WSDL Tools with a REST interface

Developer Documentation - BioCatalogue

Author: Dan Mowbray

E-mail: dan@danmowbray.com

ABSTRACT

The WSDL Tools comprise of a WSDL Parser, created originally for the EMBRACE Service Registry, to parse WSDL documents and build object models of SOAP web services. A comparison tool is also included, a utility that detects changes in two versions of a WSDL document to show the changes that have taken place in a SOAP web service over time.

Table of Contents

Introduction

The WSDL tool set that has been created includes a WSDL parser to produce an intermediate representation of a WSDL document, to yield a transition to a human readable text. A WSDL comparison tool is also included in the utility that is used to detect differences between to versions of a particular WSDL document. This is useful for testing for changes in the WSDL document over time. Along with the WSDL parser the tool will verify that a WSDL document complies with the WS-I standards for WSDL, thus helping to create stable, interoperable and high quality web services.

The tools are wrapped in a "RESTful" web service interface for portability and interoperability purposes. The use of this SOA technology allows the PHP code within the web service to interact with the underlying technologies of BioCatalogue through the use of XML as an intermediate representation.

Implementation Technologies

PHP

The primary programming language used to develop this project was PHP (PHP: Hypertext Preprocessor). PHP is one of the most popular web development languages, but is also used extensively as a command line scripting language through the use of PHP CLI(Command Line Interface). The syntax is derived from C, Java, and Perl making it easy to learn for most developers.

PHP integrates seamlessly with MySQL databases and this has almost become a de facto standard partnership for developing PHP web sites and applications.

Java

Java is a programming language that is owned, developed and maintained by Sun Microsystems. It is a platform independent programming language meaning the software can be compiled once and executed on any machine running the Java Runtime Environment (JRE).

Java has an abundance of flavours to meet the industry demand and to match the growth of the mobile device industry. This project will use Java SE JDK (Java Development Kit) version 1.5. Java SE is used for desktop application programming.

REST

REST (Representational State Transfer) is a specific architectural style adopted by the World Wide Web. The concept relating to the representational element of REST is that of a resource. A resource is any piece of information that can be identified uniquely. A message passed using a "RESTful" architecture usually contains both data and metadata. The data relates to the resource accompanied by metadata.

State Transfer within REST relates to the fact that a REST web service does not maintain any state of its interaction with a client. All state information is contained within the messages that are passed between the service and the

client. This makes for a more reliable service, one that can recover quickly from errors and doesn't consume large amounts of memory.

A "RESTful" interface is commonly referred to as a CRUD (Create, Retrieve, Update and Delete) interface. REST therefore maps perfectly onto the technology of HTTP.

HTTP supports only five operations GET and HEAD map to the retrieve property of a CRUD interface, POST to create, PUT to update and create, and DELETE to the delete property.

XML/XML-Schema

Extensible mark-up language is the ideal tool to represent messages passed between a web service and a client; the technology is self-describing and does not rely on other technologies. XML is used to represent WSDL documents following pre-defined standards. Types defined with the WSDL are defined using XML-Schema, used generally to specify the structure of an XML document. XML is a common tool in SOA technologies because of the properties it holds.

Design

WSDL Parser Component Design

The WSDL Parser will harness the majority of the functionality of the project. It will be used to build a programmatic model of a SOAP web service. An object-oriented approach will be adopted to create a *Service* object. This object will contain details of all attributes parsed from the WSDL document.

Data Structure

The WSDL Parser is required to build a model of the SOAP web service based on the information available in the WSDL document. The data structure has been developed based on the conceptualisation of each property of the WSDL specification, thus building a model that directly translates the XML of the WSDL into an object structure. The following class diagram outlines the classes that constitute this model.

Service

A Service class will be required to store key information related to the service such as:

- *name* – the name of the web service.
- *namespace* – the target namespace used within the WSDL document.
- *documentation* – a description of the web service taken from the WSDL document within the documentation elements of the XML.
- *ports* – a collection of *Endpoint* objects, reflecting the ports that are defined within the WSDL document.

The Service class will also require the functionality to output a description of the service that is readable by humans. This output will be in HTML format so that is can be integrated into the existing service description pages generated by the web site.

Endpoint

As per the specification for a WSDL document a service must have a port indicating where the client is to send request messages. Ports have the following information associated with them:

- *name* – the name of the port specified within the WSDL document.
- *protocol* – the protocol that is used to transmit the messages between the client and the web service.
- *style* – the SOAP style that is used either RPC or Document.
- *location* – the URL to the SOAP server which the client will use to send request messages.
- *functions* – an array of *Operation* objects used to represent the operations that are defined within the WSDL document.

Operation

The *Operation* class will contain only access methods to the private properties within the class. The properties for this class are:

- *inputMessage* – a message object that describes the input message specified by the WSDL document.
- *outputMessage* – a message object that describes the output message specified by the WSDL document.
- *faultMessage* – a message object that described the fault message specified by the WSDL document.
- *documentation* – a description of the operation taken from the WSDL document from within the documentation element.
- *action* – the SOAP action associated with the operation.
- *name* – the name of the operation.
- *operationType* – the operation type is determined automatically, if the operation has an input message and no output message then it is determined to be a 'request only' operation. If the operation has no input message and only an output message it is a 'response only' operation. Finally if the operation has both an input message and an output message associated with it then it is a 'request – response' operation.

Message

The *Message* class is used to store the detail of the messages sent between the client and the SOAP server. A message records details of the following:

- *name* – the name assigned to the message.
- *parts* – an array of the parts within the message.

MessagePart

The *MessagePart* class represents a part of a message. Message parts are used to specify the types sent and received as parameters within the message. A *MessagePart* has the following properties:

- *name* – the name given to the part of the message in the WSDL document.
- *type* – the type object associated with the message part, the type is parsed from the XML Schema within the *type* element of the WSDL document.

TypedElement

A *TypedElement* object represents an 'element' element defined within the XML Schema in the *type* section of the WSDL document. An element can have a type associated with it or it can reference another element. The class has the following properties:

- *name* – the name given to the element.
- *namespace* – the namespace object that represents the namespace in which the element belongs.
- *type* – the underlying type associated with the XML Schema element specified in the type attribute or nested within the element.

Namespace

A *Namespace* object represents a namespace associated with a type as specified within the XML Schema of a WSDL document. A the name space class holds the following properties:

- *prefix* – the prefix associated with the namespace within the XML Schema for example a common prefix used for the XML Schema namespace is 'xsd'.
- *uri* –URI stands for Universal Resource Identifier, this is the identifier associated with the namespace for example the URI for the XML Schema namespace is 'http://www.w3.org/2001/XMLSchema'.

Type

A *Type* object is used to represent an XML Schema type whether it is a *simpleType* or a *complexType*. The type object is part of a recursive data structure; such is the nature of XML Schema. An XML Schema e*lement* can have a *type* associated with it. The *type* can also contain one or more *elements*. The *Type* class has the following properties:

- *name* – the name associated with the type. XML Schema types can have anonymous types when a *type* is nested within an XML Schema *element*. In this case the name property would be left empty.
- *namespace* – the namespace in which this type resides.
- *elements* – a collection of *TypedElements* which are nested within the type definition.

XMLSchema

An *XMLSchema* object is used to represent an XML Schema this class is necessary because I will be creating a separate parser for parsing the XML Schema within a WSDL document. The XML Schema will be pre-processed by the *XMLSchemaParser* and create an *XMLSchema* object. The class has the following properties:

- *targetNamespace* – this property records the target namespace of the XML Schema this is gathered from the *schema* element of the XML Schema.
- *namespaces*- a collection of namespaces used within this XML Schema.
- *nestedSchemas* – a collection of *XMLSchema* objects from imported or included XML Schema from different locations.
- *types* – a collection of all *Type*s defined within the XML Schema.
- *elements* – a collection of *TypedElements* created whilst parsing the XML Schema.

The data structure outlined above builds a comprehensive model of SOAP web service and their associated properties. This model can be used to make comparisons and to programmatically build a description of the web service so that it can be displayed to the user in a human readable format rather than the raw WSDL document.

XML Schema Parser

The previous section briefly stated that the XML Schema within a WSDL document would be pre-processed using a preliminary parser. The *WSDLXMLSchemaParser* class is used to achieve this. The *WSDLParser* class invokes the parser before any WSDL is parsed.

The parser must be able to parse imported and included xml schema from different locations indicated by the *import* and *include* elements respectively within the XML Schema body. The imported schema will be parsed first as it is inevitable that the elements from the WSDL schema will need to refer to them.

The parser is required to search for all non-anonymous types within the schema. Once a type is found an instance of the *Type* class will be created. The nested elements within the type will be parsed and added to the *Type* object as nested *TypedElement* objects. This forms a recursive data structure as each of the *TypedElement*'s may have a type associated with them specified by the *type* attribute of the XML Schema *element* . The parser will search for the type either in the types that have already been parsed prior to the current type or it will find and parse the type on demand. All types that are parsed will be stored within an *XMLSchema* object.

The parser will finally parse all top-level elements with the XML Schema tag *element*. Message parts from within the WSDL refer to these elements. They also need to be stored within the *XMLSchema* object so that the *WSDLParser* class can access them via the *WSDLXMLSchemaParser* instance it created. Top-level elements are those that are a direct child of the *schema* element i.e. the root element of the XML Schema. The *WSDLXMLSchemaParser* will pass an *XMLSchema* object back to the *WSDLParser* containing all *Type, TypedElement*, and nested *XMLSchema* objects.

WSDL Parser

The WSDL Parser is contained within the *WSDLParser* class, the primary objective of the parser is to build the data structure to model a SOAP web service.

Firstly in order to pre-process the XML Schema nested within the WSDL *type* element the parser needs to invoke the parse function within the *WSDLXMLSchemaParser* class. This will create an *XML Schema* object.

The parser will start by creating the *Service* object to form the root of the data structure and parse its name from the *service* element within the WSDL definition and the target namespace of the definition.

Following the hierarchy of the data structure the parser will proceed by parsing each *port* within the *service* element and mapping the port to its *binding* within the WSDL definition by matching the binding information given by the *port*. The *binding* gives the SOAP action for each operation and the style and transport protocol used by the *port.* This information is stored in an *Endpoint* object. The parser will then look for *operation* elements nested within the *portType* element paired with the *binding* element.

Each *operation* within the *portType* definition is parsed in turn, parsing the input output and fault messages sequentially. Messages are parsed by descending through the message parts and matching the parts with a *TypedElement* defined within the *XMLSchema object* or if the element does not exist in the *XMLSchema* object, an XML Schema standard type is used. Message parts are stored as *MessagePart* objects and are stored in a *Message* object. An *Operation* object stores its input, output and fault messages. The operation is added to the corresponding *Endpoint* object.

Once the data structure is populated the *Service* object that is produced can be used to output a HTML description of the service of the web service on the web frontend.

WSDL Comparison Component Design

The WSDL comparison component will rely on the WSDL parser component to build a model of the web services scheduled for comparison. This model allows the comparison component to compare two versions of a web service by simply inspecting the data structure of each model.

When a service is added to the registry this action triggers a background process that uses the WSDL parser to build the model and cache the WSDL document in a predefined location so that it can be used to compare a future state of the web service to the current state, thus discovering any change that have occurred since the last time the WSDL was cached.

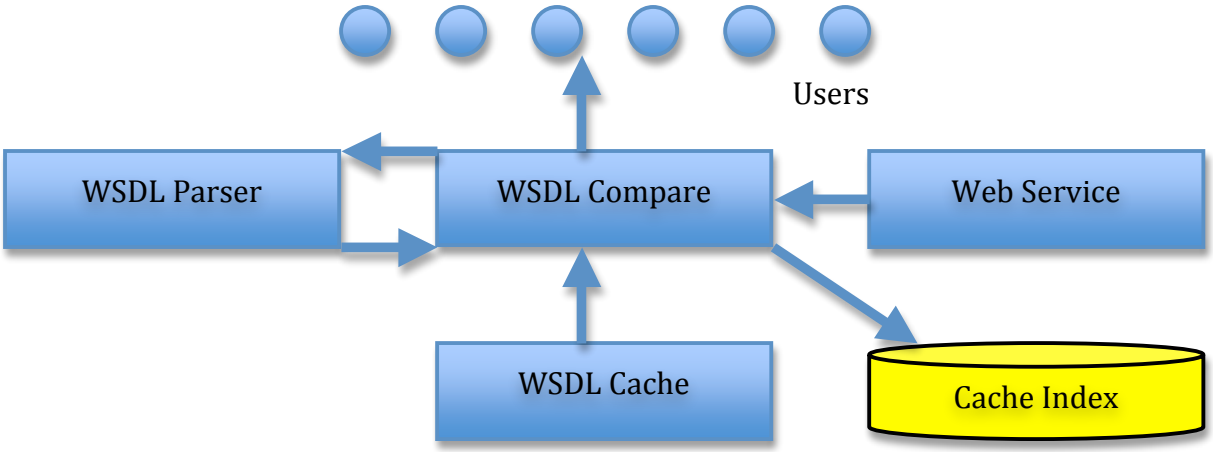

Figure A - The interactions of WSDL Compare with other system components

Figure A above shows how WSDL Compare will interact with the different components of the system. WSDL Compare obtains the WSDL document from the live web service and retrieves a cached version of the same WSDL document, it then passes the two documents to the WSDL Parser. The WSDL Parser will return a *Service* object that represents each of the documents. A comparison algorithm will then inspect the models, comparing the operations and port information across them. If there are any changes such as, new operations have been added, operations have been removed, or in fact any change to the model at all, these changes will be output. If changes were detected the new WSDL will be cached for retrieval in the future.

WSDL WS-I Compliance Checking

A third party component is used to check that the supplied WSDL document complies with WS-I standards, supplied by Ali Liaquat (Liaquat.Ali@bccs.uib.no). This component comprises of two parts, a WS-I Analyser and a Report Analyser. The WS-I Analyser investigates the WSDL document checking it against the WS-I standards; this generates a report that states the output and warning messages that describe the problems with the WSDL. This can be output to service owners to warn them of the issues.

The output from the report analyser is seamlessly integrated with the WSDL parser output that will be discussed in following sections.

REST Interface

The web service interface to the utilities will contain two functions, parse and compare that harness the functionality from the components discussed above. The REST interface will accept incoming requests via HTTP GET and POST and in return output XML representing the outcome of the operation selected. The REST interface will act as a web service wrapper that has direct access to the components discussed above, allowing heterogeneous access to the tools.

Implementation

Parsing WSDL

Following the design set out in previous sections, the WSDL Parser is the core component of this project. The primary functional property of the WSDL parser is to populate the data structure to model the SOAP web service. The parsing algorithm uses the PHP DOM (Document Object Model) to parse the WSDL document.

```
public function __construct($url)
{
                //set the file that is to be parsed and create the XML Reader

try{


$this->wsdlfile = $url;
$this->XDoc = new DOMDocument();
if($this->XDoc->load($this->wsdlfile))
{

        $this->service = new Service();
        $this->xsdParser = new WSDLXmlSchemaParser($this->wsdlfile);
}
else
{
        throw new Exception('could not load WSDL files');
}
}
catch(Exception $ex)
{
        throw $ex;

}


}
```

Figure B - The constructor for the WSDLParser class

The figure above shows the constructor for the WSDL parser creating a new DOMDocument object and loading the WSDL document into the object. DOMDocument offers functions such as *getElementByID(),* and *getElementsByTagName()* to obtain a list of XML elements from the WSDL.

A new service object is created to model the web service throughout the parsing process this object is the single access point to the model data structure.

As discussed in the design section the decision was made to pre-process all XML schema attached to the WSDL document before the remaining WSDL is parsed. This simplifies the algorithm as the XML schema doesn't have to traverse and parse the entire schema each time a type is referenced. The types are 'pre-parsed' and stored within the *XMLSchema* stored within the *WSDLXMLSchemaParser* instance. Appendix A shows the *parse* function of the *WSDLXMLSchemaParser*, it shows how each schema element within the WSDL is parsed dependent on its target namespace. Multiple *schema* elements can be defined within the WSDL *type* element, this is allowed so that different target namespaces can be used. The method used within the parse function is to sort the schema definitions into two groups. One group have the same target namespace as the WSDL document and another group have different target

namespaces. The schemas with differing target namespaces are stored in a separate *XMLSchema* and added to the native *XMLSchema* object as external XML schema objects. This prevents ambiguity between types with the same name that are defined within different namespaces. For example a type 'apple' from the namespace 'http://fruit.org' is not the same type as 'apple' from 'http://computers.org/companies', therefore the parser is required to recognise and make this distinction. Schema elements with the same name are merged into one *XMLSchema* object.

Parsing XML schema in this scenario has the following steps:

- Parse any imported XML schema specified within *import* elements.
- Parse any included XML schema specified within *include* elements.
- Parse and store any named types within the XML schema i.e. either a *complexType* or *simpleType* with a name within the *name* attribute of the element.
- Parse any elements that are an immediate descendent of the *schema* element that are referred to by the message parts within the WSDL.

Included XML schemas are merged into the native *XMLSchema* object and are assumed to have the same target namespace. However imported XML schemas are added to the *XMLSchema* object as nested schemas. Parsing imported and included XML schema is a recursive process that involves creating a new *WSDLXMLSchemaParser* instance for each import or include, returning an *XMLSchema* object that can be either nested or merged respectively.

There are three main functions within the *WSDLXMLSchemaParser* class; *parse(), parseElement(),* and *parseNamedType()*. The parse function initialises the parsing process; it essentially follows the steps listed above delegating the different parts of the process to ancillary functions. *parseNamedType()* traverses the XML structure searching for *complexType* and *simpleType* elements which have a value in their *name* attribute. Complex types are parsed by creating a *Type* object and populating the object with parsed *TypedElement* objects derived from the nested *element* XML elements within the type, as shown in figure C.

```
$elementlist = $complexNode->getElementsByTagName('element'); //list of the nested
elements within the type
for($i = 0; $i < $elementlist->length; $i++)
{

        $elementNode = $elementlist->item($i);

        $element = $this->parseElement($elementNode);

        $complexType->addComplexElement($element);
}
```

Figure C - a segment of PHP code that finds all XML Schema elements within a complexType

Simple types are parsed by creating a *Type* object as per the complex type method, but instead of storing all elements nested within the type the base type is stored this is specified within the *restriction* element. Figure D shows the algorithm for creating simple types.

```
$restrictionNode = $simpleNode->getElementsByTagName('restriction')->item(0);
if(!is_null($restrictionNode))
{
```

```
            $baseTypeElement = new TypedElement();
            $baseTypeElement->setName('restriction');
            $ns = new Namespace();
            $ns->setURI($this->currentSchema->getTarget());
            $baseTypeElement->setNamespace($ns);

            $baseText = $restrictionNode->getAttribute('base');

            $basesplit = split(':', $baseText);

            if(count($basesplit) > 1)
            {
                $baseType =  $this->findType($basesplit[0], $basesplit[1]);
            }
            else
            {
                $baseType = $this->findType(NULL, $basesplit[0]);
            }

            $baseTypeElement->setType($baseType);
            $simpleType->addComplexElement($baseTypeElement);
}
```

Figure D - a segment of PHP code that parses a simpleType and stores it in the data structure

The *parseElement* function is used for parsing an occurrence of an XML Schema *element*, whether it is a top-level element that can be referred to by WSDL message parts or nested within a *complexType*. An element is parsed using a recursive algorithm, as shown in figure E, that checks for an anonymous types defined within the *element* this may contain multiple elements, as shown in figure C, that are parsed by the *parseElement* function. The recursive process ends when there are no remaining nested types and instead types are referenced using the *type* attribute.

```
$complexTypeNodes = $elementNode->getElementsByTagName('complexType');
$simpleTypeNodes = $elementNode->getElementsByTagName('simpleType');
if(!is_null($complexTypeNodes) && $complexTypeNodes->length > 0)
{
                        $complexNode = $complexTypeNodes->item(0);

                        if($complexNode->parentNode == $elementNode)
                        {

                                $type = $this->parseComplexType($complexNode);

                                $element->setType($type);
                        }
}
elseif(!is_null($simpleTypeNodes) && $simpleTypeNodes->length > 0)
{
                        $simpleNode = $simpleTypeNodes->item(0);

                        if($simpleNode->parentNode == $elementNode)
                        {
                                $type = $this->parseSimpleType($simpleNode);

                                $element->setType($type);

                        }
}
```

Figure E - a segment of the parseElement function that checks for nested anonymous types within an element definition

Once all types, elements and external schemas have been saved into the *XMLSchema* object the *WSDLParser* instance can use this to reference types via the *WSDLXMLSchemaParser*.

The *WSDLParser* class uses the PHP DOM to search through each section of the WSDL document. Firstly it populates the properties of the *service* object created by the constructor function, as shown in figure B. From knowing the ports associated with the service definition the parser finds the binding information for each of the ports, this information is stored within the *Endpoint* instances.

The operations provided by each port are listed within the *binding* section of the WSDL document. Appendix A shows the process of matching the binding information with a *portType*, which lists the messages that are passed between the client and the service for every *operation*. An *operation* object is created for every *operation* listed in the *portType* the messages are parsed individually and are stored in the *inputMessage, outputMessage, and faultMessage* as instructed by the WSDL document. Each message is stored as an instance of the *Message* class. Each *part* of the *message* is parsed by finding the element that is referenced within the *XMLSchema* object that has been pre-processed. The *WSDLXMLSchemaParser* class provides a function called *findElement* that finds the element specified by the message part and returns a reference to the type.

After populating the *service* object with all the detail outlined above the model of a SOAP web service is complete. This object can be used for providing a description of the web service and comparing to another *service* instance.

Comparing WSDL

The comparison component consists of a comparison algorithm that makes use of the *Service* object that is created from the WSDL parser. The object allows the algorithm to inspect all aspects of the web service through access methods in the *Service* class and its children classes.

If it is the first time a particular WSDL document is parsed by the comparison algorithm it will output nothing, as it does not have any previously cached version to compare the live service too, therefore it caches a copy of the WSDL document to a WSDL cache accessed by a hash index contained in an XML file, as shown in figure F. When this WSDL document is next compared the previous version is retrieved from the cache and two instances of the WSDL parser are created. The first of the parsers is used to parse the cached copy of the WSDL document and the second to parse the current version. The models of the services can be compared in a rather trivial way. The algorithm looks for differences in every property of the *Service* objects; these differences are then output via the REST interface to the tool set.

```
try
        {
            //test for parse Errors
            $remoteParser = new WSDLParser($wsdlURI);
            $remoteService = @$remoteParser->parseFile();
            //cache wsdl;
            $wsdl_dom = new DOMDocument();
            $wsdl_dom->load($wsdlURI);
            $wsdl_dom->save("wsdl_cache/" . $hash . ".wsdl");
            chmod("wsdl_cache/" . $hash . ".wsdl", 0777);
            //record caching;
            $dom_root = $dom->documentElement;
            $wsdl_el = $dom->createElement("wsdl");
            $wsdl_el->setAttribute("hash", $hash);
            $dom_root->appendChild($wsdl_el);
```

```
            compareToXML(NULL);
        }
        catch(Exception $ex)
        {
            errorToXML($ex->getMessage());
        }
```

Figure F - caching a WSDL document for the first time.

WS-I Compliance Testing

The third party component supplied by Ali, is a Java 1.5 application that is invoked via a PHP wrapper that captures the output from the application to make use of it in the tool set. As discussed in the design section this component consists of two sub-components; a WSDL analyser and a report parser. The WSDL Analyser is called by the PHP wrapper and outputs a result of either 0 or 1. 1 means that the analyser has detected either a warning or error. If 1 is returned then the Wrapper will call the report parser that will further investigate the issue. The report parser will then output either 1 or 2, 2 meaning an error (an exception) occurred in the analysis phase and no errors or warnings were generated. 1 means the warnings and errors were generated and they are parsed from the standard output to be used in the application. If the result is 2 the application tries to gain further information from the analysis phase.

The output from the compliance test is integrated into the output from the WSDL parser.

User Guide

Installation

The application is packaged as a .tar.gz archive. Decompress the archive file and move the directory to your public html folder on the web server. The application will reside in a directory named “WSDLUtils”.

To make use of the WS-I validation tools, a configuration file will require some attention. “/WSDLUtils/wsi_tools/wsi-test-tools/common/Configuration.properties” requires alteration as follows:

```
analyzer_CMD=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/java/samples/Analyzer.sh

wrk_DIR=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-tools/java/samples/

anlyzer_CMD_Param=-config

analyzer_Input_ConfigFile=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/java/samples/analyzerConfig.xml

analyzer_CMD_Name=analyzer

analyzer_Input_ConfigFile_original=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-
test-tools/java/samples/original/analyzerConfig.xml

original_logFile_Location=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/java/samples/original/WSIMonitorOutput.xml~

log_File_Location=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/WSIMonitorOutput.xml~

reportFile_Location=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/java/samples/report.xml
```

```
test_Assertion_File=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/common/profiles/SSBP10_BP11_TAD.xml

original_reportFile_Location=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/java/samples/original/report.xml

analyzer_Input_ReportFile=<PATH TO PUBLIC HTML DIR>/WSDLUtils/wsi_tools/wsi-test-
tools/java/samples/report.xml
```

The WS-I Analyser uses this information to locate the various configuration files it makes use of. Substitute <PATH TO PUBLIC HTML DIR> with the path to your public html directory.

Change the ownership of the WSDLUtils directory and all descendent files and directories to the apache, or web server user commonly "daemon". Daemon is the user that is associated with background processes under UNIX type operating systems. This grants the permissions required to produce the report output from the WS-I analyser.

Overview

Once the configuration file is updated, the application is ready to use. The rest interface is a file in the root directory called "WSDLUtils.php" this PHP interface accepts both HTTP POST and GET requests and will return XML. The interface accepts two parameters; the "method" and the "wsdl_uri". This interface communicates with sub-components of the application, the WSDL parser, WSDL comparison and WS-I compliance testing tools. The REST interface provides two methods, "parse" and "compare". The two methods are discussed below.

Parse Method

The "parse" method makes use of the WSDL parser and the WS-I compliance testing components. An example request via HTTP GET would look something like this:

```
http://localhost/WSDLUtils/WSDLUtils.php?method=parse&wsdl_uri=http://prodom.prabi.fr/prodom/current/wsdl/essairunmkdom.wsdl
```

This request would yield the following response:

```
<wsdl>
<compliance_result>0</compliance_result>
<compliance_errors/>
<compliance_warnings/>
<service name="MkDomService">
<description/>
<namespace>http://prodom.prabi.fr/mkdom</namespace>
<documentation/>
<ports>
<port name="MkDomPort">
<protocol>http://schemas.xmlsoap.org/soap/http</protocol>
<style>document</style>
<location>http://prodom.prabi.fr/prodom/2006.1/cgi-bin/essaimkdom.cgi</location>
<operations>
<operation name="runMkDom">
<description/>
<action/>
<documentation>Calculate mkdom on fasta data</documentation>
<type/>
<messages>
<inputmessage name="MkDomRequestMsg">
<parts>
<part name="inputWrapper">
```

```
<type name="runMkDom">
<type name="InputFastaSeqs">
<type name="string"/>
</type>
<type name="ParametersBlast">
<type name="string"/>
</type>
</type>
</part>
</parts>
</inputmessage>
<outputmessage name="MkDomResponseMsg">
<parts>
<part name="outputWrapper">
<type name="runMkDomResponse">
<type name="output_in_mkdom_format">
<type name="string"/>
</type>
<type name="log_output">
<type name="string"/>
</type>
<type name="alignment_output">
<type name="string"/>
</type>
<type name="project_file">
<type name="string"/>
</type>
</type>
</part>
</parts>
</outputmessage>
</messages>
</operation>
</operations>
</port>
</ports>
</service>
</wsdl>
```

In the case of the above WSDL document the WS-I compliance phase passed hence the compliance_result element having the result of “0”. The XML also describes the ports, operations, messages, and message parts along with the associated complex types that the service contains. It is structured in a logical way making it easy for parsing by client applications.

An example of a service failing the WS-I compliance phase is as follows:

```
<wsdl>

<compliance_result>1</compliance_result>

_

<compliance_errors>

<xml-fragment xml:lang="en" xmlns:wsi-report="http://www.ws-
i.org/testing/2004/07/report/" xmlns:wsi-log="http://www.ws-
i.org/testing/2004/07/log/" xmlns:wsi-analyzerConfig="http://www.ws-
i.org/testing/2004/07/analyzerConfig/" xmlns:wsi-monConfig="http://www.ws-
i.org/testing/2004/07/monitorConfig/" xmlns:wsi-assertions="http://www.ws-
i.org/testing/2004/07/assertions/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">WSDL definition does not conform to the schema located at
http://schemas.xmlsoap.org/wsdl/soap/2003-02-11.xsd for some element using the WSDL-
SOAP binding namespace, or does not conform to the schema located at
http://schemas.xmlsoap.org/wsdl/2003-02-11.xsd for some element using the WSDL
namespace.</xml-fragment><xml-fragment xml:lang="en" xmlns:wsi-report="http://www.ws-
i.org/testing/2004/07/report/" xmlns:wsi-log="http://www.ws-
i.org/testing/2004/07/log/" xmlns:wsi-analyzerConfig="http://www.ws-
i.org/testing/2004/07/analyzerConfig/" xmlns:wsi-monConfig="http://www.ws-
i.org/testing/2004/07/monitorConfig/" xmlns:wsi-assertions="http://www.ws-
```

i.org/testing/2004/07/assertions/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">An Array declaration uses - restricts or extends - the soapenc:Array type, or the wsdl:arrayType attribute is used in the type declaration.</xml-fragment><xml-fragment xml:lang="en" xmlns:wsi-report="http://www.ws-i.org/testing/2004/07/report/" xmlns:wsi-log="http://www.ws-i.org/testing/2004/07/log/" xmlns:wsi-analyzerConfig="http://www.ws-i.org/testing/2004/07/analyzerConfig/" xmlns:wsi-monConfig="http://www.ws-i.org/testing/2004/07/monitorConfig/" xmlns:wsi-assertions="http://www.ws-i.org/testing/2004/07/assertions/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">A wsdl:types element contained a data type definition that is not an XML schema definition.</xml-fragment>

</compliance_errors>

<compliance_warnings>

<xml-fragment xml:lang="en" xmlns:wsi-report="http://www.ws-i.org/testing/2004/07/report/" xmlns:wsi-log="http://www.ws-i.org/testing/2004/07/log/" xmlns:wsi-analyzerConfig="http://www.ws-i.org/testing/2004/07/analyzerConfig/" xmlns:wsi-monConfig="http://www.ws-i.org/testing/2004/07/monitorConfig/" xmlns:wsi-assertions="http://www.ws-i.org/testing/2004/07/assertions/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">Exception:org.xml.sax.SAXException: Error: cvc-complex-type.2.4.a: Invalid content was found starting with element 'documentation'. One of '{"http://schemas.xmlsoap.org/wsdl/":part}' is expected.at
org.wsi.xml.XMLUtils$ErrHandler.error(XMLUtils.java:1128)at
org.apache.xerces.util.ErrorHandlerWrapper.error(Unknown Source)at
org.apache.xerces.impl.XMLErrorReporter.reportError(Unknown Source)at
org.apache.xerces.impl.XMLErrorReporter.reportError(Unknown Source)at
org.apache.xerces.impl.xs.XMLSchemaValidator$XSIErrorReporter.reportError(Unknown Source)at org.apache.xerces.impl.xs.XMLSchemaValidator.reportSchemaError(Unknown Source)at org.apache.xerces.impl.xs.XMLSchemaValidator.handleStartElement(Unknown Source)at org.apache.xerces.impl.xs.XMLSchemaValidator.startElement(Unknown Source)at
org.apache.xerces.impl.XMLNSDocumentScannerImpl.scanStartElement(Unknown Source)at
org.apache.xerces.impl.XMLDocumentFragmentScannerImpl$FragmentContentDispatcher.dispatch(Unknown Source)at
org.apache.xerces.impl.XMLDocumentFragmentScannerImpl.scanDocument(Unknown Source)at
org.apache.xerces.parsers.XML11Configuration.parse(Unknown Source)at
org.apache.xerces.parsers.XML11Configuration.parse(Unknown Source)at
org.apache.xerces.parsers.XMLParser.parse(Unknown Source)at
org.apache.xerces.parsers.DOMParser.parse(Unknown Source)at
org.wsi.xml.jaxp.DocumentBuilderImpl.parse(DocumentBuilderImpl.java:148)at
org.wsi.xml.XMLUtils.parseXML(XMLUtils.java:585)at
org.wsi.xml.XMLUtils.parseXMLDocument(XMLUtils.java:173)at
org.wsi.test.profile.validator.impl.wsdl.BP2703.validate(BP2703.java:88)at
org.wsi.test.profile.validator.impl.BaseValidatorImpl.processAssertions(BaseValidatorImpl.java:319)at
org.wsi.test.profile.validator.impl.wsdl.WSDLValidatorImpl.processDefinitionAssertions(WSDLValidatorImpl.java:592)at
org.wsi.test.profile.validator.impl.wsdl.WSDLValidatorImpl.validate(WSDLValidatorImpl.java:182)at
org.wsi.test.analyzer.BasicProfileAnalyzer.validateWSDL(BasicProfileAnalyzer.java:376)at
org.wsi.test.analyzer.BasicProfileAnalyzer.validateConformance(BasicProfileAnalyzer.java:209)at
net.embraceregistry.wsianalyzer.WsiAnalyzerExecutionImpl.executeAnalyzer(WsiAnalyzerExecutionImpl.java:288)at
net.embraceregistry.wsianalyzer.ExecuateAnalyzer.main(ExecuateAnalyzer.java:25)</xml-fragment><xml-fragment xml:lang="en" xmlns:wsi-report="http://www.ws-i.org/testing/2004/07/report/" xmlns:wsi-log="http://www.ws-i.org/testing/2004/07/log/" xmlns:wsi-analyzerConfig="http://www.ws-i.org/testing/2004/07/analyzerConfig/" xmlns:wsi-monConfig="http://www.ws-i.org/testing/2004/07/monitorConfig/" xmlns:wsi-assertions="http://www.ws-i.org/testing/2004/07/assertions/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">{http://www.ebi.ac.uk/WSMPsrch}inputParams,{http://www.ebi.ac.uk/WSMPsrch}WSArrayofFile,{http://www.ebi.ac.uk/WSMPsrch}data,{http://www.ebi.ac.uk/WSMPsrch}ArrayOf_xsd_string,{http://www.ebi.ac.uk/WSMPsrch}WSArrayofData</xml-fragment><xml-fragment xml:lang="en" xmlns:wsi-report="http://www.ws-i.org/testing/2004/07/report/" xmlns:wsi-log="http://www.ws-i.org/testing/2004/07/log/" xmlns:wsi-analyzerConfig="http://www.ws-i.org/testing/2004/07/analyzerConfig/" xmlns:wsi-monConfig="http://www.ws-i.org/testing/2004/07/monitorConfig/" xmlns:wsi-assertions="http://www.ws-i.org/testing/2004/07/assertions/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">The inline schema uses an element or type from the SOAP encoding namespace but the namespace has not been imported. The SOAP encoding namespace should be imported with an import statement before it is used.</xml-fragment>

```
</compliance_warnings>

…</wsdl>
```

The summarised output above shows that the WS-I compliance process output a result of 1 meaning the application found issues with the WSDL. The application returns the warnings and errors in the form of XML fragments so it is easy to parse them in external applications the value within the fragment gives the detail of either the warning or the error thus the reason why the WSDL is not WS-I compliant.

If a WSDL document fails the parsing process, meaning the WSDL document is not a valid WSDL document then the following output will be generated.

```
<wsdl>

<parseError>

Failed to access XML Schema component, invalid URL : ../common/ws_common_1_0b.xsd

</parseError>

</wsdl>
```

The output describes the error that was discovered during the parsing phase. This error means that the particular WSDL document that was supplied as an argument here cannot be parsed until the error is resolved.

Compare Method

The compare method makes use of both the parsing component and the comparison component. It uses the parser to build an object-oriented model of the SOAP web service and also a model of a cached version of the service.

The first time this method is called with a particular WSDL document it will cache the WSDL taking an initial snapshot of the document itself. This will return the following output.

```
<wsdl>

<result>0</result>

</wsdl>
```

The output above doesn't really imply any meaning other than the WSDL document is now cached and can be used in the future to compare with subsequent version of the document. The web service keeps this information in a cache.

If a WSDL document is compared to the cached version and there are no changes discovered then output will be the same of that above.

If changes are discovered, then they will be listed as shown below:

```
<wsdl>
<result>1</result>
<diffList>
<diff>Operation named xrunMkDom has been removed.</diff>
<diff>An operation named runMkDom has been added.</diff>
<diff/>
```

```
</diffList>
</wsdl>
```

The output above denotes that an operation has been removed from the SOAP service and a new one has been added.

The new version of the web service is now cached for comparison in the future.

Contact

Dan Mowbray

Mobile: +44 (0)7980 704823

E-mail: dan@danmowbray.com

Appendix A: Code Extracts

WSDLXMLSchemaParser Parse Function

```
public function parse()
        {
                try
         {
             if(!is_null($this->schemaElements))
             {
                 $sameElements = array();
                 $differentElements = array();

                 for($i = 0; $i < $this->schemaElements->length; $i++)
                 {
                     $schemaElement = $this->schemaElements->item($i);
                     if($schemaElement->hasAttribute('targetNamespace'))
                     {
                         if($schemaElement->getAttribute('targetNamespace') == $this-
>docSchema->getTarget())
                         {
                             array_push($sameElements, $schemaElement);
                         }
                         else
                         {
                             array_push($differentElements, $schemaElement);
                         }
                     }
                     else
                     {
                         array_push($sameElements, $schemaElement);
                     }
                 }

                 for($i = 0; $i < count($differentElements); $i++)
                 {

                     $this->currentSchemaElement = $differentElements[$i];
                     $this->currentSchema = new XMLSchema($this->currentSchemaElement-
>getAttribute('targetNamespace'));
                     $this->currentSchema->setNamespaces(parseNamespaces($this-
>domdoc));

                     $this->parseImports();
                     $this->parseIncludes();

                     $this->parseNamedTypes();

                     $this->parseElements();


                     $this->docSchema->addSchema($this->currentSchema);
                 }

                 for($i = 0; $i < count($sameElements); $i++)
                 {

                     $this->currentSchemaElement = $sameElements[$i];
                     $this->currentSchema = new XMLSchema($this->currentSchemaElement-
>getAttribute('targetNamespace'));
                     $this->currentSchema->setNamespaces(parseNamespaces($this-
>domdoc));

                     $this->parseImports();
                     $this->parseIncludes();

                     $this->parseNamedTypes();

                     $this->parseElements();


                     $namespaces = $this->currentSchema->getNamespaces();

                     for($nscount = 0; $nscount < count($namespaces); $nscount++)
```

```
                    {
                        $ns = $namespaces[$nscount];

                        if(!in_array($ns, $this->currentSchema->getNamespaces()))
                        {
                            $this->docSchema->addNamespace($ns);
                        }
                    }

                     //merge types
                    $types = $this->currentSchema->getTypes();
                    for($tcount = 0; $tcount < count($types); $tcount++)
                    {
                        $t = $types[$tcount];

                        $this->docSchema->addType($t);
                    }

                    //merge typed elements
                    $elements = $this->currentSchema->getElements();
                    for($ecount = 0; $ecount < count($elements); $ecount++)
                    {
                        $e = $elements[$ecount];

                        $this->docSchema->addElement($e);
                    }

                  //merge schemas
                        $schemas = $this->currentSchema->getSchemas();
                        for($scount = 0; $scount < count($schemas); $scount++)
                        {
                            $s= $schemas[$scount];

                            $this->docSchema->addSchema($s);
                        }

                }




            }
        }
        catch(Exception $ex)
        {
            throw new Exception("cannot parse xml schema");
        }

       }
```

WSDLParser ParserOperation Function

```
private function parseOperations($port, $bindingNode)
       {
              try
        {
            $bindOpNodeList = $bindingNode->getElementsByTagName('operation');

            for($count = 0; $count < $bindOpNodeList->length; $count+=2)
            {
                $bindOpNode = $bindOpNodeList->item($count);

                $portTypes = $this->XDoc->getElementsByTagName('portType');

                for($typecount = 0; $typecount < $portTypes->length; $typecount++)
                {
                    $portTypeNode = $portTypes->item($typecount);
                    $portType = $portTypeNode->getAttribute('name');

                    $bindingNodeType = $bindingNode->getAttribute('type');
                    $split = split(":", $bindingNodeType);
                    if(count($split) > 1)
                    {
                        $bindingNodeType = $split[1];
```

```
                    }
                    else
                    {
                        $bindingNodeType = $split[0];
                    }
                    if($bindingNodeType == $portType)
                    {

                        $operationNodeList = $portTypeNode-
>getElementsByTagName('operation');

                        for($opcount = 0; $opcount < $operationNodeList->length;
$opcount++)
                        {
                            $operation = $operationNodeList->item($opcount);
                            if($bindOpNode->getAttribute('name') == $operation-
>getAttribute('name'))
                            {

                            $func = new Operation();
                            $func->setName($operation->getAttribute('name'));



                            $docNodes = $operation-
>getElementsByTagName('documentation');
                            if($docNodes->length > 0)
                            {
                                $docNode = $docNodes->item(0);
                                $func->setDocumentation($docNode->nodeValue);
                            }


                            $actionNodeList = $bindOpNode-
>getElementsByTagName('operation');
                            if($actionNodeList->length > 0)
                            {
                                $actionNode = $actionNodeList->item(0);
                                $func->setAction($actionNode-
>getAttribute('soapAction'));
                            }

                            $this->parseInputMessage($func, $operation, $bindOpNode);
                            $this->parseOutputMessage($func, $operation, $bindOpNode);
                            $this->parseFaultMessage($func, $operation, $bindOpNode);

                            $port->addFunction($func);
                            break;
                            }

                        }
                        break;
                    }
                }

            }
        }
        catch(Exception $ex)
        {
            throw new Exception("cannot parse operations");
        }
       }
```

Appendix B: XML Schema for REST interface

XML Schema for Parse Function Output

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="typeType">
	<xs:sequence>
		<xs:attribute name="name" type="xs:string" />
		<xs:element name=”documentation” type=”xs:string” minOccurs=”0” />
		<xs:element name="type" type="typeType" minOccurs="0" />
	</xs:sequence>
<xs:complexType>

<xs:complexType name="partType">
	<xs:sequence>
		<xs:attribute name="name" type="xs:string" />
		<xs:element name="type" type="typeType" />
	</xs:sequence>
</xs:complexType>

<xs:complexType name="messageType">
	<xs:sequence>
		<xs:element name="parts">
			<xs:complexType>
				<xs:sequence>
					<xs:element name="part" type="partType"
maxOccurs="unbounded" />
				</xs:sequence>
			</xs:complexType>
	</xs:sequence>
</xs:complexType>

<xs:complexType name="operationType">
	<xs:sequence>
		<xs:attribute name="name" type="xs:string" />
		<xs:element name="description" type="xs:string" />
		<xs:element name="action" type="xs:string" />
		<xs:element name="documentation" type="xs:string" />
		<xs:element name="type" type="xs:string" />
		<xs:element name="messages>
			<xs:complexType>
				<xs:sequence>
					<xs:element name="inputmessage"
type="messageType" minOccurs="0" />
					<xs:element name="outputmessage"
type="messageType" minOccurs="0" />
					<xs:element name="faultmessage"
type="messageType" minOccurs="0" />
				</xs:sequence>
			</xs:complexType>
		</xs:element>
	</xs:sequence>
</xs:complexType>

<xs:complexType name="portType">
	<xs:sequence>
		<xs:attribute name="name" type="xs:string" />
		<xs:element name="protocol" type="xs:string" />
		<xs:element name="stype" type="xs:string" />
		<xs:element name="location" type="xs:string" />
		<xs:element name="operations">
			<xs:complexType>
				<xs:sequence>
					<xs:element name="operation"
type="operationType" maxOccurs="unbounded" />
				</xs:sequence>
			</xs:complexType>
		</xs:element>
	</xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="serviceType">
	<xs:sequence>
		<xs:attribute name="name" type="xs:string" />
		<xs:element name="description" type="xs:string" />
		<xs:element name="namespace" type="xs:string" />
		<xs:element name="documentation" type="xs:string" />
		<xs:element name="ports">
			<xs:complexType>
				<xs:sequence>
					<xs:element name="port" type="portType"
maxOccurs="unbounded" />
				</xs:sequence>
			</xs:complexType>
		</xs:element>
	</xs:sequence>
</xs:complexType>

<xs:element name="wsdl">
	<xs:complexType>
		<xs:sequence>
			<xs:element name="compliance_result" type="xs:integer" />
			<xs:element name="compliance_errors" type="xs:string" />
			<xs:element name="compliance_warnings" type="xs:string" />
			<xs:element name="service" type="serviceType"
maxOccurs="unbounded" />
		</xs:sequnce>
	</xs:complexType>
</xs:element>


</xs:schema>
```

XML Schema for Compare Function Output

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="wsdl">
	<xs:complexType>
		<xs:sequence>
			<xs:element name="result" type="xs:integer" />
			<xs:element name="diffList" minOccurs="0">
				<xs:complexType>
					<xs:sequence>
						<xs:element name="diff" type="xs:string"
maxOccurs="unbounded" />
					</xs:sequence>
				</xs:complexType>
			</xs:element>
		</xs:sequence>
	</xs:complexType>
</xs:element>

</xs:schema>
```