Designing, setting up and accessing Web Services

Martin Senger
martin.senger@gmail.com
A starting point...

- Google is full of Web Services; perhaps too full...
- An “official site” is perhaps:
  - http://www.w3.org/2002/ws/
- The best starting point is a question:
  - “Do I need it?” “How Web Services can solve my problem?”
  - Which leads us to the distributed architecture...
Distributed architecture

• **gives**
  – access to distributed resources
  – development encapsulation
    • maintainability, re-usability, legacy-awareness
  – implementation independence

• **requires**
  – adding a communication layer between parts
  – synchronization of efforts
    • including such nasty things as distributed garbage collection
Distributed architecture

- **Basic questions are:**
  - What kind of protocol to use, and what data to transmit
  - What to do with requests on the server side
Traditional CGI-based approach

- **cgi-bin scripts:**
  - Data transmitted as name-value pairs (HTML forms)
  - Transport over (state-less) HTTP protocol
  - No standards for keeping user sessions (state-fullness)
  - Server side: a script is called
CORBA-based approach

- **CORBA:**
  - Data transmitted as objects (at least it looks like that)
  - Transport (usually) over well standardised IIOP protocol
  - User sessions (state-fullness) very inter-operable
  - Server side: an RPC call is made
Many more approaches...

• Direct access to data sources (JDBC, DBI,…)
• VPN (Virtual Private Network)
• Various shell programs
• p2p networks (Jaxta,…)
• And perhaps some more…
  – ….but let’s jump to the one we are interested the most at the moment:
SOAP-based communication

- **SOAP:**
  - Data in a well-defined XML format
  - Transport over various protocols
    - HTTP, SMTP are the most used, perhaps because they are firewall-friendly
  - server side: either an RPC call or a message delivered
W3C (working group) definition

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/
Web Services Architecture

Let a program “click on a web page”
SOAP

- **Simple Object Access Protocol**
  - [http://www.w3c.org/TR/SOAP/](http://www.w3c.org/TR/SOAP/)

- A lightweight protocol for exchange of information in a decentralised, distributed environment

- Two different styles to use:
  - to encapsulate RPC calls using the extensibility and flexibility of XML
  - ...or to deliver a whole document without any method calls encapsulated
What is inside SOAP

• SOAP is an XML based protocol that consists of three parts
  – an envelope that defines a framework for describing what is in a message and how to process it
  – a set of encoding rules for expressing instances of application-defined datatypes
  – a convention for representing remote procedure calls and responses
XML Messaging Using SOAP

1. Service Requestor
   - Application
   - SOAP
   - Network Protocol

2. Service Provider
   - Application
   - SOAP
   - Network Protocol

3. Response
4. Request
   (service invocation)
WSDL

- **Web Services Definition Language**
  - [http://www.w3.org/TR/wsdl/](http://www.w3.org/TR/wsdl/)
- An XML-based language for describing Web Services
  - what the service does (description)
  - how to use it (method signatures)
  - where to find the service
- It *does not* depend on the underlying protocol
- But: It is not much human-readable
Hello.wsdl

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://localhost:8080/axis/services/Hello"
    xmlns:impl="http://localhost:8080/axis/services/Hello-impl"
    xmlns:intf="http://localhost:8080/axis/services/Hello"
    xmlns:soap="http://schemas.xmlsoap.org/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:service name="HelloWorldService">
    <wsdl:port binding="intf:HelloSoapBinding">
      <wsdl:operation name="getHelloMessage">
        <wsdl:input message="intf:getHelloMessageRequest"/>
        <wsdl:output message="intf:getHelloMessageResponse"/>
      </wsdl:operation>
      <wsdl:operation name="setHelloMessage">
        <wsdl:input message="intf:setHelloMessageRequest" parameterOrder="in0"/>
        <wsdl:output message="intf:setHelloMessageResponse"/>
      </wsdl:operation>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
UDDI (and alternatives)

- Universal Description, Discovery and Integration
  - http://www.uddi.org

- UDDI creates a platform-independent, open framework & registry for:
  - Describing services
  - Discovering businesses
  - Integrating business services

- The UDDI may be less used than predicted, especially on the Internet level

- BioMoby - an alternative for Life Sciences domain?
MOBY hosts & services

- Gene names
- Sequence
- Align
- Phylogeny
- Primers
- Protein
- Alleles
- Sequence Express.
A Web Service example in Java

HTTP Server

Servlet engine (e.g. Apache Tomcat)

SOAP-aware Servlet
(usually: ApacheAxis)

Any class processing the incoming requests ("business logic")

Sending requests, getting results
Steps to develop a WS in Java

1. **Write your service implementation**
2. **Make all your classes available to the toolkit**
3. **Deploy your service** *(usually done just once)*
4. **Restart the whole servlet engine**
5. **Test it with a client request**
Java Web Services Toolkit

• More of them, but The One is:
  – Apache Axis: http://ws.apache.org/axis/

• Principles:
  – Writing server is easier than writing clients (but only regarding the toolkit, not the business logic)
  – Servers may be written independently on the used toolkit
  – Always test interoperability with a non-Java client (because of data serialization and de-serialization)
package hello;
public interface HelloWorld {
    String getHelloMessage();
    void setHelloMessage (String newHello);
}

group: hello

package hello;
public class HelloWorldService implements HelloWorld {
    String message = "Hello, world!";
    public String getHelloMessage() {
        return message;
    }
    public void setHelloMessage (String newMessage) {
        message = newMessage;
    }
}
import org.apache.axis.client.*;
public class HelloWorldClient {
    public static void main (String [] args) {
        try {
            // prepare the call (the same for all called methods)
            Call call = (Call) new Service().createCall();
            call.setTargetEndpointAddress
            (new java.net.URL("http://localhost:8080/axis/services/Hello"));

            // call "get message"
            if (args.length == 0) {
                call.setOperationName ("getHelloMessage");
                String result = (String) call.invoke ( new Object [] {} );
                System.out.println (result);
                System.exit (0);
            }

            // call "set message" and afterwards "get message"
            call.setMaintainSession (true);   // TRY also without this line...
            call.setOperationName ("setHelloMessage");
            call.invoke ( new Object [] { args[0] } );
            call.setOperationName ("getHelloMessage");
            System.out.println (call.invoke ( new Object [] {} ));
        } catch (Exception e) {
            System.err.println ("ERROR:
            + e.toString());
        }
    }
}
HelloWorldService

HelloWorldServiceLocator

HelloWorld

HelloSoapBindingStub

1. Make an instance of this

2. Use it to make an instance of this

3. Call methods on this proxy object

getHello()
public class HelloWorldClientFromStubs {
    public static void main (String [] args) {
        try {
            // prepare the calls (the same for all called methods)
            hello.generated.HelloWorldService service =
                new hello.generated.HelloWorldServiceLocator();
            hello.generated.HelloWorld myHelloProxy = service.getHello();

            // call "get message"
            if (args.length == 0) {
                String result = myHelloProxy.getHelloMessage();
                System.out.println (result);
                System.exit (0);
            }

            // call "set message" and afterwards "get message"
            myHelloProxy.setHelloMessage (args[0]);
            System.out.println (myHelloProxy.getHelloMessage());
        } catch (Exception e) {
            System.err.println ("ERROR:
            + e.toString());
        }
    }
}
Java <=> XML Data Mapping

• How Java objects are converted to/from XML data (in order to be able to be put into SOAP messages)

• Important especially for the non-basic data types

• It’s easier if your non-basic data types are Java Beans (having set/get methods for members)
A Web Service example in Perl

package HelloPerl;
use strict;
use vars qw( $Message );
$Message = 'Hello, here is Perl.';
sub getHelloMessage { $Message; }
sub setHelloMessage { $Message = shift; } 1;

#!/usr/bin/perl -w
use SOAP::Transport::HTTP;
SOAP::Transport::HTTP::CGI
    -> dispatch_to('HelloPerl')
    -> handle;

#!/usr/bin/perl -w
use SOAP::Lite
    on_fault => sub {...};
print SOAP::Lite
    -> uri ('HelloPerl')
    -> proxy ('http://localhost/cgi-bin/helloserver.cgi')
    -> getHelloMessage
    -> result;
SOAP::Lite

- a collection of (many) modules
  - but they are loaded automatically when needed
- supports SOAP 1.1 specification
- all methods can be used for both setting and retrieving values:
  - if you provide no parameters, you will get current value, and if parameters are provided, a new value will be assigned to the object
  - and the method in question will return the current object (if not stated otherwise) which is suitable for stacking these calls like:

```perl
$lite = SOAP::Lite
    -> uri('openBQS')
    -> proxy('http://industry.ebi.ac.uk/soap/openBQS')
;
```
Using “wsdl” - directly

- getting “.wsdl” file by using its URL
- then, you do not need to worry about autotyping

#!/usr/bin/perl -w

use SOAP::Lite on_fault => sub {...};
print SOAP::Lite
    -> service ('file:/home/senger/ws-ws/perl/Hello.wsdl')
    -> setHelloMessage (123);

#!/usr/bin/perl -w

use SOAP::Lite on_fault => sub {...};
my $service = SOAP::Lite -> service ('file:./Hello.wsdl');
$service->setHelloMessage ($ARGV[0] or "Hello!!");
print $service->getHelloMessage, "\n";
Why to use Web Services...

• WS are easier to deploy because of their firewall-friendliness
• WS are quite well marketed (both from IT companies and Open Source projects)
• It well integrates into workflows (hence this tutorial at NETTAB!)
• The programming effort and maintainability is similar to other distributed technologies
But be aware that…

• Client is different from server; both have different resources and restrictions
  – notification by “server-push” is harder to achieve
• User sessions are less standardised; there is no real standard:
  – HTTP level (cookies)
  – SOAP headers
  – Within your implementation code
  – but: Web Services Resource Framework (WSRF) and Web Services Notifications (WSN) is perhaps an incoming standards for state-full web services
Designing principles

• Make it simple
  – use data structures that are understood by many languages/toolkits
  – avoid (if possible) the need for specialized [de]serializers

• Follow standards
  – allow to use WSDL (if possible)
  – try to localize dependency on a used toolkit/servlet engine

• Test the interoperability
  – between languages
  – think about different encodings