponding value for that key will be an object of the type specified by the port type's `[ResourceInitializerType]` attribute. The various `InitResource` methods can select the object that contains the information they need and use it to initialize the portion of the Resource declared in that port type. The code in Figure 6 shows how to use the `GCGFactoryPortType` (see [10] for a discussion of how a client invokes the `Create` method).

```
namespace MyNamespace
{
    [WSRFPortType(typeof(GCGResourceFactoryPortType))]
    [WSRFPortType(typeof(MySpecialPortType))]
    public class MyService : ServiceSkeleton
    {
        // body of MyService
    }

    [ResourceInitializerType(typeof(MyInitType))]
    public class MySpecialPortType : ServiceSkeleton
    {
        public override void InitResource(Hashtable parameters)
        {
            MyInitType mit = parameters["MyNamespace.MyService"]
                               as MyInitType;

            // use the params
        }
    }
}
```

Figure 6. Using the `GCGResourceFactoryPortType`

- **How do I define the contents of the `<ReferenceProperties>` element used in EPRs that refer to a service's Resources?**

Any WS-Addressing header may contain a `<ReferenceProperties>` element [13]. In WSRF, the contents of this element are used to refer to a particular Resource managed by a service. WSRF.NET allows the contents of this element to be defined by the service author. The `ServiceSkeleton`'s `create` method (discussed above) can generate a unique name for a Resource. The `[ReferenceProperties]` attribute allows the service author to define the name and namespace for an XML element that will contain that unique name. This element will then be used in the `<ReferenceProperties>` element of that WS-Resource. For example, consider the following definition of the package service.

```

[ReferenceProperties("TrackingNumber", "http://shipping.com")]
[ResourceProperty("ReceiverComments", typeof(string), true, null,
    true, "0", "unbounded")]
public class PackageService
{

```

Figure 7. [ReferenceProperties] Attribute

The [ReferenceProperties] attribute means that all requests for a particular package Resource should contain a <ReferenceProperties> element with a <TrackingNumber> sub-element in the http://shipping.com namespace. An example EPR for this service then would be:

```

<EndpointReference
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
  <Address>
    http://shipping.com/PackageService/PackageService.asmx
  </Address>
  <ReferenceProperties>
    <TrackingNumber xmlns="http://shipping.com">
      some number
    </TrackingNumber>
  </ReferenceProperties>
</EndpointReference>

```

Figure 8. An Example EPR

▪ **How do I access ResourceProperties declared on the web service class?**

All services and port types in WSRF.NET derive of ServiceSkeleton. The C# property ServiceSkeleton.ResourceProperties returns a ResourcePropertyDocument containing all the resource properties that are declared on the service. The ResourcePropertyDocument can be indexed by qualified name and will return a ResourcePropertyValues object. ResourcePropertyValues.GetValues() will return an ICollection containing all the values of the RP. The ResourcePropertyInfo object can be used to serialize the values returned by ResourcePropertyValues.GetValues(). The following example shows this usage.

```

ResourcePropertyDocument rpDoc = ServiceSkeleton.ResourceProperties;
XmlQualifiedName qn = new XmlQualifiedName("someRP", "someNS");
ResourcePropertyValue rpv = rpDoc[qn];
ICollection vals = rpv.GetValues();
// now the values can be analyzed or you can
// serialize the whole document
XmlElement serializedRPDoc = rpDoc.Serialize();

```

Figure 9. Using the ResourcePropertyDocument

▪ **How do I add an AnyElement to a service's ResourcePropertyDocument schema?**

The WS-ResourceProperties specification [4] allows a ResourceProperty document to contain an "xsd:Any" element in the schema describing a service's ResourceProperties. In WSRF.NET, this can be accomplished by placing the [AnyResourceProperty] attribute

on the port type (or service) class. By using this attribute, the service is saying that the definition of its ResourceProperties can change over time. Changes can come from either external clients, or from the service itself. The attribute takes a single Boolean parameter, “settable”. If this parameter is set to true, then any reference by a client to a ResourceProperty that does not currently exist will cause the service to add that ResourceProperty. Presumably, client(s) would then add values to that ResourceProperty through SetResourceProperties calls. If “settable” is set to false, then only the service itself can add ResourceProperties. In this case, when a reference in the service code is made to a ResourceProperty that does not exist, the ResourcePropertyDocument will not throw an error, but add a new entry for the RP. *Note:* that while WS-ResourceProperties allows dynamically adding RPs, discovery of these new RPs is an open issue in WSRF.

- **How can one of a service’s port type’s access functions from another of the service’s port types?**

Sometimes it is necessary for a function in one port type to invoke a function of another port type. To do this, use the ServiceSkeleton.ServiceBase.GetPortTypeInstance() function. This function takes a string parameter that is the fully qualified type name of the port type that you wish to access. The function will return a ServiceSkeleton object that corresponds to the instance of that port type that is defined on the service. For example, the following code gets the ImmediateResourceTerminationPortType for the service.

```
using UVa.GCG.WSRF.Service.ResourceLifetime;

[WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
public class MyService : ServiceSkeleton
{
    ...
    string tn = "UVa.GCG.WSRF.Service.ResourceLifetime."
               + "ImmediateResourceTerminationPortType";
    public void SomeMethod()
    {
        ImmediateResourceTerminationPortType WSRLport =
            ServiceBase.GetPortTypeInstance(tn)
            as ImmediateResourceTerminationPortType;
        // now methods can be called on WSRLport
    }
}
```

Figure 10. Internally Invoking Methods on a Service's Port Types

- **How do I use WS-Notification in WSRF.NET?**

Generating a WS-Notification compliant notification message involves creating the message topic and then sending the message. Each of these is addressed below.

- **How do I create a notification topic?**

First, the service must import the NotificationProducerPortType using the [WSRFPortType] attribute. Then the service can generate a ProducibleTopic object for the topic of its notification(s). This can be done in two ways. First, the addTopicHeirarchy function of the ServiceBase.Topics member can be used. This function takes two parameters, an XmlQualifiedName and a bool. The qname contains

the name and namespace for the new topic. If the Name element of the XmlQualifiedName contains any slashes (“/”) the topic will be treated as a hierarchical topic (see Concrete topics in [8]) and all parent topics will also be created. For example, creating the topic “a/b/c” in some namespace will cause the topics “a” and “a/b” to be created also. The Boolean parameter indicates whether or not the topic is “final” [8], i.e. whether or not new topics can be dynamically added below this topic in the hierarchy. The second method for creating topics is to use the createTopicSpace() method of ServiceBase.Topics to create a new topic space for the service. This function takes a topic namespace. Then the addRoot() method of the generated TopicSpace can be called to add new root topics. This function takes a topic name and a Boolean indicating if the topic is final. Then each root topic created by addRoot() can add child topics with the addChild() method. This method takes similar parameters to addRoot(). Both mechanisms for creating topics are shown in Figure 11.

```
using UVa.GCG.WSRF.Service.Notification.Topics;

ProducibleTopic pt = ServiceBase.Topics.addTopicHierarchy(
    new XmlQualifiedName("new_topic/abc", "new_topic_ns"), false);

TopicSpace ts = ServiceBase.Topics.createTopicSpace("new_topic_ns");
ProducibleTopic pt2 = ts.addRoot("new_topic", false);
pt2.addChild("subtopic", true);

// to get topic (abc, new_topic_ns), use this:
ProducibleTopic pt3 = ServiceBase.Topics[new XmlQualifiedName("abc",
    "new_topic_ns")];
```

Figure 11. Creating a Notification Topic

Figure 11 also shows how to get a handle to the topics created using addTopicHierarchy by using an index (an XmlQualifiedName) into the ServiceBase.Topics member.

▪ **How do I generate/send a notification message?**

Once a ProducibleTopic object has been created, a notification message can be sent on that topic by calling that ProducibleTopic’s notify() function. This function takes a single parameter of type Object. If this Object is an XmlElement, the InnerXml of that element will be used as the <message> element [6] of the notification message. If the Object is of any other type, it will be XmlSerialized and used as the <message> element. If the Object is not XmlSerializable, an exception is thrown.

```
ProducibleTopic pt = ...;
XmlElement xe = ...
int i = 5;
pt.notify(xe);
pt.notify(i);
```

Figure 12. Sending a Notification Message on a Topic

▪ **How does a web service receive a notification message from another service?**

First, the receiving web service should import the NotificationConsumerPortType using the [WSRFPortType] attribute. Then you need to create a TopicExpression object. Recall

that a topic expression is a pattern that specifies a group of relevant topics. A WSRF.NET service that is a NotificationConsumer can have any number of TopicExpressions registered with it. Each registered TopicExpression has any number of associated delegate functions that will be called when a message on a topic matching the TopicExpression arrives. When a notification message arrives, the service compares the message topic against all of its registered TopicExpressions. If any TopicExpression matches the topic, that expressions delegate functions are called. A single notification message can trigger multiple delegates if it matches multiple registered TopicExpressions.

WSRF defines three “topic expression dialects” or ways of expression groups of topics named Simple, Concrete and Full [8]. WSRF.NET implements the Simple and Concrete dialects. Figure 13 shows how a TopicExpression is created and used.

```
TopicExpression te = WellknownDialects.SIMPLE.createExpression(  
    new XmlQualifiedName("name", "ns"));  
te.addHandler(new TopicExpressionListener(...));  
ServiceBase.TopicExpressions.registerExpression(te);  
  
public delegate void TopicExpressionListener(Topic topic,  
    NotificationMessageHolderType msg);
```

Figure 13. Receiving Notification Messages with the NotificationConsumerPortType

This code shows a TopicExpression that matches the topic defined by the QName (name, ns). This is a “Simple” topic expression, as defined by WSRF, because it matches exactly 1 topic. The `addHandler()` method takes a delegate of type `TopicExpressionListener`, the prototype for which is also shown in Figure 13. The final step is to register the new TopicExpression with the `ServiceBase.TopicExpressions` member. After this is done, whenever the service receives a notification message on a topic that matches the TopicExpression `te`, the delegate specified with `addHandler()` will be called.

When the `TopicExpressionListener` delegate is called, the `topic` parameter will contain the message’s topic and the `msg` parameter will contain the actual message.

▪ **How do I know when a Resource has been or is about to be destroyed?**

When one of the WS-ResourceLifetime [3] functions destroys a Resource, there are two Events that are raised. Interested port types within the service destroying the Resource can register callbacks to receive these Events and take appropriate action. The first Event is raised as the Resource is being destroyed and the second Event is raised once the Resource destruction is finished.

A port type can register for the “as Resource is being destroyed” Event by calling `WebServiceBase.addDeletionListener()` and passing in an object of type `DeletionDelegate`. In WSRF.NET, Resource destruction involves removing all entries related to that Resource from WSRF.NET’s database. Although the event is raised before the Resource is deleted, there is no guarantee that delegates will finish their execution before the deletion completes. In other words, delegates should not count on the Resource being available when they run. This event is useful when a port type must

do some cleanup (that is not dependent on the Resource's values) whenever a resource is destroyed.

WSRF.NET also raises a "destroyed" event when Resource deletion is finished. Interested port types can receive this event by calling `WebServiceBase.addDeletedListener()` and passing in an object of type `DeletedDelegate`. This event is useful for tasks such as sending notification messages when a Resource is destroyed. The prototypes for the `DeletionDelegate` and the `DeletedDelegate` are shown in Figure 14.

```
public delegate void DeletionDelegate(string resourceID);  
public delegate void DeletedDelegate(string resourceID);
```

Figure 14. Prototypes for the Deletion Delegate Functions

The string parameter to the delegates will be the string representation of the Resource's unique identifier (the value used in the element defined by the `[ReferenceProperties]` attribute).

▪ **How do I have a Resource that is (or includes) a Win32 process?**

If `[Resource]` is placed on a data member of type `ProcessHandle`, a Win32 process becomes part of the Resource. WSRF.NET automatically installs a Windows service called `ProcSpawnService` (which can be started from Control Panels -> Administrative Tools -> Services). This service can be invoked from within a WSRF.NET service to start a Win32 process and tie it to the `ProcessHandle` object. When a method invocation comes in to the service, a unique key for the process is loaded from the database and used to bind the `ProcessHandle` object to the process. Figure 15 shows code for using the `ProcessHandle` as a Resource and how to create a new `ProcessHandle` Resource, using the `createProcess` method.

```

[WSRFPortType(typeof(ProcSpawnNotificationConsumerPortType))]
public class MyService : ServiceSkeleton
{
    [Resource]
    private ProcessHandle proc = null;

    public override void InitResource (Hashtable parameters)
    {
        // start a process that is part of the new Resource
        parameters = (Hashtable)(parameters["LaunchAsUserPortType"]);

        // code omitted to extra executable, workingDir, username
        // and passwd strings as well as the args string[] from
        // the hashtable

        string[3] stdioNames = new string[3];
        stdioNames[0] = "stdin.txt";    // redirect stdin here
        stdioNames[1] = "stdout.txt";   // redirect stdout here
        stdioNames[2] = null;          // do not redirect stderr
        EndpointReferenceType clientEPR =
            (EndpointReferenceType)parameters["clientEPR"];

        proc = ServiceBase.createProcess(username, passwd, executable,
                                         workingDir, args, stdioNames,
                                         clientEPR);
    }

    protected override void portInit()
    {
        TopicExpression gramTopic = WellknownDialects.SIMPLE.
            createExpression(UVa.GCG.WSRF.Common.WS.
                Notification.WellknownTopics._PROCESS_DIED_TOPIC);
        gramTopic.addHandler(
            new TopicExpressionListener(OnNotification));
        ServiceBase.TopicExpressions.registerExpression(gramTopic);
    }

    private void OnNotification(Topic topic,
                               NotificationMessageHolderType msg) {...}
}

```

Figure 15. Using a Win32 Process as a Resource

The `InitResource` method shows how to start a new process using the `ProcSpawnService`. The `ServiceBase.createProcess()` method will communicate with the `ProcSpawnService` to start a process and returns a `ProcessHandle` object. The `createProcess()` method takes 7 parameters. The first and second are strings representing the username and password for the account that the process should run under. The 3rd parameter is a string with the full path to the executable. The 4th is a string specifying the initial working directory for the process. The 5th parameter is a string array containing the arguments to be given to the binary. The 6th parameter is a string array containing 3 strings that represent filenames used to redirect standard in, standard out and standard error respectively. If any of these strings is null, the corresponding stream is not redirected. Similarly if this parameter is null, no streams are redirected. Finally, the last parameter is an `EndpointReferenceType` that represents

the endpoint to which the ProcSpawnService will send a notification to when the process terminates.

Once a process is started as part of the creation of a new Resource, the `ProcessHandle` object will “point” to that process. `ProcessHandle` exports the same methods as the `System.Diagnostics.Process` class. However, WSRF.NET will keep the `ProcessHandle` object bound to the process even though the web service instance itself may be cleaned up by IIS between method invocations. Service authors can use the methods of the `ProcessHandle` object to, for example, create `ResourceProperties` that expose information about the running process. They may also create service’s that spawn computational jobs for users and return EPRs representing those process Resources.

The `ProcSpawnService` sends a notification message when a process terminates. In order to receive this notification, a service author must take the following steps. First, they must include the `ProcSpawnNotificationConsumerPortType` in their service. This can be done using the `[WSRFPortType]` attribute as shown in Figure 15. Second, they must register to receive the `_PROCESS_DIED_TOPIC`. This can be done using the `portInit` function (also shown in Figure 15). The code shown registers the web service to receive the “process died” notification by first creating a topic expression containing a single, well known, QName defined by WSRF.NET. Next a callback is registered with the `addHandler` method. In this case, the `OnNotification` callback when a notification on the `_PROCESS_DIED_TOPIC` topic arrives. Finally, the topic expression is registered with the service as a listener. Although the `ProcSpawnService` can technically send a notification message to any EPR, it is advisable to only have the `ProcSpawnService` notify the web service which originally launched the job. This service can then use WSRF.NET’s notification mechanisms, fully backed by IIS/ASP.NET, to send messages to any number of other interested parties.

NOTE: Although not required, it is perhaps easiest to call `ServiceBase.createProcess()` from within the `InitResource` method. This ensures that the EPR for the new Resource has been created (it will be passed to the `ProcSpawnService`). This is essential so that the `ProcSpawnService`’s notification messages contain the correct EPR of the Resource.

▪ **What hooks does WSRF.NET provide for me to customize the initialization of my web service?**

WSRF.NET provides two hooks that can be used to provide custom initialization of your web service. These hooks are the function `portInit()` and the event `WSResourceLoaded`.

Any class that derives off of `ServiceSkeleton` may implement (override) the function `portInit`. This function is a void function that takes no parameters. Its purpose is to perform initialization of a service/port type that cannot be done in the constructor because the `ServiceBase` member of `ServiceSkeleton` has not yet been initialized. In other words, `portInit()` is called after the `ServiceSkeleton`’s constructor is called.

On each invocation of a web service, state associated with a Resource is loaded from the database. The `WSResourceLoaded` event is raised after this occurs. This event allows for further initialization of the web service to do done (based on Resource state) before the invoked method begins.

One use of these hooks is to have a service that produces notifications on a dynamic set of topics. The current set of topics is stored as part of the Resource. The `portInit` function is used to add an event handler to the `WSResourceLoaded` event. When the `WSResourceLoaded` event is raised, the event handler adds the current set of topics to the topics on which the service can produce notifications. This is shown in Figure 16.

```
[WSRFPortType(typeof(NotificationProducerPortType))]
public class MyService : ServiceSkeleton
{
    [Resource]
    TopicInfo[3] topics;

    ...

    protected override void portInit()
    {
        ServiceBase.WSResourceLoaded += new EventHandler(myHandler);
    }

    private void myHandler(object sender, EventArgs e)
    {
        foreach (TopicInfo ti in topics)
        {
            ServiceBase.Topics.addTopicHierarchy(ti.Topic, ti.Final);
        }
    }
}
```

Figure 16. WSRF.NET Initialization Hooks

- **How do I find a group of resources in the database that all have a particular pattern?**

If your service wishes to access a group of resources, for example, if you write a web method that allows for resource discovery, `ServiceSkeleton` provides a function called `GetResources` for this purpose. `GetResources` works by automatically generating an XPath query that can be used to find multiple resources stored in the data base. `GetResources` allows the service author to specify values for `[Resource]`-annotated members that are value types. For example:

```
ResultElement[] results = GetResources(GetType(), "counterVal",
QueryOperations.MoreThanOrEqual, minVal);
```

returns Resources (in their serialized state) where the `[Resource]`-annotated member “counterVal” has a value greater than or equal to “minVal”.

- **How do I control the name of my service's ResourceProperty document as it appears in the Service's WSDL?**

The [ResourcePropertyDocument] attribute allows you to specify the name and namespace of your service's ResourceProperty document as it appears in the service's WSDL. The attribute should be placed on the service class.

- **How can I control concurrent access to a particular WS-Resource?**

The [LockResource] attribute allows many reader/one writer access to a particular WS-Resource. Each message that arrives at a service creates an independent thread (in ASP.NET) to process that request. If multiple messages arrive that refer to the same WS-Resource (i.e. have the same EPR in the message's <To> header), a potential race condition exists. If the [LockResource] attribute is placed on a method, WSRF.NET will use a mutex style lock whenever any thread tries to write new values to the database for a WS-Resource. All concurrent threads will block when they try and write until the current thread's write is finished.

- **How do I make my WSRF.NET service compatible with the version of WS-Addressing used by GT4?**

The Globus Toolkit version 4 (4.0.1) uses a different version of WS-Addressing (WSA) than WSE 3.0. In many cases, this difference is handled automatically by WSRF.NET because a WSRF.NET service will use the same WSA namespace in a response message as it received in the request message. This means that GT4 clients and WSE 3.0 clients can both interact with WSRF.NET services transparently. However, if a WSRF.NET service must make outcalls to GT4 services or to WSE 3.0 web services, additional configuration is required.

As installed, the services of WSRF.NET are configured to use the GT4 namespace in all service proxies that are part of the WSRF.NET service code base. If you are writing a service and want to use GT4-compatibility mode, do the following:

- set the <policy> element of your web.config file to have its filename attribute set to the file in which you declare the service's policy
- in your service's policy file, include the following boldface lines.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    <extension name="requireActionHeader"
type="Microsoft.Web.Services3.Design.RequireActionHeaderAssertion,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
    <extension name="NamespaceFilter" type="WsaFilterLibrary.NamespaceFilter,
UVa.GCG.WsaFilterLibrary" />
  </extensions>
  <policy name="WSANamespacePolicy">
    <requireActionHeader />
    <NamespaceFilter
      origNs="http://schemas.xmlsoap.org/ws/2004/03/addressing"
      newNs="http://schemas.xmlsoap.org/ws/2004/08/addressing"
      serverReflects="true"
      filterClient="true"/>
  </policy>
</policies>
```

Figure 17. Policy File to Make WSRF.NET Services use GT4 WS-Addressing Namespace

- add a [Policy] attribute to your service class with its string parameter set to the name of the policy you used in your policy file (for Figure 17, this would be [Policy("WSANamespaceFilter")]

If you are writing a service and you do not want to use the GT4 compatible WSA namespace, perform the same steps, but set value of the “filterClient” attribute of the <NamespaceFilter> element in Figure 17 to “false”. This will preserve the ability of your service to respond to requests using the namespace that was in the request, while making outcalls using the WSE 3.0 version of the namespace.

Any code written by the service author that uses a web service proxy to make outcalls must use the SetPolicy() method to choose which policy applies to that web proxy. If the service to be addressed by that web proxy is a GT4 service, the service author must make sure that the applied policy contains the boldface lines from Figure 17.

Note that a service can make outcalls to both GT4 services and WSE 3 services by using multiple proxies with different policies. However, WSRF.NET’s internal proxies (the code inside of the WSRF.NET service and common libraries) can only use GT4-compatibility mode or WSE 3 compatibility mode. This will typically make no difference to the service author except in the case of notification. A single NotificationProducer cannot send notifications to both GT4 services and WSE 3 services using WSRF.NET’s notification automation. However, it should be noted that GT4 services, WSRF.NET services, and web services based on WSE 2 can all interoperate using the GT4-compatible version of the WSA namespace (this is the default configuration).

5.2. Client Programming Information

WSRF.NET 3.0 is mainly focused on providing tools for service authors. However, WSRF.NET does provide some useful tools for client-side programmers as well. This section discusses how to write clients that access WSRF.NET services.

▪ How do I receive a notification message in my client program without using IIS?

A WSRF.NET service can receive notification messages simply by using the [WSRFPortType(typeof(NotificationConsumerPortType))] attribute. This works because services have both IIS and the ASP.NET framework to receive and process web requests (recall the notification messages are just like any other invocation on a service – they are invocations of the Notify() method). However, client-side programs need a light-weight way to receive notification messages without using IIS. There are two ways to receive messages using WSRF.NET. The classes AsynchronousNotificationListener and BlockingNotificationListener (from the UVa.GCG.WSRF.Common.HttpServer namespace) can be used as simple, light-weight HTTP servers for receiving notifications on the client-side. First, we consider using the non-blocking AsynchronousNotificationListener. This class allows delegates to be registered and called whenever notification messages arrive. To use it, create an AsynchronousNotificationListener with either an integer port number or a System.Net.IPEndPoint. The first code segment of Figure 18 shows how to do this.

```

using UVa.GCG.WSRF.Common.HttpServer;

{
    // using the asynchronous notification listener
    AsynchronousNotificationListener listener =
        new AsynchronousNotificationListener(5432);
    TopicExpression te = WellKnownDialects.SIMPLE.createExpression(
        new XmlQualifiedName("name", "ns"));
    te.addHandler(new TopicExpressionListener(notify));
    listener.registerExpression(te);
    listener.start();

    // do work
    listener.stop();
}

public void notify(Topic t, NotificationMessageHolderType msg)
{
    // do work
}

// using the blocking notification listener
BlockingNotificationListener listener =
    new BlockingNotificationListener(
        new IPEndPoint(IPAddress.Any, 5432));
listener.start();
NotificationMessageHolderType[] messages =
    listener.waitForMessages(0, 10000, null);

// do work
listener.stop();

```

Figure 18. Receiving a Notification Message on the Client-side

First an `AsynchronousNotificationListener` is created. Then, just as in Figure 13, we create a `TopicExpression` object and register a delegate to be called when notifications whose topics match that expression arrive. This `TopicExpression` is then registered with the `AsynchronousNotificationListener`. At this point, calling the listener's `start()` method will create a new thread that listens for notification messages. Calling the listener's `stop()` method stops the execution of this thread.

The `BlockingNotificationListener` is even easier to use. First a `BlockingNotificationListener` object is created with either an integer port or a `System.Net.IPEndPoint`. Then the listener's `start()` method is called. Now the `waitForMessages()` method can be used to block until notifications arrive. The method takes three parameters, a minimum wait time (in milliseconds), a maximum wait time (also in milliseconds) and a `TopicExpression` to wait for. If null is specified for this third parameter, the `BlockingNotificationListener` will listen for all message topics. Once this method is invoked, the `BlockingNotificationListener` will wait for at least the minimum wait time. If no messages are received within this time, the listener will wait until either a message is received or the maximum wait time elapses. The method returns

an array of messages received (multiple messages might possibly be received during the minimum wait). The second code segment of Figure 18 shows the use of this listener.

- **How do I create a WS-Resource when a service is using the GCGResourceFactoryPortType?**

Consider a service with the following attributes:

```
[WSRFPortType(typeof(GCGResourceFactoryPortType))]
[ResourceInitializerType(typeof(MyInitType))]
public class MyService : ServiceSkeleton
{ }
```

This service exports the `Create` web method which is defined by the `GCGResourceFactoryPortType`. The `Create` method takes a `Create` object as its parameter. This object contains a member called `PortTypeInitializers`, which is an array of `XmlElement`s containing the `XmlSerialized` versions of any objects specified by any `[ResourceInitializerType]` attributes on the service's port types. For example, since `MyService` has only one `[ResourceInitializerType]` attribute (because the `GCGResourceFactoryPortType` does not have this attribute), the `PortTypeInitializers` array for creating a new `Resource` on `MyService` should contain an `XmlSerialized` object of type `MyInitType`. Figure 19 shows the usage.

```
using UVa.GCG.WSRF.Common.WS;
using UVa.GCG.WSRF.Common.WS.Grid;

...
{
    GCGResourceFactoryBinding proxy = new
        GCGResourceFactoryBinding(serviceURL);
    Create creationParams = new Create();
    MyInitType mit = new MyInitType();
    // add some values to MyInitType

    creationParams.PortTypeInitializers = new XmlElement[]
    {
        { WSUtilities.Serialize(mit) }
    };

    CreateResponse cr = proxy.Create(creationParams);
    // cr.ResourceEndpoint contains the new WS-Resource's EPR
}
```

Figure 19. How a Client Uses the GCGResourceFactoryPortType

The `GCGResourceFactoryBinding` is a pre-generated proxy class for this port type that all clients can use by simply including the “Common Lib”, `UVa.GCG.WSRF.Common.dll`, from the `WSRF` bin directory. The `WSUtilities` class of `WSRF.NET` provides many helpful functions including `Serialize`, which will create an `XmlSerialized` version of the object passed in as its parameter. Finally, the `CreateResponse` object returned by calling the proxy's `Create` function includes the new `WS-Resource`'s `EPR`.

5.3. Other Issues

▪ Transactions in WSRF.NET

WSRF.NET uses transactions when interacting with the database in which it stores WS-Resources. For each invocation of a web method on a service, a transaction is started. If that invocation completes without an exception, the transaction is committed and any changes to the Resource used in that invocation are saved to the database. If an exception occurs in the invocation, then the transaction is rolled back and any changes made to the Resource are not saved. While these are the semantics that are most often desired (“commit on success”), they have implications when using MSDE (and possibly MySQL).

If, in the midst of a transaction, some information is written to the database, no other service will be able to read from the database until the transaction completes (this happens because WSRF.NET stores all Resources on a machine in the same table). This means that there should be no outcalls to other services (which will access the database) after your service writes to the database. Normally, this is not an issue because the service author is not writing directly to the database, only the WSRF.NET infrastructure does that. However, there are two situations in which the service author may cause a write to occur. The first is when a service uses the `WebServiceBase`'s `DBConnection` member to write. We advise against this kind of direct database manipulation. The second occurs when using WSRF.NET's `WebServiceBase.create()` method because this method writes the newly created Resource to the database immediately.

In summary, a web method should not invoke methods on other services (or try and manipulate the database) after it has written using the `DBConnection` or called the `create()` function. If you do, deadlock may occur.

6. 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/ws-resourcelifetime.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/ws-resourceproperties.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/ws-notification/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/ws-notification/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 (WS-Topics). <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] Morgan, M. and Wasson, G. 2004. WSRF.NET Developer Tutorial.
http://www.cs.virginia.edu/~gsw2c/WSRFdotNet/WSRF.NET_Developer_Tutorial.pdf
- [11] 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/ws-resource/ws-basefaults.pdf>.
- [12] 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
- [13] WS-Addressing. 2004. IBM/BEA/Microsoft Corporation.
<http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-addressing.asp>