ACM Web Service Tutorial

19/12/2014

ACM STUDENT CHAPTER AUTH

ACMCHAPTER@AUTH.GR

FB: ACM STUDENT CHAPTER AUTH

Outline

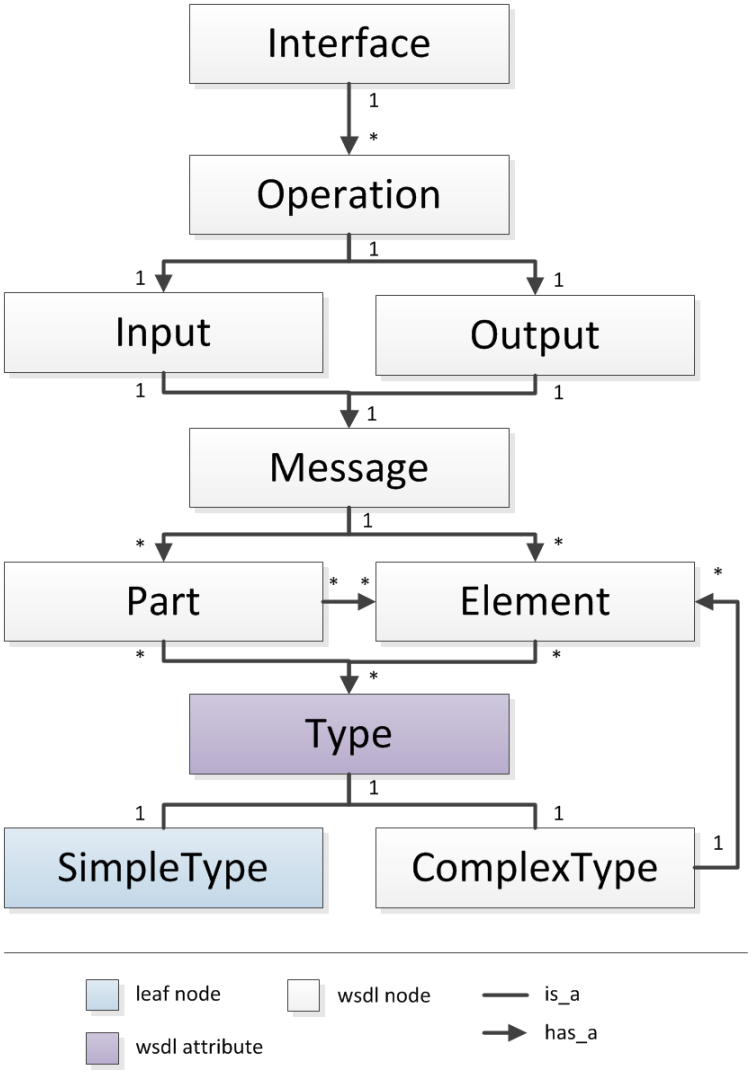
Introduction
- What is WSDL/SOAP
- WSDL Definition
- Alternatives
- Who the hell uses WSDL/SOAP in 2014?

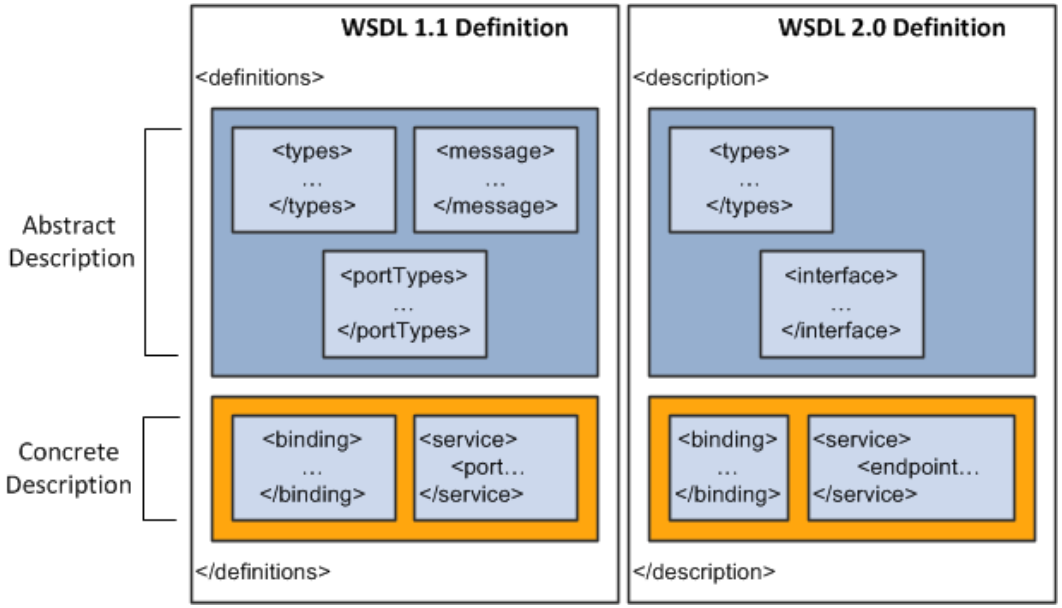
Tutorial
- Requirements
- WSDL
  - WSDL Service
  - WSDL Client
  - WSDL Remote Client
- Servlet
  - Set up Servlet
  - Test servlet with browser
  - WSDL to HTTP: Wrap WSDL Client in Servlet
  - Java Applications as (OS) Services

What is WSDL/SOAP

- WSDL is a syntactic (data-type) Description of Web Service APIs
  - Platform/language independent
  - SOAP is the exchange protocol
  - Both are XML-based
  - W3C Recommendation from 2001 - http://www.w3.org/TR/wsdl
- Wide-spread in research
  - due to well-defined syntactic modelling, it aids:
    - SwEng
    - Primary Web Service Discovery, Selection, Matching and Composition
    - Addition of semantics with OWL-S, WSMO, SAWSDL
- Already many tools automate WSDL/SOAP development
  - NetBeans, Eclipse, Visual Studio IDEs (JAX-WS library explored today)
  - SOAP UI testing tool
  - Language-specific libraries for ANY language/SDK

WSDL Definition





Alternatives
REST, Servlets

- Simple HTTP GET/POST Exchange
  - Undefined structure
  - Ad-hoc invocation and parsing
- Dominant solution in every current Cloud API
- We will also explore Servlet
  - development
  - usage
  - wrapping a WSDL into a Servlet

Who the hell uses WSDL/SOAP in 2014?

- WSDL/SOAP pitfalls
  - Still trickier than simple HTTP exchange
  - Widely used in industry (~2000-2010)
  - More and more disregarded: amazon, programmableweb
- Still has dedicated use cases
  - SwEng modelling/usage + Semantics
    - Exchange of Complex Types is easily supported
    - Poor alternatives for semantics e.g. hREST
    - HTTP (REST) APIs are handled in an ad-hoc manner
  - Industrial applications
  - R & D projects
- A lot to learn from it
  - e.g. Complex Type exchange

Tutorial

Requirements

- NetBeans IDE Web (Java EE) Edition 6.x – 8.x
- JDK 6.x – 8.x
- Glassfish Server or Tomcat server (needs adaptation)

- Basic Java knowledge recommended

Set up WSDL Server

- New Project -> Web Application “SensorServer”
- New class -> “SensorService”
- Add @WebService notation
- Add function getName
  - return System.getProperty("user.name");
- Add function getTemperature(String SensorID)
  - Return
- Add suggested target namespace and imports
- Automated method: add new Web Service

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

@WebService(targetNamespace = "http://my.org/ns/")
public class SensorService {

    @WebMethod
    public String getName() {
        return System.getProperty("user.name");
    }

    @WebMethod
    public double getTemperature(String sensorID) {
        if (sensorID.equals("151")) {
            //could get actual temp here
            return 25.6;
        } else {
            //could be a fault
            return -1.0;
        }
    }
}
```

Test WSDL Server

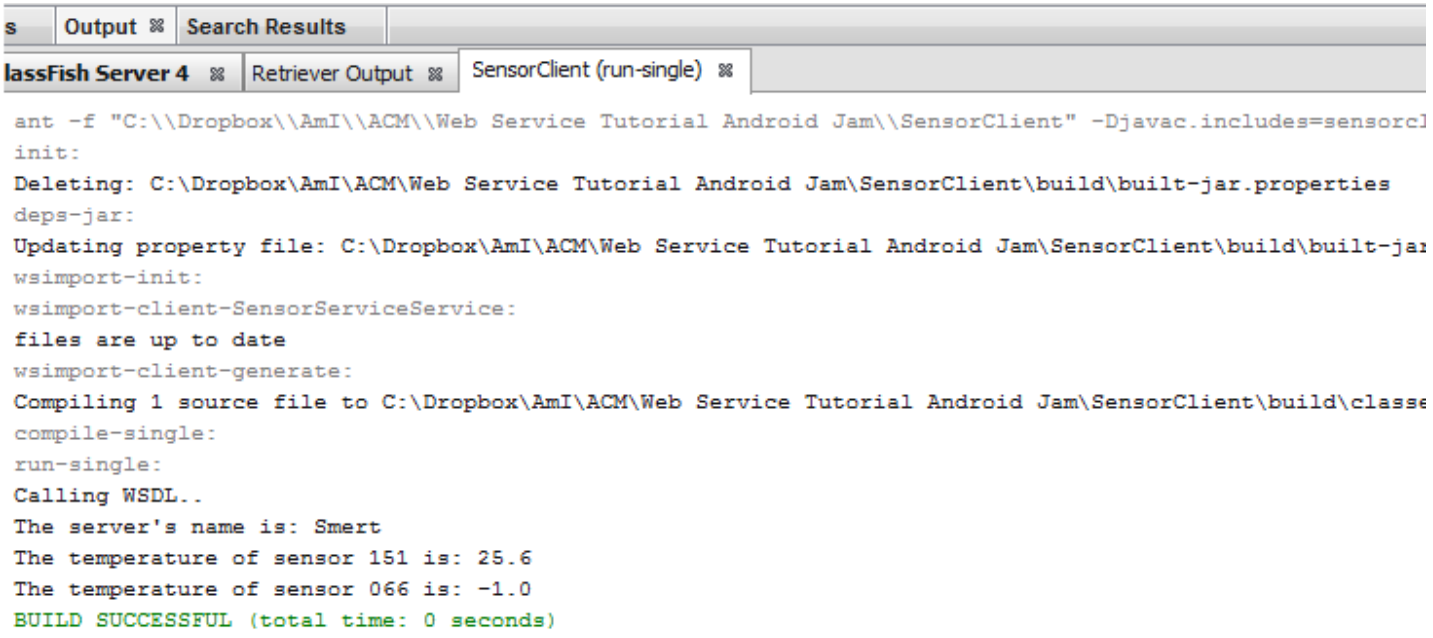
- Clean and Build, Run
- Open
  - http://localhost:8080/SensorServer/
    Project homepage (index.html)
  - http://localhost:8989/SensorServer/SensorServiceService
    Service Glassfish generated page
- Tester
  - http://localhost:8080/SensorServer/SensorServiceService?wsdl
- View generated WSDL
  - http://localhost:8080/SensorServer/SensorServiceService?wsdl

WSDL Client

- New Java (or any kind of) Application
- New Web Service Client
  - Enter WSDL URL http://localhost:8080/SensorServer/SensorServiceService?wsdl
  - Generally do not enter package! (Changes the namespaces)
- Drag'n'Drop service operations in SensorClient.java
- Test local service
  - Call getName, getTemperature from Main and print results

```
package sensorclient;

public class SensorClient {

    public static void main(String[] args) {
        System.out.println("Calling WSDL.. ");
        System.out.println("The server's name is: "+ getName());
        System.out.println("The temperature of sensor 151 is: "
                + getTemperature("151"));
        System.out.println("The temperature of sensor 066 is: "
                + getTemperature("066"));
    }

    private static String getName() {
        org.my.ns.SensorServiceService service =
                new org.my.ns.SensorServiceService();
        org.my.ns.SensorService port = service.getSensorServicePort();
        return port.getName();
    }

    private static double getTemperature(java.lang.String arg0) {
        org.my.ns.SensorServiceService service =
                new org.my.ns.SensorServiceService();
        org.my.ns.SensorService port = service.getSensorServicePort();
        return port.getTemperature(arg0);
    }
}
```

result


s
Output
Search Results
lassFish Server 4
Retriever Output
SensorClient (run-single)
ant -f "C:\\Dropbox\\AmI\\ACM\\Web Service Tutorial Android Jam\\SensorClient" -Djavac.includes=sensorcl
init:
Deleting: C:\Dropbox\AmI\ACM\Web Service Tutorial Android Jam\SensorClient\build\built-jar.properties
deps-jar:
Updating property file: C:\Dropbox\AmI\ACM\Web Service Tutorial Android Jam\SensorClient\build\built-jar
wsimport-init:
wsimport-client-SensorServiceService:
files are up to date
wsimport-client-generate:
Compiling 1 source file to C:\Dropbox\AmI\ACM\Web Service Tutorial Android Jam\SensorClient\build\classe
compile-single:
run-single:
Calling WSDL..
The server's name is: Smert
The temperature of sensor 151 is: 25.6
The temperature of sensor 066 is: -1.0
BUILD SUCCESSFUL (total time: 0 seconds)

WSDL Remote Client

- Two options:
  - Re-generate client with remote WSDL URL
  - Parameterize the cline in constructor (new URL)
    - // has to be the new WSDL
  - Add new URL("wsdl url"); in Service constructure
  - Catch Malformed URL Exception
    *some best practices

```
package sensorclient;

import java.net.MalformedURLException;
import java.net.URL;

public class SensorClient {

    private static String ServerURL = "http://localhost:8989/SensorServer/SensorServiceService";

    public static void main(String[] args) {
        System.out.println("Calling WSDL.. ");
        try {
            System.out.println("The server's name is: " + getName());
            System.out.println("The temperature of sensor 151 is: "
                    + getTemperature("151"));
            System.out.println("The temperature of sensor 066 is: "
                    + getTemperature("066"));
        } catch (MalformedURLException ex) {
            System.err.println("Error: Bad URL " + ServerURL);
            System.err.println(ex.getMessage());
        }
    }

    private static String getName() throws MalformedURLException {
        org.my.ns.SensorServiceService service
                = new org.my.ns.SensorServiceService(new URL(ServerURL));
        org.my.ns.SensorService port = service.getSensorServicePort();
        return port.getName();
    }

    private static double getTemperature(java.lang.String arg0) throws MalformedURLException {
        org.my.ns.SensorServiceService service
                = new org.my.ns.SensorServiceService(new URL(ServerURL));
        org.my.ns.SensorService port = service.getSensorServicePort();
        return port.getTemperature(arg0);
    }
}
```

Servlet

- New Project -> Web Application
- New Servlet -> SensorServlet
- Add methods
  - HTTPGet function
  - Choose context
  - Choose parameters
  - Edit web.xml
  - Add getName, getTemperature
- Clean and Build, Run
  - Test in any browser
- Print in HTML!

```
protected void processRequest(HttpServletRequest request, Ht  ervl  esponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        /* output your page here. You may use following sample code. */
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet SensorServlet</title>");
        out.println("</head>");
        out.println("<body>");
        //get user params
        String userPath = request.getServletPath();
        // if addToCart action is called
        if (userPath.equals("/getName")) {
            //http://localhost:8989/SensorServlet/getName
            //do stuff
            //print in HTML, JSON etc.
            out.println(System.getProperty("user.name"));
        } else if (userPath.equals("/getTemperature")) {
            String sensorID = request.getParameter("sensorID");
            //same logic, or call library
            if (sensorID.equals("151")) {
                //could get actual temp here
                out.println("25.6");
            } else {
                //could be a fault
                out.println("-1.0");
            }
        }
        //end doc
        out.println("</body>");
        out.println("</html>");
    }
}
```

Servlet vs Web Service Conclusion

- Pros
  - Very easily invoked with AJAX
  - Suitable to build backend applications
    - Security
    - Developer Collaboration
- Cons
  - Much more difficult to parse
  - Not even going to build it ☺

Servlet as an (OS) Service

- Bonus
  - Wrap any Java application
- Add methods context initialized / destroying
- Edit web.xml

Web.xml and OS Service

- Add in web.xml
- Under Servlet-mapping
  - <url-pattern>/getName</url-pattern>
  - <url-pattern>/getTemperature</url-pattern>

Service (OS)

- Under web-app
- <listener>
  <listener-class>
  package.AppStartUpClass
  </listener-class>
  </listener>

Add

- Implements ServletContextListener

```
@Override
public void
contextDestroyed(ServletContextEve
nt arg0) {
      // Do cleanup operations here
 }
```

```
 @Override
 public void
 contextInitialized(ServletContextEvent
 arg0) {
            // Invoke the
 daemon/background process code
 }
```

Extensions

- Wrap WSDL in Servlet
- JAXB for ComplexTypes
- Invoke with AJAX from Web App
- Full event to follow at the IHU!
  - You will call real sensors and actuators

Thank you

Follow us!

FB: ACM Student Chapter AUTH