Web Services-Web Service Provider

WebObjects supports Web Services both as a producer and a consumer, and it actually works quite well once you figure out how to get things properly configured. Hopefully this walkthrough can jumpstart that process for you.

Setting up a WO Web Services Project

Here are the basic steps for setting up a Web Services producer with WebObjects and Eclipse/WOLips:

1. Create a new WOApplication project
2. Edit the project's Build Path, and go to the Libraries tab
    a. Add the following external jars from /Library/WebObjects/Extensions.
        - axis.jar
        - commons-logging.jar
        - commons-discovery.jar
        - wsdl4j.jar
        - saaj.jar
        - jaxrpc.jar
    b. Edit the WO Frameworks collection and add the JavaWebServicesSupport framework from the System frameworks
3. Create a class to hold your web service methods. The methods do not need to be static and can both take complex types as parameters and return complex types as return values. For now, just return primitive types and/or String.
4. Edit your Application class and add WOWebServiceRegistrar.registerWebService("PublishedNameOfYourWebService", NameOfTheClassYouJustMade.class, true);

That's it. Now when you start your app, you can request http://yourserver.com/cgi-bin/WebObjects/YourApp.woa/ws/PublishedNameOfYourWebService?wsdl and it will return the autogenerated WSDL document that you can use with any number of web service clients to interact with your server.

Complex Types with WO Web Services

So now the issue of complex types. Returning complex types is fine, but you have to register the serializer and deserializer classes for each complex type you reference. If you do not, the server will attempt to serialize your object using the ArraySerializer (you'll see this exception on the server), and the client will complain about a nonsensical error with SYSTEMID (gotta love terrible error handling!). The fix for this is for each of your complex types, call the following method in your Application constructor:

```
WOWebServiceRegistrar.registerFactoriesForClassWithQName(new BeanSerializerFactory(_class, _qName), new BeanDeserializerFactory(_class, _qName), _class, _qName);
```

where _class is the Class object that represents your complex type, and _qName is the QName (fully qualified name) of the class as it appears in your WSDL document. For instance, if you created a complex return type named Person and it is in the com.yourserver.service package, _class would be com.yourserver.service.Person.class and _qName would be new QName("http://service.yourserver.com", "Person"). Notice that the namespace is the inverse of your package name. You will need to call this method for each of the parameters and return types your reference.

For the record, I have no idea why you have to do this step manually - The WSDL was autogenerated, and thus it KNOWS the classes and their QName WSDL mappings, but I was not able to get things to work properly without this step. If anyone knows why this is, or a way around it, please update this article.

With these registrations, you should now be able to communicate with WO using any standard Web Service client (Axis, .NET, etc).

Sessions and WO Web Services

You may have noticed in your Web Service methods that you have no WOContext, WORequest, WOSession, and friends passed in. Do not fret. The WebServiceRequestHandler takes care to hook you up in this department using Axis's MessageContext class. You can use the following code to get to your WOSession:

```
WOContext context = (WOContext)MessageContext.getCurrentContext().getProperty("com.webobjects.appserver.WOContext");
WOSession session = context.session();
```

or the shortcut

```
WOSession session = WOWebServiceUtilities.currentWOContext().session();
```

The following additional keys are accessible through the MessageContext:

- "com.webobjects.appserver.WOContext" = the WOContext for this request
- "transport.url" = I /believe/ this contains the full request URL up to the query string
- org.apache.axis.transport.http.HTTPConstants.MC_HTTP_SERVLETPATHINFO = contains the request's request handler path
- "Authorization" = contains the Authorization header, in the event that you need to process things like Kerberos/SPNEGO, etc.
- "remoteaddr" = contains the request's remote address

Consuming with Axis in Java

If you are using Axis to consume a WO Web Service, be advised that there is an outstanding bug (open since circa 2003, no less) that axis by default does not support passing more than one cookie to the server. WO sends both woinst AND wosid, so you lose your session ID from the client on the return trip to the server. This can be fixed by applying the patch from http://issues.apache.org/jira/browse/AXIS-1059 to your client's axis.jar. Axis 1.1 has been archived at Apache, but you can download the source from http://archive.apache.org/dist/ws/axis/1_1/ . The patch does not perfectly apply. There are two rejected hunks, but it should be very obvious how to fix the rejects (the patch has two System.out.println that it claims were in the original source that were not). After fixing that, you can setStoreSessionIdInCookies(true) on your server's WOSession and setMaintainSessions(true) on your client's ServiceLocator and you'll be good to go.

This Axis bug appears to be fixed in recent versions of Axis, including version 1.4. Trying to upgrade the version of Axis in your WO Web Services server is not likely to be a happy experience (and likely neither will be upgrading Axis in a Direct To Web Services client - though I haven't tried this). However, it does seem to be possible to use a later version of the Axis jars on the classpath of a WebObjects application that intends to use classes generated by WSDL2Java to connect to a remote Web Services server - assuming that there are no WebObjects classes included in the WSDL. It is important in this case that you use matching version of WSDL2Java.

Consuming with WebServicesCore.framework

There are several complications when it comes to using WebServicesCore with WebObjects, all of which stem from the WSMakeStubs generated code. Upon using the code generated by WSMakeStubs, you will run into the following issues that need to be fixed in its code:

WSMakeStubs

Apple provides a program called WSMakeStubs that is similar to WSDL2Java in Axis, except that it sucks. It will, however, at least give you a starting point for building your web service client code, and with the changes outlined below, you can end up with decent client APIs.

Running WSMakeStubs is very simple:

/Developer/Tools/WSMakeStubs -x ObjC -name NameOfServiceClass -url http://yourserver.com/cgi-bin/WebObjects/YourWOA.woa/ws/YourService?wsdl

This will produce Objective-C code that you can use to call your web service. As opposed to Axis, WSMakeStubs produces stateless code for your service (i.e. no session tracking or cookie support - only static methods for each method of your web service). All of the methods appear at the end of NameOfServiceClass.m that you will need to call. WSMakeStubs also produces WSGeneratedObj.m, which contains the lower level web service core calls.

Service Methods Without Return Values

Another bug in WSMakeStubs is related to methods that don't have return values. For void methods, the methods are never actually CALLED by WSMakeStubs. If you look at the code for the returnValue method, you will see that it never calls super getResultDictionary. The problem with this is that super getResultDictionary is the code that actually executes the web service method. Simply change the definition for your void method to be:

```
- (id) resultValue {
  return [self getResultDictionary];
}
```

And everything will work as planned.

Bugs and Changes to WSGeneratedObj

WSGeneratedObj is MOSTLY bug free. However, there there are a couple changes required to fix a memory leak it generates (from cocoadev.com):

At the end of getResultDictionary, add:

```
if (fRef) { // new code
  WSMethodInvocationSetCallBack(fRef, NULL, NULL); // new code
} // new code
return fResult; // original code
```

which now reveals that the NSURL that is used is double-freed, fixable by removing one line from createInvocationRef:

```
 NSURL* url = [NSURL URLWithString: endpoint];
 if (url == NULL) {
   [self handleError: @"NSURL URLWithString failed in createInvocationRef" errorString:NULL errorDomain:
kCFStreamErrorDomainMacOSStatus errorNumber:paramErr];
 } else {
   ref = WSMethodInvocationCreate((CFURLRef) url, (CFStringRef)methodName, (CFStringRef) protocol);
   // [url release]; remove this line
   ....
```

Another change I like to make in the generated is to remove the hard-coded service URLs and pass them in from the code that calls the service (much like Axis does). This should be a fairly straightforward change, but I wanted to make a note about doing it. It will be fairly common that you want to talk to a development server and a production server using the same code, and as a result, you will want that variable to be parameterized.

Passing a Complex Type to WO

WSMakeStubs provides no direct support for passing complex types around - All you get is an NSDictionary, and all you can send back is an NSDictionary, with no instructions as to what exactly is IN these dictionaries.

To send a complex type back to WO, you have to set the following keys in your dictionary:

```
 [dictionary setObject:@"http://extranet.mdtask.mdimension.com" forKey:(NSString *)kWSRecordNamespaceURI];
 [dictionary setObject:@"WSCompany" forKey:(NSString *)kWSRecordType];
```

Where kWSRecordNamespaceURI's value is the XML namespace of the type of the complex object you are passing, and kWSRecordType's value is the name of the type. On the WO side, the namespace will be the reverse of the type's class name, and the record type will be the name of the class. For instance, in the example above, the actual class on the server was named com.mdimension.mdtask.extranet.WSCompany .

The rest of the dictionary contains attribute=>value mappings. For instance, WSCompany in the example above has a "name" attribute, so the dictionary would also contains a "name" key that maps to the corresponding value.

When sending NSDictionary instances from Cocoa, the WO will fire the WOGlobalIDDeserializer and it will not properly parse the nsdictionary or nsarray, it appears that there is no default deserializer on the WO side for those classes.

One solution is to add

```
@implementation NSObject (NSObject_WOXML)

- (NSString*)xmlPlist {
   NSString*  error;
   NSData*  data = [NSPropertyListSerialization dataFromPropertyList:self
                                                              format:NSPropertyListXMLFormat_v1_0
                                                    errorDescription:&error];
   return [[[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding] autorelease];
}

@end
```

on the cocoa side, than call it when compiling the arguments for the WSMethodInvocationRef
Than on the WO side use NSPropertyListSerialization.propertyListFromString(xmlPlist) to recreate the object.

Return Values from WO

One of the other problems WSMakeStubs has is that it doesn't produce a valid identifier for retrieving a WO web service return value. In the generated code, you will see something like

```
 - (id) resultValue {
   return [[super getResultDictionary] objectForKey: @"getBillableCompaniesReturn"];
 }
```

however, the actual return value name requires its namspace to be included. The fixed version of the routine looks like:

```
- (id) resultValue {
  return [[super getResultDictionary] objectForKey: @"ns1:getBillableCompaniesReturn"];
}
```

Notice the key starts with "ns1:". This value should match the value that appears in your WSDL.

Example Type Wrappers

Here's an example type wrapper I use based on the WSCompany example above. In the static methods that WSMakeStubs creates that wrap my web service methods, I simply initWithDictionary this type with the result dictionary from the web service and return an instance of WSCompany rather than the dictionary. When I send one of these objects back, I simply send wsCompany dictionary in the wrapper method.

```
@interface WSCompany : NSObject {
  NSMutableDictionary *myDictionary;
}

-(id)initWithDictionary:(NSDictionary *)_dictionary;
-(NSDictionary *)dictionary;
-(NSString *)name;
-(NSString *)companyID;
@end
```

```
@implementation WSCompany

-(id)initWithDictionary:(NSDictionary *)_dictionary {
  self = [super init];
  myDictionary = [[_dictionary mutableCopy] retain];
  [myDictionary setObject:@"http://extranet.mdtask.mdimension.com" forKey:(NSString *)kWSRecordNamespaceURI];
  [myDictionary setObject:@"WSCompany" forKey:(NSString *)kWSRecordType];
  return self;
}

-(void)dealloc {
  [myDictionary release];
  [super dealloc];
}

-(NSDictionary *)dictionary {
  return myDictionary;
}

-(NSString *)name {
  return [myDictionary objectForKey:@"name"];
}

-(NSString *)companyID {
  return [myDictionary objectForKey:@"companyID"];
}
@end
```

Fault Handling

WSMakeStubs doesn't handle the fault properly but it's in the dictionary. In +resultForInvocation: I added a few lines to check for and return the fault

```
+ (id) resultForInvocation:(WSGeneratedObj*)invocation; {
  result = [[invocation resultValue] retain];
  // Added check if a fault occured and return the fault string if so
  if([invocation isComplete]) {
    if([invocation isFault]) {
      result = [[invocation getResultDictionary] valueForKey:@"/FaultString"];
    }
  }
  //
  [invocation release];
  return result;
}
```

Stateful Services

Below is the necessary code to enable cookie support and stateful session with the files generated by WSMakeStubs. This code also includes changes so the base web services URL is supplied in the init method and allows specifying a timeout value (which I defaulted to 30 seconds). To WSGeneratedObj.h, add three new member variables:

```
@interface WSGeneratedObj : NSObject {
  WSMethodInvocationRef fRef;
  NSDictionary* fResult;
  NSDictionary* fCookies;
  NSString fURLString;
  int fTimeout;

  id fAsyncTarget;
   SEL fAsyncSelector;
};
```

Here are the new methods to add to WSGeneratedObject.m:

```
-- (id) initWithWebServicesURLString:(NSString*)urlString
{
  if (self = [super init]) {
    fURLString = [urlString copy];
  }
  return self;
}

- (NSString*) getWebServicesURLString { return fURLString; }

- (NSURL*) getWebServicesURL { return  [NSURL URLWithString: [self getWebServicesURLString]]; }

- (NSArray*) getReturnedCookies
{
  NSDictionary *results = [self getResultDictionary];
  if (nil == results)
    return nil;
  CFHTTPMessageRef msgRef = (CFHTTPMessageRef)[results objectForKey: (id)kWSHTTPResponseMessage];
  NSDictionary *headers = (NSDictionary*)CFHTTPMessageCopyAllHeaderFields(msgRef);
  [headers autorelease];
  //parse the cookies
  NSArray *cookies = [NSHTTPCookie cookiesWithResponseHeaderFields: headers forURL: [self getWebServicesURL]];
  return cookies;
}

- (void) setCookies:(NSArray*)cookies
{
  [fCookies release];
  fCookies = [[NSHTTPCookie requestHeaderFieldsWithCookies: cookies] retain];
  WSMethodInvocationSetProperty([self getRef], kWSHTTPExtraHeaders, fCookies);
}
```

```
- (int)timeoutValue { return fTimeout; }
- (void)setTimeout:(int)t
{
  if (t >= 0 && t < 600)
    fTimeout = 30;
}
```

You will need to modify -dealloc to release fCookies and fURLString. Below is my modified version getCreateInvocationRef. It is modified to get the URL using the new accessor methods above, to get the method name from the class name (which makes a lot more sense than hard-coding it to the class name in every subclass), and to set the timeout. After that is a generic resultValues method so that your generated subclasses can have their -resultValues and -getCreateInvocationRef methods removed---the only methods they require are for setting parameters. There is also a commented out line that you can uncomment to have debug information included in the results dictionary. This is very helpful when trying to debug the transfer of complex objects.

```
- (WSMethodInvocationRef) genCreateInvocationRef
 {
    WSMethodInvocationRef invRef = [self createInvocationRef
              /*endpoint*/: [self getWebServicesURLString]
                methodName: NSStringFromClass([self class])
                protocol: (NSString*) kWSSOAP2001Protocol
                     style: (NSString*) kWSSOAPStyleRPC
                soapAction: @""
           methodNamespace: @"http://DefaultNamespace"];
   //set a time-out value
   if (fTimeout > 0) {
     WSMethodInvocationSetProperty(invRef, kWSMethodInvocationTimeoutValue, (CFTypeRef)[NSNumber numberWithInt:
fTimeout]);
 //      WSMethodInvocationSetProperty(invRef, kWSDebugIncomingBody, (CFTypeRef)kCFBooleanTrue);
   }
  return invRef;
 }

 - (id) resultValue
 {
   NSString *key = [NSString stringWithFormat: @"ns1:%@Return", NSStringFromClass([self class])];
   return [[self getResultDictionary] objectForKey: key];
 }
```

To use stateful services, call getReturnedCookies after the first request and store the cookie dictionary. Then call setCookies: with that dictionary on all of your subsequent web services calls. Depending on the cookies you use, you might want to save a new copy of the cookies dictionary after each request.