Tutorial

Getting started

In order to annotate a service you must have the following things in place:

1. An OWL (<=v1.1) ontology describing the services's inputs and outputs, typically written using Protege
2. A JAX-WS web service. On the whole the builder will work will default annotation schema, however some JAX-WS annotations are not supported and will break the groundings. The service inputs and outputs can be expressed as conventional java beans, however some caveats apply
    - No `java.util.Maps`: maps are unsupported in the bindings, as there is no convention for serialising them to RDF. If you have map properties expose them as a collection of wrapped key-value pairs. (unordered collections (Sets) are supported as mappings to non-functional properties and ordered collections (arrays, descendents of java.util.List) are supported as mappings to (subclasses of) `http://www.ai.sri.com/daml/services/owl-s/1.2/generic/ObjectList.owl#List`
    - JAX-WS Schema elements must be qualified : by default in JAX-WS they are not, to make them so, add the following package annotation to pkg-info.java in the service package (and any packages containing beans)

```
@javax.xml.bind.annotation.XmlSchema(
      attributeFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED,
      elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
      package edu.bath.owlsbuilder.tests.annotated;
```

We'll start with a trivial example: Assuming we have the following ontology which describes a number object with an integer value:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY numbers "http://numbers.org/" >
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY Numbers "http://numbers.org/Numbers.owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<rdf:RDF xmlns="&numbers;Numbers.owl#"
     xml:base="&numbers;Numbers.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:numbers="http://numbers.org/"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:Numbers="&numbers;Numbers.owl#">
    <owl:Ontology rdf:about=""/>
    <owl:Class rdf:about="#Integer"/>
    <owl:DatatypeProperty rdf:about="#hasValue">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Integer"/>
        <rdfs:range rdf:resource="&xsd;int"/>
    </owl:DatatypeProperty>

</rdf:RDF>
```

And we have a service which performs, say addition:

```
package edu.bath.owlsbuilder.tests.annotated;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(targetNamespace = "http://test-primitive/")
public class AddServicePrimitive {

        @WebMethod
        public int add(
                        @WebParam(name = "x")  int x,
                        @WebParam(name = "y")  int y) {
                return x + y;
        }

}
```

Adding Service class annotations

We now add annotations to the service class which do the following:

1. Define a URI binding between the prefix `&num1;` and the Ontolgy URI `http://numbers.org/Numbers.ow`
2. Import the domain ontology into the service
3. Define the class as an OWL-S class with the numbers ontology as the default domain ontology

```
@URINamespace(prefix = "num", value = "http://numbers.org/Numbers.owl")
@OwlsImport("&num;")
@OwlsClass(defaultOntology = "&num;")
```

4. Define the service properties on the appropriate service method (annotated with `@WebMethod`)
5. Define OWL-S outputs on the method itself using the `@OwlsOutParam` annotation:

```
 @OwlsOutParam(name = "rv", owlType = "#Integer", bindings = { @OwlsBinding(from = ".", to = "hasValue")
})
```

For the output we define the type to be `#Integer` and the name to be `rv`

   a. Because the output of the method is a primitive value and the output type of the service is an object, we must bind the java location (relative to the return value, in this case "." : the return value itself) to an appropriate location within the return object (in this case to the OWL property "hasValue"). When the output groundings for this service are invoked, the value of the return type will be encapsulated into RDF as the `hasValue` property property of the `#Integer` object returned. More detail about bindings can be found below. For either sides of the binding the location "." is implicit and may be omitted, indicating either the Java parameter or return value, or the OWL-S parameter respectively.
   b. Multiple `@OwlsOutParam` annotations can be placed within an `@OwlsOutParams` annotation, each output can bind to the same or different parts of the output value.

6. We also define similar descriptions for input parameters on the Java parameters using the `@OwlsInParam` annotation:

```
   @OwlsInParam(name = "x", owlType = "#Integer", bindings = { @OwlsBinding(from = ".", to = "hasValue")
})
```

as with outputs each Java parameter can bind to several input values, although note that each OWL input parameter can only be linked to one java parameter (at the moment, until the OWL-S API handles Document encoding properly)

The annotated Java Class will now look something like this:

```
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

import edu.bath.owlsannotations.OwlsBinding;
import edu.bath.owlsannotations.OwlsClass;
import edu.bath.owlsannotations.OwlsInParam;
import edu.bath.owlsannotations.OwlsOutParam;
import edu.bath.owlsannotations.OwlsService;
import edu.bath.owlsannotations.URINamespace;

@WebService(targetNamespace = "http://test-primitive/")
@URINamespace(prefix = "num", value = "http://numbers.org/Numbers.owl")
@OwlsClass(defaultOntology = "&num;")
public class AddServicePrimitive  {

        @WebMethod
        @OwlsService(name = "AddServicePrimitive")
        @OwlsOutParam(name = "rv", owlType = "#Integer", bindings = { @OwlsBinding(from = ".", to = "hasValue")
})
        public int add(
                        @WebParam(name = "x") @OwlsInParam(name = "x", owlType = "#Integer", bindings = {
@OwlsBinding(from = ".", to = "hasValue") }) int x,

                        @WebParam(name = "y") @OwlsInParam(name = "y", owlType = "#Integer", bindings = {
@OwlsBinding(from = ".", to = "hasValue") }) int y)

        {
                return x + y;
        }
}
```

Generating an OWL-S description

To generate the OWL-S description from the class use the `edu.bath.owlsbuilder.OWLSBuilder` utility (a unix script which invokes this is in the install directory).

In order for the script to work, the service class and dependencies must be in your classpath.

The parameters are as follows:

> *owlsbuilder -h c-g <grounding> -g <gronuding2> -o <dir> -t <ontologyDirs> -w <wsdlLoc> <className> methodName*
> *-h --help*
> *-g --grounding <GroundingName> add a grounding (currently 'WSDL')*
> *-o --out <dir> write services to directory (with default names)*
> *-f --file <filename.owl> write single service to file (when method is specified)*
> *-t --ontologyDirs <paths> colon-seperated list of places to serach for ontologies*
> *-w --wsloc <http://a/Service> theoretical location of service base to add to OWL-S*

To generate the description for the above service, include WSDL groundings we must know the WSDL location of the service when it is deployed, assuming this is `http://localhost:8080/services/AddServicePrimitive?wsdl` we would invoke the buidler as follows:

> *./owlsbuilder.sh -g WSDL -f Primitive.owl -t test/ *
> *-w "http://localhost:8080/services/AddServicePrimitive?wsdl" *
> *edu.bath.owlsbuilder.tests.annotated.AddServicePrimitive*

This will generate a gronuded service description into `Primitive.owl`

Bean Bindings

In the above example the service described had parameters and return values which were primitive types it is also possible to bind complex java types (represented as java beans) to complex objects in the ontology. We do this by adding java annotations to the bean accessors themselves much as you do in the java XML binding system JAXB.

The following types of object can be bound to

- Primitive types (int/float/String/double/boolean etc.) can be bound to OWL dataType properties
- Beans can be bound to service parameters or owl Object properties
- Unsorted collections (java.util.Set) can be bound to OWL properties (ones which aren't declared as functional)
- Sorted collections (java.util.List, arrays) of Objects (can be bound to OWL-S ObjectList types and subclasses thereof, Within the ontology you may subclass the generic OWL ObjectList class to restrict it to the appropriate OWL contained types.

The following types cannot be bound:

- Sorted collections of primitive types: this doesn't work at the moment - but this may change if required
- Maps: these can be bound by exposing the map as an unsorted collection of an explicit type which refers to the key-value pairs of the map in the bean.
- Parameterised types other than lists or sets.
- Classes which do not have bean semantics
- Classes which cannot be serialised into JAXB: this is a restriction on JAX-WS itself.
- Beans which serialise to XML as a graph as opposed to a tree (e.g. using XmlIDRef).

Binding a Java bean classes

We start with a java bean which has a number of complex properties:

```
@XmlType(namespace = "urn:cmp-bean")
public class CmpBean {
        BNumber nestedBean;
        BNumber nestedBeanList[];
        int valInt;
        Set<String> stringCollection;
        Integer valCmpInt;
        boolean valBool;
        Date valDateTime;

        public BNumber[] getNestedBeanList() {
                return nestedBeanList;
        }

        public void setNestedBeanList(BNumber[] nestedBeanList) {
                this.nestedBeanList = nestedBeanList;
        }

        public BNumber getNestedBean() {
                return nestedBean;
        }

        public void setNestedBean(BNumber nestedBean) {
                this.nestedBean = nestedBean;
        }

        public int getValInt() {
                return valInt;
        }

        public void setValInt(int valInt) {
                this.valInt = valInt;
        }

        public Set<String> getStringCollection() {
                return stringCollection;
        }

        public void setStringCollection(Set<String> stringList) {
                this.stringCollection = stringList;
        }

        public Integer getValCmpInt() {
                return valCmpInt;
        }

        public void setValCmpInt(Integer valCmpInt) {
                this.valCmpInt = valCmpInt;
        }

        public boolean isValBool() {
                return valBool;
        }

        public void setValBool(boolean valBool) {
                this.valBool = valBool;
        }

        public Date getValDateTime() {
                return valDateTime;
        }

        public void setValDateTime(Date valDateTime) {
                this.valDateTime = valDateTime;
        }

}
```

and a corresponding ontology which describes an OWL object `#CmpBean` with corresponding object and datatype properties. We re-use the Number Ontology from the previous examples to encapsulate the list of nested beans.

Note the presence of an `@XmlType` declaration which binds the bean to a specific namespace in JAX-B if this is omitted then a namespace will be generated based on the bean's class name and package.

We annotate the bean in a similar way to annotating a service as follows:

```
@URINamespaces( {
                @URINamespace(prefix = "num", value = "http://numbers.org/Numbers.owl"),
                @URINamespace(prefix = "cmp", value = "http://complex.org/CmpBean.owl") })
@OwlBean(ontology = "&cmp;", name = "#CmpBean")
```

* This declares a couple of namespace prefixes , and identifies the class as a bound bean (using `@OwlBean`) .

- the `@OwlBean(ontology = "&cmp;", name = "#CmpBean")` annotation indicates which OWL class the bean is bound to and the ontology within which it is bound. In this case the name may be ommitted as the OWL class and java class names match.

For each property we then declare it's corresponding OWL properties using the `@OwlProp` annotation on the **getter** method of the bean property (field annotations are not supported). e.g. for the "valInt" property we would add simply:

```
@OwlProp()
       public int getValInt() {
               return valInt;
       }
```

In this case as the name of the OWL object property is in the same ontology as the bean and the bean property name ("valInt") matches that of the owl property "#valInt" the annotation body can be left empty.

If the property has a different name in the ontology, or exists in a different ontology, these may be specified in the annotation:

```
@OwlProp(value="someIntProperty", ontology="&otherOntology;")
       public int getValInt() {
               return valInt;
       }
```

We can use this mechanism to bind to other annotated beans as properties. e.g. if we had a bean property of type `BNumber` with the following annotations:

```
package edu.bath.owlsbuilder.tests.annotated;

import javax.xml.bind.annotation.XmlType;

import edu.bath.owlsannotations.OwlBean;
import edu.bath.owlsannotations.OwlProp;
import edu.bath.owlsannotations.URINamespace;

@URINamespace(prefix = "num", value = "http://numbers.org/Numbers.owl")
@OwlBean(ontology = "&num;", name = "Integer")
@XmlType(namespace="urn:a-number")
public class BNumber {

        public BNumber(){

        }
        public BNumber(int val){
                this.value = val;
        }


        int value;

        @OwlProp("hasValue")
        public int getValue() {
                return value;
        }

        public void setValue(int value) {
                this.value = value;
        }
}
```

we could bind the corresponding property in `CmpBean` using:

```
@OwlProp("nestedBean")
        public BNumber getNestedBean() {
                return nestedBean;
        }
```

to use the bound bean in a service we no longer need to explicitly bind the bean's properties in the service method:

```
@URINamespaces( {
                @URINamespace(prefix = "num1", value = "http://numbers.org/Numbers.owl"),
                @URINamespace(prefix = "cmp1", value = "http://complex.org/CmpBean.owl") })
@OwlsImports( { @OwlsImport(prefix = "num", value = "&num1;"),
                @OwlsImport(prefix = "cmp", value = "&cmp1;") })
@OwlsClass(defaultOntology = "&cmp;#")
@WebService(targetNamespace = "http://test-complexservice/")
public class ComplexService extends TestService {


        @OwlsService(name = "owlOP", label = "OpService")
        @OwlsOutParam(name = "rv", owlType = "CmpBean")
        @WebMethod
        public CmpBean cmpOp(
                        @WebParam(name = "inp") @OwlsInParam(name = "inp", owlType = "CmpBean") CmpBean inp) {
                return inp;
        }

}
```