Rapid Information Systems

Rapid SOA

CHANGE CONTROL

Date	Person	Description
16/05/2014	GE	Start of document
29/05/2014	GE	Added DataFactory and ErrorResponse

CONTENTS

INTRODUCTION

Rapid SOA is a lightweight library for supporting SOAP webservices in java web applications. Webservice operations are specified by naming a request class that implements the Rapid SOA getResponse interface. Getter and setter methods, or public variables, are decorated with annotations that detail the business and data rules. The request and response classes, and all of their getters, setters, and public variables are analysed at application start up with reflection to generate .wsdls with highly detailed request and response schemas. At runtime valid SOAP XML is unmarshalled into request objects, the getResponse method is invoked, and the response object is then marshalled back into SOAP XML, which is returned to the caller. Arrays are supported and a utility library for accessing relational databases is also provided.

The Rapid-SOA-1.1.jar contains the following packages and classes:

- com.rapid.soa
    - WSFactory.java
    - WSGateway.java (extends HttpServlet)
    - ErrorResponse.java
- com.rapid.data
    - DataFactory.java

CREATING A RAPID SOA WEBSERVICE APPLICATION

The intended use is to create a java web application with at least one servlet that extends WSGateway. It is also convenient to have a ServletContextListener in which you add your operations to the WSFactory as the application starts. You can also set up your logging at the same time, and deregister your logging and any jdbc libraries when the application is stopped.

Here is an example Gateway.java

```
package org.lbc.test;

import com.rapid.soa.WSGateway;
import com.rapid.soa.WSFactory;

public class Gateway extends WSGateway implements ServletContextListener {

        private static final long serialVersionUID = 1000L;

        public void contextInitialized(ServletContextEvent event) {

                // set the log path
                System.setProperty("logPath", event.getServletContext().getRealPath("/") + "WEB-
INF/logs/RapidSOA.log");

                // Log that we're starting
                getLogger().debug("Rapid SOA - Online Repairs is starting...");

                // get the WSFactory
                WSFactory wsFactory = getWSFactory();

                try {

                        wsFactory.addOperation("getHelloWorld",lbc.org.HelloWorld.class);

                } catch (Exception ex) {
                        System.out.println("Exception : " + ex.getMessage());

                }

                // Log that we've started
                getLogger().info("Rapid SOA - Online Repairs has started");

        }

        public void contextDestroyed(ServletContextEvent event){

                // get a reference to the logger
                Logger logger = getLogger();

                // This manually deregisters JDBC drivers, which prevents Tomcat from complaining about memory leaks
from this class
        Enumeration<Driver> drivers = DriverManager.getDrivers();
        while (drivers.hasMoreElements()) {
            Driver driver = drivers.nextElement();
            try {
                DriverManager.deregisterDriver(driver);
                logger.info(String.format("Deregistering jdbc driver: %s", driver));
            } catch (SQLException e) {
                logger.error(String.format("Error deregistering driver %s", driver), e);
            }
        }

                // Log that we're stopping
                logger.info("Rapid SOA - Online Repairs has stopped");

                // Shut down the logger
                LogManager.shutdown();

        }

}
```

Note how after the logging has been initialised, the WSFactory has been retrieved and an operation “getHelloWorld” is added to the factory. As the operations are added the library will begin the reflection and generation of the meta-data required to produce the detailed schema in the .wsdls.

This is the web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">

  <display-name>Rapid SOA – Test application</display-name>

  <servlet>
    <servlet-name>Gateway</servlet-name>
    <servlet-class>org.lbc.test.Gateway</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Gateway</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <listener>
    <listener-class>org.lbc.test.Gateway</listener-class>
  </listener>

</web-app>
```

Finally the HelloWorld class:

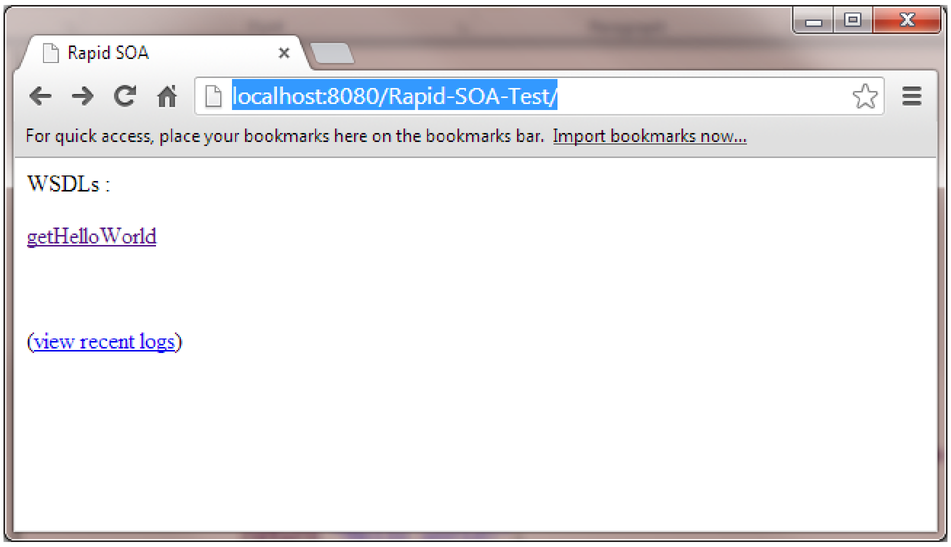
```
package org.camden.test;

import javax.servlet.ServletContext;

import com.rapid.soa.WSFactory;

public class HelloWorld extends WSFactory.Request  {


        @Override
        public String getResponse(ServletContext servletContext) throws Exception {


                return "Hello world!";


        }


}
```

And that’s the bare minimum for a working webservice!

Note that if you published your web application to localhost with context “Rapid-SOA-Test” you can visit this url:

http://localhost:8080/Rapid-SOA-Test/

It will show:


Rapid SOA
localhost:8080/Rapid-SOA-Test/
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

WSDLs :

getHelloWorld

(view recent logs)

Clicking on the getHelloWorld link will show the wsdl:

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://rapid-
is.co.uk/soa/.wsdl" xmlns:xsd="http://rapid-
is.co.uk/soa/"xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://rapid-is.co.uk/soa/.wsdl">

  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" targetNamespace="http://rapid-
is.co.uk/soa/">
      <xs:element name="HelloWorld">
```

```
        <xs:complexType>
          <xs:sequence/>
       </xs:complexType>
      </xs:element>
      <xs:element name="String" type="xs:string"/>
    </xs:schema>
  </wsdl:types>

  <wsdl:message name="Input">
    <wsdl:part element="xsd:HelloWorld" name="body"/>
  </wsdl:message>

  <wsdl:message name="Output">
    <wsdl:part element="xsd:String" name="body"/>
  </wsdl:message>

  <wsdl:portType name="PortType">
    <wsdl:operation name="getHelloWorld">
      <wsdl:input message="tns:Input"/>
      <wsdl:output message="tns:Output"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="getHelloWorldBinding" type="tns:PortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getHelloWorld">
      <soap:operation soapAction="getHelloWorld"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="Service">
    <wsdl:port binding="tns:getHelloWorldBinding" name="Port">
      <soap:address location="http://localhost:8080/Rapid-SOA-Test/"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

This can be passed to your favourite Webservice testing tool, like SOAPUI, and tested. As the webservice evolves over time so will the .wsdl. Updating the webservice definition in your testing tool will bring in the latest .wsdl and make sure it is kept in sync.

GETTERS AND SETTERS

To provide values to your request objects and receive them from your response objects Rapid SOA indentifies public “get” methods where there is a corresponding “set”, for example:

```
public class HelloWorld extends WSFactory.Request  {

        private String _name;

        public String getName() { return _name; }
        public void setName(String name) { _name = name; }

        @Override
        public String getResponse(ServletContext servletContext) throws Exception {

                return "Hello " + _name;

        }

}
```

At runtime Rapid SOA will “inflate” a “HelloWorld” object using the SOAP request and call the setName method to pass the contents of the “Name” element into the “_name” private variable.

Getters and setters must both be **public** and have exactly the same names (except for starting with a “g” or “s”, and the setter must have a single input parameter of the same type as the getters return type. If not, this property will not be included as an element in your webservice

PUBLIC VARIABLES

An alternative technique for passing values is public variables.

```
        public String Name;
```

Rapid SOA will simply pass the contents of the Name element in the SOAP request directly into the “Name” variable.

ANNOTATIONS

The annotations that define the rules present in the schema are placed above the get method for each property. They are:

`@XSDorder` – as reflection does not guarantee the order in which the properties appear, use this to specify the order the elements will appear in the sequence

`@XSDminOccurs` – the least number of times this element can appear. By default it is 1, set to 0 for the element to be non-mandatory (beware that when processing the response the property will then be null). When used with arrays it specifies the minimum number of elements required in the array.

`@XSDmaxOccurs` – the greatest number of times this element can appear. By default it is 1. When used with arrays it specifies the maximum number of elements in the array.

`@XSDnillable` – whether or not an empty element is allowed. Use to stop .Net sending empty elements instead of absent elements. This the difference between the property being available as null and an empty string and can make a big difference to your database!

`@XSDminLength` – the least number of characters required in the contents of the element. Works best with strings, set to 1 to stop empty elements.

`@XSDmaxLength` – the greatest number of characters required in the contents of the element. Works best with strings to limit the size of data passed to your database.

`@XSDpattern` – a regular expression used to compare the value of the element. Good for phone numbers, email addresses, etc.

`@XSDenumeration` – a comma separated list of values to create an enumeration. The element value must be present in the list.

`@XSDminInclusive` – the smallest allowed value. Works best with whole numbers.

`@XSDmaxInclusive` – the greatest allowed value. Works best with whole numbers.

`@XSDminExclusive` – values must be greater than this. Works best with floating point numbers.

`@XSDmaxExclusive` – values must be less than this. Works best with floating point numbers.

`@XSDchoice` – use to create a choice in the schema where, instead of all elements, only one element from those with the choice annotation must be provided. The annotation must be placed either at the top of the class, or on all properties in the class.

`@XSDname` – override the name of the element in the schema. Use with caution!

`@XSDtype`– override the type of the element in the schema. Use with caution!

Here is an example of a property including a private variable, getter, and setter. We have specified an optional string of between 1 and 20 characters when provided.

```
private String _name;

@XSDorder(1)
@XSDminOccurs(0)
@XSDminLength(1)
@XSDmaxLength(20)
public String getName() { return _name; }
public void setName(String name) { _name = name; }
```

Note how the XSD annotations appear above the getter method.

Here is an example of the equivalent public variable:

```
@XSDorder(2)
@XSDminOccurs(0)
@XSDminLength(1)
@XSDmaxLength(20)
public String Name;
```

Public variable are slightly weaker and don’t provide full object-orientated benefits, but see how much typing they save!

PRIMITIVE AND COMPLEX TYPES

Rapid SOA recognises the following as “primitive” types:

Java type	**SOAP type**	**Comments**
boolean	boolean	
float	decimal	
int	integer	
String	string	Not a Java primitive but treated as one by Rapid SOA
Date	date	This is the Java sql.Date and does not carry a time
Timestamp	dateTime	This is the Java sql.Timestamp and does carry a time

The above types can all appear as childless peers in a parent element. Should the type not be recognised it is considered a complextype and gets its own type definition in the schema.

Custom classes must all have zero parameter constructors and be public. For convenience the response classes are often defined within the request class. These “child” classes must be defined as “public static”.

ARRAYS

Rapid SOA will turn any primitive or complex type that is an array (for example String[] Party) into a type starting with “ArrayOf” (for example ArrayOfParty):

```
<xs:element name="Response">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ArrayOfParty" type="xsd:ArrayOfPartyType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="ArrayOfPartyType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="Party" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Note that it is better not to pluralise the names of array variables as the variable name is also used for the array member name. So in the above example we could have had “ArrayOfParties” as the array but then “Parties” as the name of the member element.

At this stage Rapid SOA does not support the List<?> interface. This is planned for a future release. In the meantime Lists need to be converted into arrays before being returned. Here is an example:

```
String[] partyArray = new String[partyList.size()];

response.setParty(partyList.toArray(partyArray));
```

LOGGING

Rapid SOA uses log4j and your project will have to have log4j.jar on its classpath as well as valid appender specified so the logger is initialised. A lot of detail is available on the DEBUG level including the full SOAP request and response.

DATA FACTORY

Rapid SOA includes an object called the DataFactory; its job is to return database result set and similar objects as easily as possible.

It can be initialised either with a jdbc connection string, user, and password, or if passed the servlet context will look for context parameters in your web.xml file named “jdbc.connectionstring”, “jdbc.user”, “jdbc.password”.

Usually only one data factory is required per method call with recordsets and related objects being automatically closed as new ones are requested. You should however call close on the data factory before the method returns to ensure all internal objects are cleaned up correctly.

Parameters

Pass parameters to the Data factory when requesting other objects by initialising a DataFactory.Parameters collection. This class has a number of overrides for easily passing in parameters of various types.

The parameters collection can be emptied with .clear() and also re-used in other data factory methods without needing to reinitialise it each time.

Prepared resultset

Call the DataFactory.getDataPreparedResultSet, passing in a sql statement, and parameters collections to get a java.sql.ResultSet. You can then loop the resultset in the usual manner.

The data factory uses prepared statements to avoid sql injection. Enter placeholders for your parameters in your sql as question marks “?”. For example:

```
select dummy, lower(?) input from dual where dummy = ?
```

The data factory will then match entries to the question marks in the order they were placed into the collection. This means that there must be as many entries in the parameters collection as there are question marks, and that should parameters be used in the sql more than once, they should be added to the collection as many times as they are required.

The sql statement above expects a parameters collection with two entries. The value of the first entry will be return in the resultset in lower case if it matches the value of “dummy”.

Prepared scalars

Call the DataFactory.getDataPreparedScalar method to have the value of the first column of the first row returned to you. Or “null” if no rows are found.

This method is very useful when only a single value is quickly required.

ERROR HANDLING

Rapid SOA includes a class com.rapid.soa.ErrorResponse. It has two properties "ErrorCode" and "ErrorMessage" and can be initialised by passing an object of type Exception. It also seeks to identify whether the Exception is a SQLException and if so incorporates the code value returned from this exception and attempts to clean up the messages returned by the Oracle database management system removing the ORA exception and line numbers.

It is especially useful when used with a "choice" xsd element:

```
public class Request extends WSFactory.Request {

        // response details object

        public static class Detail {

                ...

        }

        // response object

        public static class Response {

                @XSDorder(1)
                @XSDchoice
                public Detail Detail;

                @XSDorder(2)
                @XSDchoice
                public ErrorResponse ErrorResponse;

        }

        // request method

        @Override
        public Response getResponse(ServletContext servletContext) throws Exception {

                Response response = new Response();

                try {

                ...


                } catch (Exception ex) {

                        response.ErrorResponse = new ErrorResponse(ex);

                }

                return response;
        }

}
```

Consumers of your webservice will know from the .wsdl that they will receive either an ErrorResponse element, or a Detail element in the root of the response.

Unhandled exceptions will be returned by the WSGateway as SOAPExceptions.