# WebServiceServer

## Using Cocoon as WebServiceServer

an approach with Flowcontrol, Continuations and JXTemplates based on ideas of

*http://www.mail-archive.com/users@cocoon.apache.org/msg20832.html
*joose and ugocei (irc.freenode.net#cocoon) Big Thanks to you both!

**Status:** Draft

---

Please add comments and corrections --JanHinzmann

### Motivation/Idea

The approach is to generate a received SOAP-Envelope with a StreamGenerator and extracting the called method and the arguments using XSLT, Flow and JXTemplates.

The flowscript will generate the SOAP-Envelope, extract the methodname and given parameters. It then will compute the request/method and finally send an answer-envelope to the client.

### Example 1

I have written an complete example with an 'echo' and a 'version'-method.

This can be downloaded here: soapwebservice.2005.06.01.tar.gz

### Example 2

Here is an example of a possible setup, building a webservice which provides an echo-method:

```
public class Webservice(){
    public synchronised String echo(String echo){
        return echo;
    }
}
```

Now customers are sending SOAP-envelopes (using a middleware like axis) like the following:

```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <ns1:echo soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="CodataWS">
        <ns1:arg0 xsi:type="xsd:string">Hello Echo!?</ns1:arg0>
      </ns1:echo>
    </soapenv:Body>
  </soapenv:Envelope>
```

This will be generated with a *org.apache.cocoon.generation.StreamGenerator* and forwarded from the sitemap to the flowscript with (be sure to have an XMLSerializer who will not include the <?xml ...-declaration):

```
...
      <map:serializer name="xmlnope"
                  logger="sitemap.serializer.xml" mime-type="text/xml"
                  src="org.apache.cocoon.serialization.XMLSerializer">
          <omit-xml-declaration>yes</omit-xml-declaration>
        </map:serializer>

<!-- == Webservice by Cocoon == -->
      <map:pipeline>
          <map:match pattern="webservice">
              <map:call function="mainWS"/>
          </map:match>

        <!-- getting the soap-envelope -->
        <map:match pattern="soapData">
            <map:generate type="stream"/>
            <map:serialize type="xmlnope"/>
        </map:match>
...
```

The first matcher calls a flowscript, which will request the envelope and save it in an *ByteArrayOutputStream* using the second matcher. Here is the relevant part from the flowscript:

```
  function mainWS(){
    var soapData  = new java.io.ByteArrayOutputStream();

    //getting the envelope out of the request (can be done only once)
    cocoon.processPipelineTo("soapData", null, soapData);
  ...
```

Now that we have saved the soapData in an *ByteArrayOutputStream*, we are sending it back to the Sitemap, to get the Method out of the envelope ('echo' in this case). Therefore we are using an JXTemplate with a macro and a function in the flowscript which converts the string into SAX (this is from the mailinglist, I have found an error in it at line 6 (...setRootElement(ignore) NOT ...setRootELement(true) right joose?):

```
/**
 * function from joose: http://joose.iki.fi/cocoon/saxInJX.txt
 */
function stringToSAX( str, consumer, ignoreRootElement ) {
        var is = new Packages.org.xml.sax.InputSource( new java.io.StringReader( str ) );
        var ignore = ( ignoreRootElement == "true" );
        var parser = null;
        var includeConsumer = new org.apache.cocoon.xml.IncludeXMLConsumer(consumer, consumer );
        includeConsumer.setIgnoreRootElement( ignore );
        try {
                parser = cocoon.getComponent(Packages.org.apache.excalibur.xml.sax.SAXParser.ROLE );
                parser.parse( is, includeConsumer );
        } finally {
                if ( parser != null ) cocoon.releaseComponent( parser );
        }
}
```

now lets go back in the flowscript right to the point, after we have called the "soapData" matcher:

```
function mainWS(){
    var soapData  = new java.io.ByteArrayOutputStream();

    //getting the envelope out of the request (can be done only once)
    cocoon.processPipelineTo("soapData", null, soapData);
  ...
//here we go:

    //getting the method out of the soap-content
    cocoon.session.setAttribute( "saxer", stringToSAX  );
    var soapMethod   = new java.io.ByteArrayOutputStream();
    cocoon.processPipelineTo("soapMethod", {"soapData":soapData}, soapMethod);
    clog("soapMethod:\n " + soapMethod + "\n");
```

Back in the sitemap we generate the content with the JXGenerator and a Template as follows (be aware, that linebreaks and blanks after the </jx:macro> are going right in the xmlcontent):

```
<?xml version="1.0"?>
<soap xmlns:jx="http://apache.org/cocoon/templates/jx/1.0">
 <jx:macro name="inject">
  <jx:parameter name="value"/>
  <jx:parameter name="ignoreRootElement" default="false"/>
  <jx:set var="ignored" value="${cocoon.session.saxer( value, cocoon.consumer, ignoreRootElement )}"/>
 </jx:macro><inject value="${soapData}" ignoreRootElement="false"/></soap>
```

The matcher in the Sitemap looks like:

```
    <!-- which Method is called? -->
    <map:match pattern="soapMethod">
            <map:generate type="jx" src="xml/dummy.jx">
                    <map:parameter name="soapData" value="{flow-attribute:soapData}"/>
            </map:generate>
        <!--<map:transform src="xsl/soapMethod.xsl"/>-->
            <map:serialize type="xml"/>
              </map:match>
```

**Note**: the soapMethod.xsl should now transform the methodname out of the envelope. (work in progress) it could be something like:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>

    <xsl:template match="/soap/Envelope/Body">
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="*[1]">
        <xsl value-of select="local-name()"/>
    </xsl:template>

</xsl:stylesheet>
```

TODO: dispatch the method and generate eventually passed parameters with matchers like *soapMethod*

The answer can then be serialized with an other matcher like:

```
              <map:match pattern="answer">
            <map:generate src="xml/dummy.xml" />
            <map:transform src="xsl/soap.xsl">
              <map:parameter name="ret" value="{flow-attribute:ret}"/>
            </map:transform>
            <map:serialize type="xml"/>
         </map:match>
```

and a soap.xsl like:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="ret"/>

<xsl:template match="/">

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
     <ns1:helloResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:ns1="CodataWS">
           <ns1:helloReturn xsi:type="xsd:string">
                   <xsl:value-of select="$ret"/>
           </ns1:helloReturn>
    </ns1:helloResponse>
  </soapenv:Body>
</soapenv:Envelope>

</xsl:template>
</xsl:stylesheet>
```

The returnparameter would be passed from the flowscript as follows:

```
//sending the answer
        cocoon.sendPage("answer", {"ret":ret});
}//end of mainWS
```

Where this should be refactored to a JXTemplate maybe

---

🙂 Ok, thats my first wikipage. If its all crap, just delete it 😉

What do you think about this approach?

Any suggestions are welcome.

[JanHinzmann]

### Testing the echo webservice

Many webservice testing tools (like the ones included in famous XML editors such as XMLSpy and OxygenXML) need a Web Service Description in WSDL to invoke the service. Here is the description with which I managed to test Jan's implementation successfully :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="cocoonWS"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ec="cocoonWS"
    xmlns:types="cocoonWS/types">
    <!-- Type definitions -->
    <wsdl:types>
        <xs:schema targetNamespace="cocoonWS/types">
            <xs:element name="echo" type="xs:string"/>
        </xs:schema>
    </wsdl:types>

    <!-- Message definitions -->
    <wsdl:message name="EchoRequest">
        <wsdl:part name="echo" element="types:echo"/>
    </wsdl:message>
    <wsdl:message name="EchoResponse">
        <wsdl:part name="result" element="types:echo"/>
    </wsdl:message>

    <!-- Port type definitions -->
    <wsdl:portType name="EchoPortType">
        <wsdl:operation name="echo">
            <wsdl:input message="ec:EchoRequest"/>
            <wsdl:output message="ec:EchoResponse"/>
        </wsdl:operation>
    </wsdl:portType>

    <!-- Binding definitions -->
    <wsdl:binding name="EchoSOAPBinding" type="ec:EchoPortType">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="echo">
            <soap:operation soapAction="cocoonWS/echo"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="Echo">
        <wsdl:port name="Echo" binding="ec:EchoSOAPBinding">
            <soap:address location="http://localhost:8080/cocoon/webservice"/><!-- CUSTOMIZE -->
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

Don't forget to customize the location parameter for the soap:address element in the service definition (where there is a "CUSTOMIZE" comment inside the code above).

I just made a updated version of Jan's sample to add this WSDL file and I added a pattern in the sitemap to make the WSDL file available remotely : http://www.epseelon.org/cocoon/soapwebservice.zip

With that sample, you can just deploy your application, and when your testing tool asks for a URL where it can find the WSDL description you can just enter http://localhost:8080/cocoon/webservice (or whatever you customized above).

## Second approach (recommended) : using Axis API's and RTTI

I had published a first version of that but since then I managed to make the code more generic and use reflexion to call the method. In my particular application, the business code is on a Spring layer and I have several web services, each of which corresponding to a particular module. In that structure, the only thing you need is a flowscript file and a few simple pipelines. And it's important to understand that I wanted to be able to retrieve the result of the method call before sending it back to the requestor, in order to be able to apply a few pipelines on it, to "adapt" the output. The rest is commented code... First of all, the sitemap :

```
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
    <!-- ========================== Components ================================= -->
    <map:components>
        <!-- As in the previous approaches, we have to customize an xml serializer to remove the xml
declaration -->
        <map:serializers>
            <map:serializer name="xmlnope"
                logger="sitemap.serializer.xml" mime-type="text/xml"
                src="org.apache.cocoon.serialization.XMLSerializer">
                <omit-xml-declaration>yes</omit-xml-declaration>
            </map:serializer>
        </map:serializers>

    </map:components>

    <!-- This declares the flowscript file we will describe afterwards -->
    <map:flow language="javascript">
        <map:script src="flow/webservice.js" />
    </map:flow>


    <!-- ========================== Pipelines ================================= -->
    <map:pipelines>
        <!-- Web Service pipeline -->
        <map:pipeline>
                <!--
                   This pipeline extracts the SOAP envelope out of the HTTP body
                -->
                <map:match pattern="requestMessage">
                        <map:generate type="stream"/>
                        <map:serialize type="xmlnope"/>
                </map:match>

                <!--
                   And this one reinserts the response envelope in an HTTP body
                -->
                <map:match pattern="soapResponse">
                        <map:generate src="module:flow-attr:soapData"/>
                        <map:serialize type="xml"/>
                </map:match>

                <!--
                   This pipeline is just a convenience to allow access to the service description.
                   Don't forget to adapt the pattern to fit your needs
                -->
                <map:match pattern="**.wsdl">
                        <map:read src="wsdl/{1}.wsdl" mime-type="text/xml"/>
                </map:match>

            <!--
                And this pipeline is the one that reroutes all the requests to the webservice flowscript
function.
                This pattern was good for me because that sitemap is dedicated to webservices in my application.
                So don't forget to adapt it to your needs.
            -->
            <map:match pattern="*">
                    <map:call function="webservice">
                            <map:parameter name="module" value="{1}"/>
                    </map:call>
            </map:match>

        </map:pipeline>

    </map:pipelines>
</map:sitemap>
```

And that's it. As you can see, we made things much simpler. Now after all this plumbing thing, let's see what's inside with the flowscript file, webservice.js :

```
importPackage(Packages.java.io);
importPackage(Packages.org.apache.axis);
importPackage(Packages.org.apache.axis.message);
importPackage(Packages.java.util);

importClass(Packages.java.lang.System);
importClass(Packages.org.springframework.web.context.WebApplicationContext);

/**
 * This function is for the webservice, it is called, when a SOAP-Envelope is
 * received, gets the Parameters out of it and then dispatches the called method to
 * javaclasses in the package.
 * Then it finally generates a SOAP-response and serialises it back to the client.
 */
function webservice(){
        // The module corresponds to a specific webservice
        var module = cocoon.parameters["module"];
        var input = unpackRequest();
        debug("input",input);
        var output = processRequest(module, input);
        debug("output",output);
        //FIXME add fault generation if output == null
        var adapted = adapt(output);
        packResponse(adapted,input);
}

/**
 * This function unpacks the request, that is extracts the SOAP body out of the SOAP envelope.
 * WARNING ! It doesn't consider headers so you would have to add code to this function to take headers
 * into account.
 */
function unpackRequest(){
    var requestMessage = new java.io.ByteArrayOutputStream();
    cocoon.processPipelineTo("requestMessage", null, requestMessage);
    var message = new org.apache.axis.Message(requestMessage.toString());
    var soapPart = message.getSOAPPart();
    var envelope = soapPart.getEnvelope();
    var body = envelope.getBody();
    var it = body.getChildElements();
    var messageContent = it.next();
    return messageContent;
}

/**
 * That function is the contrary of unpackRequest. It packs the response content into a SOAP body and envelope.
 * The trick here, is to use the input object to get information like the prefix and namespace of the response.
 */
function packResponse(result,input){
    var envelope = new SOAPEnvelope();
    var content = new SOAPBodyElement(new PrefixedQName(
                    input.getNamespaceURI(),input.getMethodName()+"Response",input.getPrefix()
    ));
    var response = new MessageElement(
                    input.getNamespaceURI(),input.getMethodName()+"Return",result
    );
    content.addChild(response);
    envelope.addBodyElement(content);
    cocoon.sendPage("soapResponse", {"soapData": envelope.toString().getBytes("UTF-8")});
    return;
}

/**
 * This function is the core of the XML-RPC call. It parses the message content to extract the method name and
 * parameters and forwards all of that to the corresponding Java class in order to process the request. The
result
 * of this function is the return value of the method.
 */
function processRequest(module, input){
        var moduleName = (new java.lang.String(module)).toLowerCase();
        var moduleService = getModule(moduleName);
        var methodName = input.getMethodName();
```

```
	var moduleClass = moduleService.getClass();
	// Usually, using RTTI, you could retrieve the right method using the method signature
	// which is the combination of the method name, and the types of the parameters
	// but we don't know the types of the parameters as we all get them as String's
	// so we choose the method with the right name, which forbids to have several service methods
	// with the same name in a particular class.
	var methods = moduleClass.getMethods();
	var methodToInvoke = null;
	for(var i = 0; i < methods.length; i++){
		if(methods[i].getName().equals(methodName)){
			methodToInvoke = methods[i];
		}
	}

	if(methodToInvoke == null){
		return null;
	}

	// This part is tricky, because we have to parse all the String parameters we get into the
	// types the method to invoke awaits. This is done thanks to the parseParam function.
	var expectedTypes = methodToInvoke.getParameterTypes();
	// Should work but there seems to be a bug in RPCElement class so we need to go down a level
	// var parameters = input.getParams()
	var parameters = input.getChildren();
	var itParams = parameters.iterator();
	var parametersValues = new Vector();
	var cptParam = 0;
	while(itParams.hasNext()){
		var rpcParam = itParams.next();
		var typedParam = parseParam(rpcParam.getValue(),expectedTypes[cptParam]);
		parametersValues.add(typedParam);
		cptParam++;
	}

	try{
		var invocationResult = methodToInvoke.invoke(moduleService,parametersValues.toArray());
		return invocationResult;
	}
	catch(exc){
		exc.printStackTrace();
	}
}

/**
 * The purpose of this function is to parse a parameter according to the expected type.
 * Once again, there is no error management and this code assumes that stringValue can be parsed to type.
 * Furthermore, that function doesn't allow the use of complex type parameters, which is not really a problem
in
 * my case.
 */
function parseParam(stringValue,type){
	if(type.equals(java.lang.Double.TYPE)){
		return java.lang.Double.valueOf(stringValue);
	}
	else if(type.equals(java.lang.Integer.TYPE)){
		return java.lang.Integer.valueOf(stringValue);
	}
	else if(type.equals(java.lang.Boolean.TYPE)){
		return java.lang.Boolean.valueOf(stringValue);
	}
	else if(type.equals(java.lang.Float.TYPE)){
		return java.lang.Float.valueOf(stringValue);
	}
	else if(type.equals(java.lang.Byte.TYPE)){
		return java.lang.Byte.valueOf(stringValue);
	}
	else if(type.equals(java.lang.Short.TYPE)){
		return java.lang.Short.valueOf(stringValue);
	}
	else if(type.equals(java.lang.Long.TYPE)){
```

```
                return java.lang.Long.valueOf(stringValue);
        }
        else if(type.equals(java.lang.Character.TYPE)){
                return new java.lang.Character(stringValue.charAt(0));
        }
        else return stringValue;
}

/**
 * That is a convenience method that you MUST adapt to your architecture. This only suits my AndroMDA generated
 * Spring service structure.
 */
function getModule(name){
    var appCtx = cocoon.context.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
    var module = appCtx.getBean("beanRefFactory").getBean(name);
    return module;
}

/**
 * This is where you can add pipeline processing to the service output, using cocoon.processPipelineTo for
example.
 */
function adapt(input){
        //FIXME add more complex adaptation
        return input;
}

/**
 * logs the given String with a timestamp to <cocoonDir>/WEB-INF/logs/flow.log
 */
function log(logString){
    cocoon.log.info(logString);
}

/**
 * Console Logging aka consoleDebugging
 * if tomcat is used entriesare found in
 * $CATALINAHOME/logs/catalina.log
 */
function clog(logString){
    //print(logString);
    System.out.println(logString + "\n");
}

function debug(varName, varValue){
        clog(varName + " [" + varValue.getClass().toString() + "] : \n" + varValue);
}
```

And that's it. What we have here is very similar to some kind of "mini-XML-RPC-engine", something that we could do with Axis handlers (at least I assume so, since I never came to implement Axis handlers), except that we combine the power of Axis libraries with the one of Cocoon pipeline processing. Finally, it's very important to notice that there is absolutely no error management in that code while there can be so many problems. So don't forget to add some for your application, especially using SOAP faults.

[SebastienArbogast]

## Integrating external webservices into cocoon

An alternative way to integrate webservices into cocoon is to use the [WebServiceProxyGenerator] as a generator for your pipeline. The idea is to accept incoming connections from webservice consumers, pass them to the actual ws endpoint and make the result available within your pipeline.

```
<map:pipeline>
    <map:match pattern="wsproxy/*">
    <map:generate type="wsproxy" label="xml"
        src="http://your.url/context/services/{1}">
      <map:parameter name="wsproxy-method" value="post"/>
    </map:generate>
    <!-- do whatever you want here .. -->
    <map:serialize type="xml"/>
  </map:match>
</map:pipeline>
```

See also:
*http://cocoon.apache.org/2.1/userdocs/generators/wsproxy-generator.html
*http://codefoo.blogspot.com/2005/07/serving-webservices-using-apache-axis.html

[StefanPodkowinski]