Groovy and Web Services

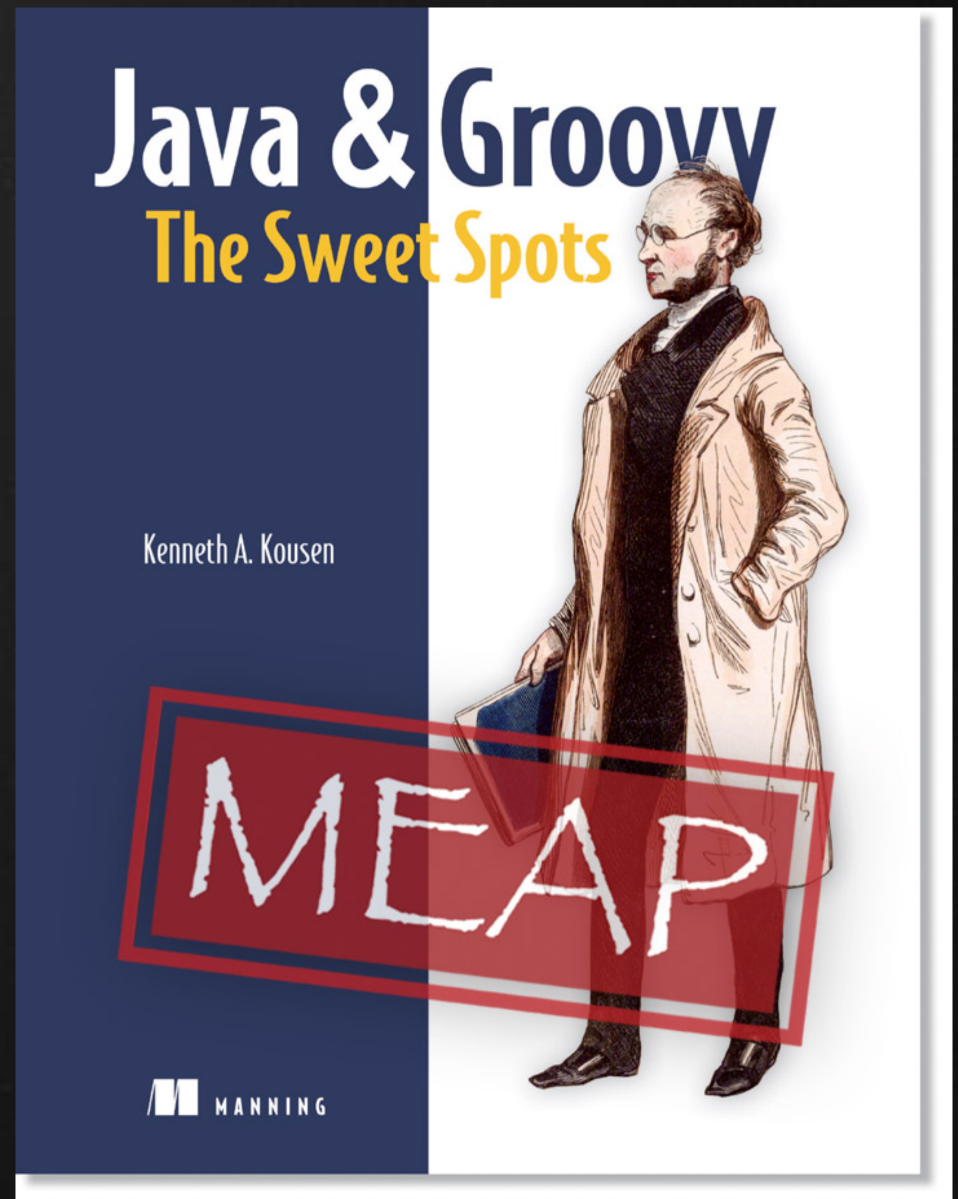
Ken Kousen
Kousen IT, Inc.
ken.kousen@kousenit.com
http://www.kousenit.com

Who Am I?

Ken Kousen
Trainer, Consultant, Developer
ken.kousen@kousenit.com
http://www.kousenit.com
@kenkousen

SCJP, SCWCD, SCBCD, ...
Several academic degrees

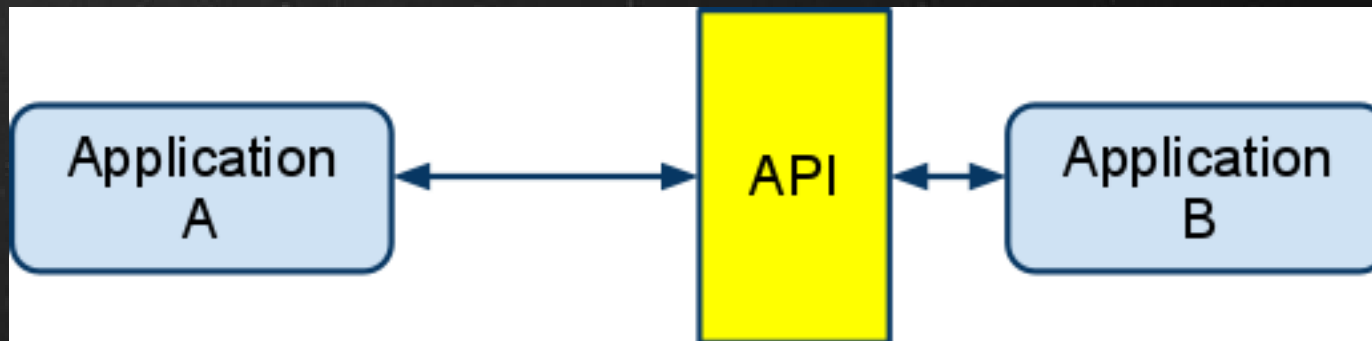
Working with Java since 1996



Web Services

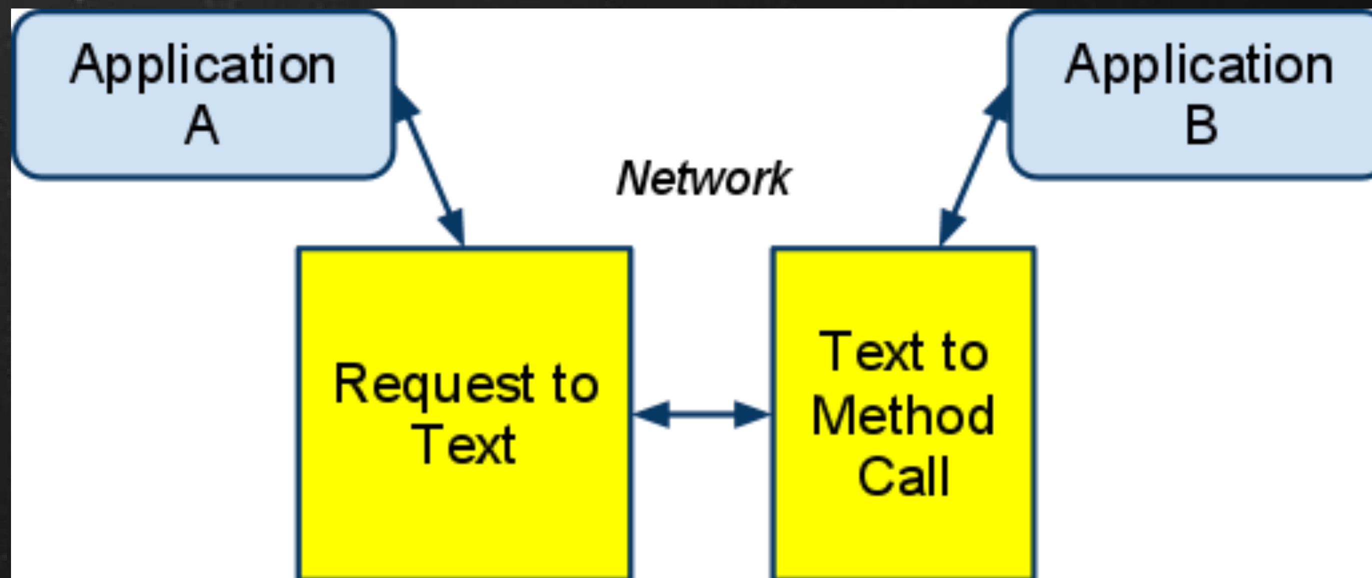
Integration technique

- Connect programs using API



Web Services

- API not available
  - Different language, different platform, ..
  - Convert method call to text and back



Web Services

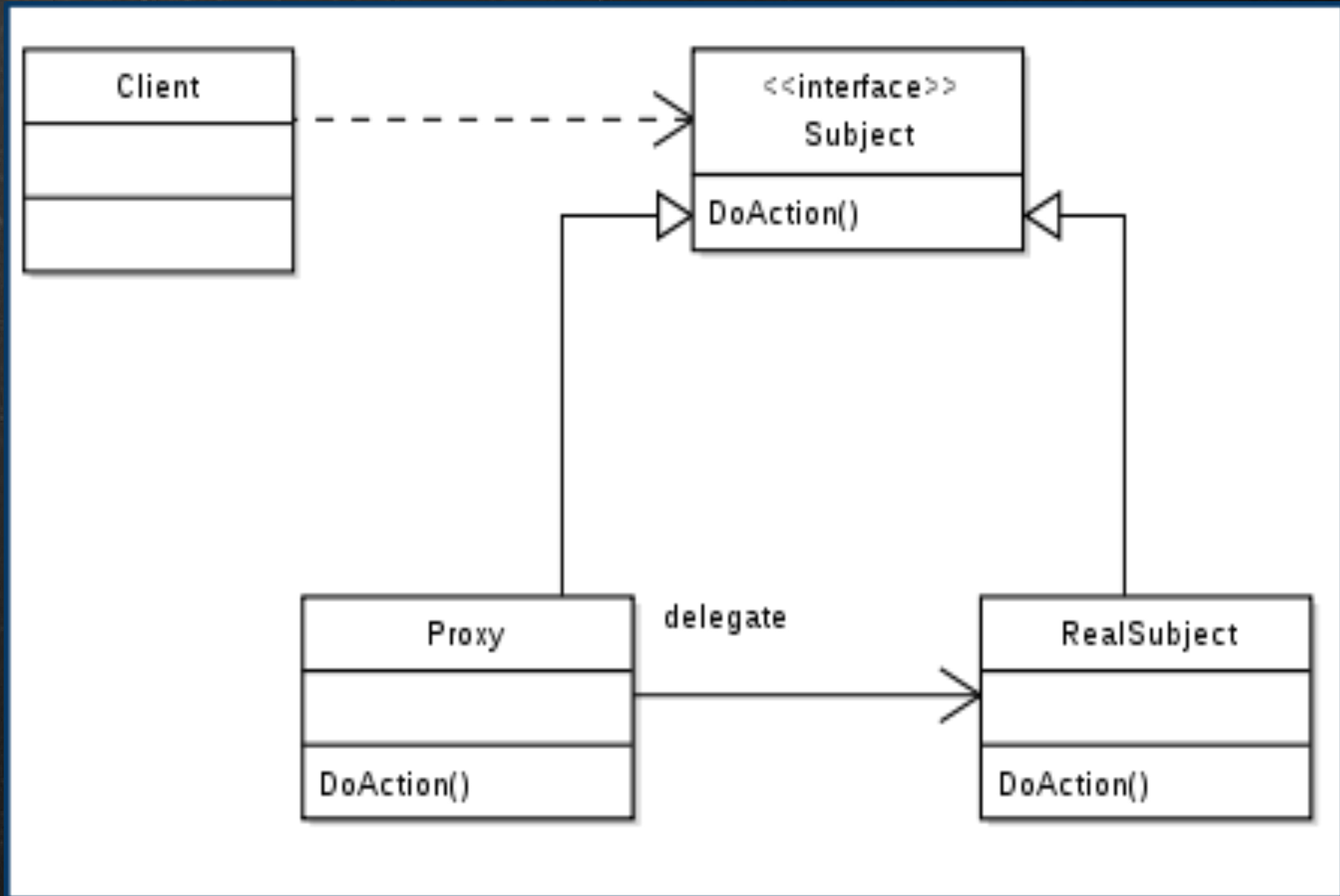
A gets reference to B proxy

- Invoke method on B proxy

- Use Proxy to serialize arguments
- Transmit text over network to B
- Use Proxy to convert text to method call
- Invoke method on B

- Reverse process with return value

Proxy Design Pattern



http://en.wikipedia.org/wiki/Proxy_design_pattern

Two Flavors

- SOAP-based web services
  - Use SOAP wrapper for payload
  - Makes header elements available
  - Lots of automatic code generation
- RESTful web services
  - Everything is an addressable resource
  - Leverage HTTP transport
  - Uniform interface (GET, POST, PUT, DELETE)

RESTful Web Services

REST -> Representational State Transfer

Term coined by Roy Fielding in Ph.D. thesis

Architectural Styles and Design of Network-based Software Architectures

Idempotent

Repeated requests give same result

GET, PUT, DELETE are all supposed to be idempotent

POST is not

Safe

Requests do not change the state of the server

GET requests are both idempotent and safe

Content Negotiation

Each resource can have multiple representations

1. XML
2. JSON

...

Request will state which representation it wants

RESTful Client

1. Assemble URL with query string
2. Select the HTTP Request verb
3. Select the content type
4. Transmit request
5. Process results

Example: Google Geocoder

The Google Geocoding API

Convert address information to latitude, longitude

Base URL is

http://maps.googleapis.com/maps/api/geocode/output?parameters

where output is xml or json

parameters include

- URL encoded address
- sensor true/false

Geocoder Client

Groovy makes it easy to assemble a query string

- Map of parameters:
    - [key1:value1, key2:value2, ..., keyN:valueN]
- Convert to query string using collect and join
    - `map.collect { k,v -> "$k=$v" }.join('&')`
- results in
    - key1=value1&key2=value2&...&keyN=valueN

Geocoder Client

Groovy makes it easy to parse resulting XML

```
def response = new XmlSlurper().parse(url)
```

Just walk the XML tree to get data

```
latitude = response.result.geometry.location.lat
longitude = response.result.geometry.location.lng
```

Geocoder Client

Content negotiation is based on URL

- http://.../xml for XML response
- http://.../json for JSON response

- Note only GET requests supported
  - Not adding, updating, or deleting information

- Very effective use case for REST
  - Simple, read-only, look-up processing

Twitter API

- Closer to true REST
    - Twitter API at http://dev.twitter.com/doc

Downside to REST

No WSDL

therefore, no proxy generation tools

WADL : Web Application Description Language

attempt to create WSDL for REST

slowly gaining acceptance, but not common yet

RESTful Web Services

In Java, there is JSR311

- JSR = Java Specification Request
- JCP = Java Community Process http://jcp.org
  - that's how Java grows

Currently, JAX-WS is part of Java EE 5

- but also part of Java SE 1.6

Proposal for JAX-RS

- part of Java EE 6

Generated Proxies

Client proxy usually called stub

Server proxy usually called tie (old CORBA term)

Don't want to write them
--> generate them

Java Tools

Java has tools for SOAP-based proxy generation:

- wsimport --> generate client proxy
- wsgen --> generate server ties
- Part of Metro project, https://metro.dev.java.net/
  - Built into JDK 1.6

What about Groovy?

Java and Groovy together principle:

Java is good for tools, libraries, and infrastructure

Groovy is good for everything else

So let Java do what it does best --> generate stubs
Use Groovy for client, service implementation and more

Global Weather Service

Microsoft, publicly-available web service

http://www.webservicex.net/WCF/ServiceDetails.aspx?SID=48

Service described by WSDL file

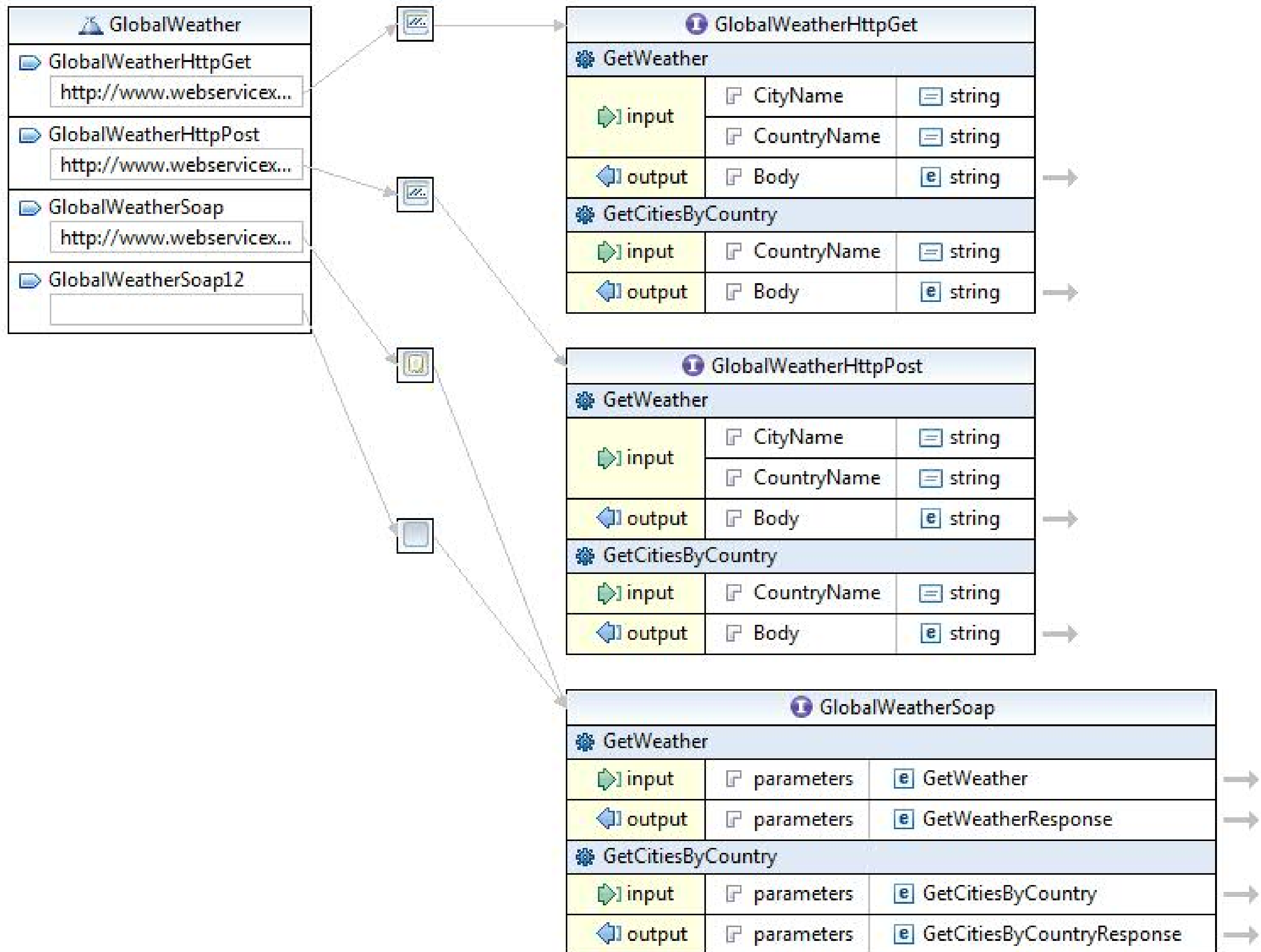
Web Services Definition Language

--> operations

--> arguments and return types

--> binding (how messages are formed and transmitted)

--> location of service


GlobalWeather
GlobalWeatherHttpGet
http://www.webservicex...
GlobalWeatherHttpPost
http://www.webservicex...
GlobalWeatherSoap
http://www.webservicex...
GlobalWeatherSoap12
GlobalWeatherHttpGet
GetWeather
input
CityName
string
CountryName
string
output
Body
string
GetCitiesByCountry
input
CountryName
string
output
Body
string
GlobalWeatherHttpPost
GetWeather
input
CityName
string
CountryName
string
output
Body
string
GetCitiesByCountry
input
CountryName
string
output
Body
string
GlobalWeatherSoap
GetWeather
input
parameters
GetWeather
output
parameters
GetWeatherResponse
GetCitiesByCountry
input
parameters
GetCitiesByCountry
output
parameters
GetCitiesByCountryResponse

Key Sections

From WSDL file:

- portType --> Interface
- schema --> arguments and return types
- binding --> SOAP/HTTP,
  document, literal, wrapped
  (like all these days)
- service --> endpoint location

Global Weather WSDL

GlobalWeatherSoap --> interface
getWeather(city,country) --> operation

Can generate everything from WSDL

```
wsimport -d bin -s src -keep
    http://www.webservicex.net/globalweather.asmx?WSDL
```

Add "?WSDL" to endpoint to access WSDL file...
generates all client stubs

Groovy Client

Get stub traditional way:

```
<portType> stub = new <service>().get<port>()

GlobalWeatherSoap stub =
    new GlobalWeather().getGlobalWeatherSoap()

stub.getWeather 'Grenoble','France'
```

Returns XML --> easy to process with Groovy

Global Weather Response

<?xml version="1.0" encoding="utf-16"?>
<CurrentWeather>
<Location>Grenoble / St. Geoirs, France </Location>
<Time>..</Time>
<Wind> from the E (080 degrees) at 3 MPH (3KT):
0</Wind>
<Temperature> 48 F (9 C)</Temperature>
<DewPoint> 48 F (9 C)</DewPoint>
<RelativeHumidity> 100%</RelativeHumidity>
<Pressure> 29.85 in. Hg (1011 hPa)</Pressure>
<Status>Success</Status>
</CurrentWeather>

Groovy XmlSlurper

Parse response using XmlSlurper

```
def root = new XmlSlurper().
        parseText(stub.getWeather(city,country))
println "Temp in $city, $country is ${root.Temperature}"
```

--> Temperature in Grenoble, France is 48 F (9 C)

Accessing data is as easy as walking the tree...

Web Service Client

Summary:

- Let Java generate stubs using its tools (wsimport)
- Use Groovy to parse XML response

Build a Web Service

Clients always generated from WSDL

Service from pre-existing WSDL
    generate implementation "skeleton"
        --> top down

Service can be pre-built, and exposed as web service
        --> bottom up

Bottom Up Web Service

The "Hello, World!" application for web services is the World's Slowest Calculator

Create Calculator class to add, multiply, etc.

Expose methods as web service

Generate client as before

Add numbers by generating SOAP, transmitting over HTTP, etc.

JAX-WS

Java API for XML-based Web Services

- Uses annotations to specify metadata about service
  - @WebService, @WebParam, @WebResult, ...
- Uses JAXB (Java API for XML Binding)
  - to translate schema types to Java classes and back
- Provides wsgen tool for proxy generation

JAX-WS

One quirk, which is presumably supposed to be a feature:

- JAX-WS does NOT require an interface
  - Operates on compiled SIB

- SEI = Service Endpoint Interface
- SIB = Service Implementation Bean

But wsgen tool assumes

- SEI = Service Endpoint Implementation
  - (which is just evil, but so be it)

Java Calculator without SEI

```
@WebService
public class Calculator {
@WebResult(name="sum")
public double add(
@WebParam(name="x") double x,
@WebParam(name="y") double y) {
return x+y;
}
        ...
}
```

Generate Server Stubs

Use wsgen tool

```
wsgen -d bin -s src -r resources -wsdl -keep
    -cp bin jag.calc.service.Calculator
```

- Generates Java code
  - compiled in bin
  - source in src
  - wsdl in resources

Trivial Deployment

For testing, JDK 1.6 includes very convenient java.xml.ws.Endpoint class

Single-threaded server that deploys web services

```
Endpoint.publish("...URL endpoint...", new SIB())
```

Calculator Client

While server is running, generate stubs based on deployed WSDL file

Everything works as before

What about Groovy?

Annotations on SIB run into a problem
- to serialize arguments, tool searches for properties
- Groovy classes have a getMetaClass() method

Tool fails because it doesn't know
- how to serialize metaClass object

Solution --> separate interface from implementation
- (good idea anyway)

Java Interface

```
@WebService
public interface Calculator {
@WebResult(name="sum")
double add(
@WebParam(name="x") double x,
@WebParam(name="y") double y);

        ...
}
```

Groovy Implementation

```
@WebService(
  endpointInterface="jag.calc.service.Calculator")
class GroovyCalculator implements Calculator {
double add(double x, double y) { x + y }
    ...
}
```

wsgen still operates on SIB (GroovyCalculator)

but everything works

Issue

Since wsgen operates on SIB
- name of SIB embedded in WSDL file

  Calculator calc =
    new GroovyCalculatorService().groovyCalculatorPort

  Client shouldn't be exposed to SIB name
    that's an implementation detail

Enter @Delegate

@Delegate AST Transformation

Allows class to delegate all calls to a contained property

```
@WebService(
    endpointInterface="jag.calc.service.Calculator")
class GenericCalculator {
    @Delegate Calculator calc
}
```

Multiple SIBs

Generate proxies from GenericCalculator

When running server, specify implementation

```
Calculator calc =
  new GenericCalculator(calc:new JavaCalculator())
Endpoint.publish "http://localhost:1234/calc", calc
```

or set calc property to new GroovyCalculator()

Client code based on GenericCalculator only

```
Calculator c =
    new GenericCalculatorService().genericCalculatorPort
```

Calculator Summary

Separate interface from implementation

- Interfaces allow Java tools to work
- Groovy implementations use Groovy capabilities

Groovy @Delegate AST Transformation

- makes it easy to generalize implementations

All we did was pass doubles around;
what if our domain model is more interesting?

Potter's Potions

```
@WebService
interface Wizard {
@WebResult(name="potion")
Potion brewPotion(
        @WebParam(name="cauldron") Cauldron c);
}
```

Supply cauldron of ingredients to service, returns brewed potion

Potion class (Java)

```java
public class Potion {
private String name;
private String effect;
private Date expiration;
private double probability;
private List<Ingredient> ingredients;

	// ... getters and setters, ctors, toString, ...
}
```

Cauldron class (Java)

```
public class Cauldron {
@XmlElementWrapper(name="ingredients")
@XmlElement(name="ingredient")
private List<Ingredient> ingredients =
        new ArrayList<Ingredient>();

    // ... getters and setters ...
}
```

Annotations used to make XML look nice

<ingredients><ingredient>...</ingredient></ingredients>

Ingredient class (Groovy)

Potion, Cauldron in Java --> no problems

Ingredient in Groovy --> same metaClass issue as before

Tell JAXB to use field access rather than getters

```
@XmlAccessorType(XmlAccessType.FIELD)
class Ingredient {
String name
double amount
String units
}
```

Groovy SIB

```
class Granger implements Wizard {
Potion brewPotion(Cauldron c) {
Potion p = new Potion(name:'polyjuice',
                effect:'look like someone else',
expiration:new Date() + 1, probability:0.8);
p.ingredients = c.ingredients;
return p
}
}
```

Add ingredients from cauldron to potion and brew...

@Delegate Again

```
@WebService(endpointInterface="jag.pp.service.Wizard")
class HogwartsWizard {
@Delegate Wizard wizard
}
```

Specify Wizard implementation at run time

```
Wizard wiz = new HogwartsWizard(wizard:new Granger())
Endpoint e = Endpoint.publish(
    "http://localhost:1234/wizard", wiz)
```

Wizard Client

Generate stubs using wsimport as before

Dynamic proxy client this time (just for fun):

```
Service service = Service.create(
new URL('http://localhost:1234/wizard?WSDL'),
new QName('http://service.pp.jag/','HogwartsWizardService'))
Wizard w = service.getPort(
new QName('http://service.pp.jag/','HogwartsWizardPort'),
Wizard.class)
Cauldron c = new Cauldron()
c << new Ingredient(name:'sopophorous bean',amount:1,units:'bean')
c << new Ingredient(name:'gillyleaf',amount:1,units:'handful')
Potion p = w.brewPotion(c)
```

Problem

What if our client sends us ingredients that can be used for nefarious purposes?

Need a pre-processor to check for potential trouble

Handlers

Benefit of tool generators is no exposed XML

SOAP message details hidden under the hood

But here we need to examine XML directly

Handlers --> pre- and post-processors for web services

Handlers

For direct SOAP manipulation
- Java has I/O streams (ugh)
  - treat message like strings
- or DOM parsing (better, but ugh)
  - treat message like XML
- or SAAJ API (better still, but not fun)
  - treat message like SOAP

Groovy can do better, as we'll see

Java interface for Handlers

```
public class CauldronHandler implements
        SOAPHandler<SOAPMessageContext> {

@Override
public boolean handleMessage(
        SOAPMessageContext context) {
        // process SOAP message directly here
    }

    // handleFault, close, getHeaders, ...
}
```

Groovy handler

Use Java handler for tools

- It can use Groovy implementation for XML:

```
public class CauldronHandler implements ... {
    private CauldronProcessor cp;
    private boolean darkArtsDetected;
    ...
```

- Pass SOAPBody to Groovy CauldronProcessor
  - Why? Groovy DOMCategory for Element

Handle Incoming Message

```
public boolean handleMessage(
    SOAPMessageContext context) {
        // if incoming message ...
SOAPBody body =
            context.getMessage().getSOAPBody();
cp.setBody(body);
darkArtsDetected = cp.detectDarkArts();
...
}
```

Handle Outgoing Message

```
if (darkArtsDetected) {
    try {
cp.setBody(context.getMessage().getSOAPBody());
cp.addDisclaimer();
    } catch (SOAPException e) {
        e.printStackTrace();
    }
}
```

Groovy Processor

```
class CauldronProcessor {
Element body

boolean detectDarkArts() { use (DOMCategory) { return
body.'**'.any { it =~ /.*unicorn.*/ } } }

    // DOMCategory adds convenient methods for
    // searching DOM tree (SOAP body)
```

Groovy Processor

```
def addDisclaimer() {
use (DOMCategory) {
def e = body.'**'.find { it.name() == 'effect' }
e.replaceNode {
effect '''Possible dark arts detected.
                         No warranty expressed or implied'''
}
}
}
```

// Use builder syntax to replace effect node

Conclusions

Java good for tools, libraries, and infrastructure
Groovy good for everything else (especially XML)

Separate interface from implementation to help tools

Use @Delegate to hide SIB class from client

DOMCategory adds convenient methods to DOM trees
    Use when direct XML manipulation necessary