A Practitioner's approach to modelling Resources and Services in cloud based Telecom

(A FREE offering from http://ossnext.org *read our disclaimer[*])*

Decide whether this is worth your read before you dig further:

Intended audience:

IT Architects and Designers, CIOs, VPs, Directors and PMs entrusted with strategic transformation in their enterprise.

Some of the benefits of the approach here may appear as clichés you found elsewhere; sadly, they were probably only claims not backed by proof or demonstration. We will actually demonstrate some benefits that come from a 'Practitioner Approach', i.e. applying strategic design/architecture patterns. This tutorial and its accompanying software framework complete with a working implementation are FREE.

New to Information and Communications Technology (ICT) sector?

Modelling Telecom Resources such as network elements and Services such as the internet or wireless service is a universal requirement in Information and Communication Technology industry. IT systems carry out several business functions by operating on these models. Example of business functions are managing Quality of Services (QoS), monitoring Resource availability (e.g. network element outage), selling and fulfilling Telecom Services for customers e.g. the Home Internet Service. Thus, IT systems need models or software representation of these real life Resources and Services.

Let's get to the point

In today's cloud and SOA (Service Oriented Architecture) paradigm, we can expect to see increased sourcing of Telecom Resources and Services via web services, because web services offer cost effectiveness through reusability and economies of scale and provide ubiquity and technology standardization. IT Architects and Designers have before them a huge opportunity – and some challenges – to transform legacies and silos of Telecom enterprises. CIOs love this story, because it allows to retro-fit generations of IT systems worth millions of $ for retention in the cloud. Thus, the fastest most cost-effective transformation to cloud and web services remains a strong competitive advantage.

Why the CIO will love your story after this tutorial -

-- Transform to win: learn key patterns to make Telecom Resources and Services ubiquitously and cost-effectively available to IT - a key demand of the cloud paradigm

-- Protect IT investment – build once, use often: avoid some teething issues that defeat reusability and rapid development

-- Make sense of competing architecture paradigms SOA and ROA (Resource Oriented Architecture); see and find extents to which each should be applied for optimal benefit

Issues with state of art that we care about

Software model of a Telecom service is typically hierarchical. E.g., the "Home Internet Service" can be modelled as a Java Business Object. This Java Object is composed of a hierarchy of other Java Objects representing resources (network elements) such as "DSL Port", "DSL Line", "DSL Modem", "Dynamic IP Address" and "MAC Address".

The tendency of current designs is to create Service Oriented Architecture (SOA) pattern to model Services. In our example above, a single SOA "Service" is probably going to be exposed to represent the Home Internet Service. Let's call this SOA Service "S1". Let us assume our web service S1 is implemented using the common Java/XML/HTTP based RPC web service standard, JAX-WS. Being a Service, S1 would provide "Business Services" identified by verbs such as "Calculate Service Availability", "Assign Service to Customer", "Activate Service for Customer". That's all good.

Now, what if you needed to manipulate Resources that an instance of S1 is composed of? E.g., you want to monitor S1's DSL Port Resource over a period of time, or update the Resource. There could be two options to do the Update:

1. Taking the "access by value" OO analogy, you would probably implement a getter method on S1 to obtain the DSL Port Resource. Then, you'd update the Resource object and set it back in S1 via a setter method
2. Going the "access by reference" way, you'd probably get a web service url reference to the DSL Port Resource, then Post the updated Resource object via the url

Taking Option 1 apart

In Option 1 above, a Resource getter and a setter method may be simple to implement on S1. However, this creates a dependency on S1 just for manipulating the Resource. Such getter and setters don't fit well with the semantic of a Service which is to provide Business relevant functionality (Business Logic, Flow, Rules, etc.). If your web service client wants to purely operate on the Resources, the client should not have to go via S1 just to access the Resources. This is also an improper overhead on S1: imagine following scenario:

In order to monitor the DSL Port Resource associated with the instance of S1 over a period of time, the client needs to obtain 100 snapshots of the DSL Port Resource instance. Thus, the client needs to make

100 getter calls on S1. Can you see where we are going with this? S1's "day job" is to provide "Business Services", not get bogged down with 100 Resource getter requests. So, clearly, for clients that are only going to operate on S1's Resources and not really access S1's Business Services, S1's SOA design is not an optimal choice. You might find that there is no current need for "Resource only" clients, and that S1 seems to fit nicely into your current service requirements. However, such requirements can easily arise in future, possibly creating an impact to S1. Establishing volumetric requirements for S1 can be tricky in this case, since it begins with one set of capacity requirements for its Business Services, and then has to cater to Resource access volumes. Resource access volumes tend to be larger than those for Service access, so it can create an imbalance in the utilization of S1. So, on to Option 2

Can Option 2 rescue?

In Option 2, S1 provides a reference to each of its underlying Resource to a requesting client. The client can then directly access a Resource via its reference, essentially bypassing S1 and freeing it up. In implementation terms, each Resource reference is a WSDL describing both the url to the Resource as well as the associated Resource access methods. Thus, every Resource url is that of a web service for that Resource. To access a Resource, the client will have to integrate the Resource's web service at build time and bind to its url during runtime.

But now we are faced with a problem of a different kind. Does this approach require us to build a web service for every Resource in the enterprise? If there are 10,000 Resources, we are certainly not going to build 10,000 web services just for the sake of Resource access! Compare a web service to a data source for database access. We don't build a data source for every table; we build *one* data source for a database schema, and access every table inside that schema using that single data source. The need is to build *one* web service that can cater to every Resource access.

In terms of semantics, this is where we depart from SOA, and head into the realm of ROA or Resource Oriented Architecture. ROA provides Resource management as most Resources have to do with CRUD (Create, Read, Update and Delete) primitives. The name of ROA evokes REST or Representational State Transfer, an HTTP based web service that, according to many in the industry, is intended to be more light weight than SOAP (Standard Object Access Protocol) that is common with SOA web service on HTTP. REST semantics fit well with Resource management because it maps CRUD to HTTP Methods GET, PUT, POST and HEAD. So REST can afford to directly utilize HTTP for Resource Management rather than using an additional SOAP aspect necessary for SOA Business Services. This is one of its key differences with SOA web service which uses SOAP. Because only HTTP Methods can be used in REST, it also constrains the client side to be Hyper Text driven.

We, on the other hand, want keep things simpler by employing the same JAX-WS web service standard for ROA (Resource Management). Of course, we preserve REST like semantics (CRUD) of Resource Management because that is fundamental to ROA. We utilize Java Servlet to map a web service method to the proper HTTP Method. But this is just our choice of technology for ROA. It still does not answer how we can avoid having an explosion of web services for every Resource and instead have a single ROA web service for all Resources.

Hitting a wall?

In typical JAX-WS SOA, every web service method is mapped to a WSDL "Operation" among other things. That is why, every time a method is modified or a new method is introduced to the web service, the web service redeployed and the WSDL is modified. With JAX-WS ROA, when we introduce a new Resource, our apparent choices are either to create a new web service or update an existing web service through which the new Resource is made available. We know we can't run with creating a new web service for every Resource, and redeploying a web service entails some operational downtime that can be a show stopper in the 24x7, ubiquitous cloud environment (Figure 1 below is a high-level view of JAX-WS based web service. Figure 2 shows components that are affected during web service redeployment). So, what's the way out?

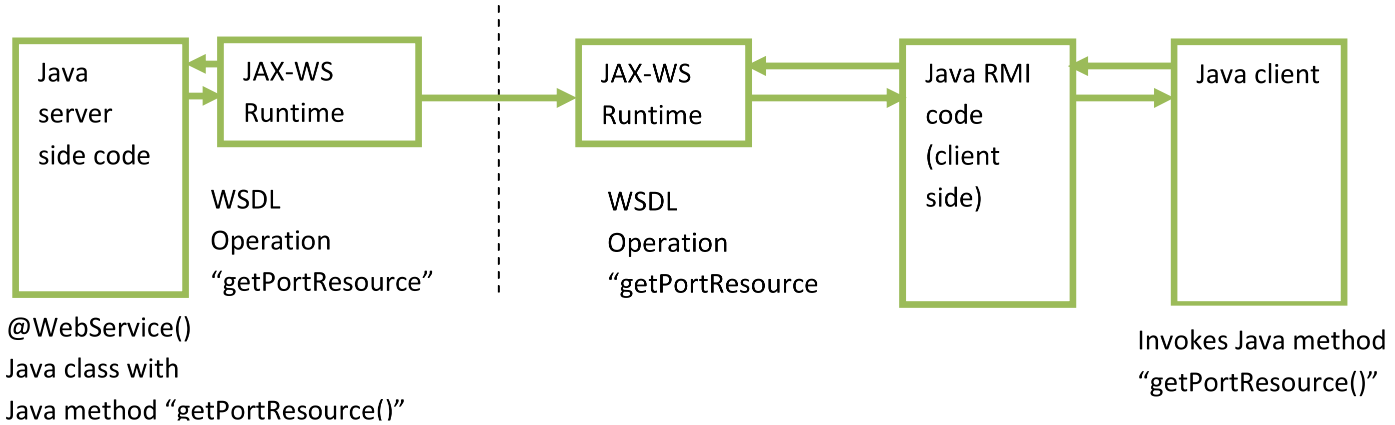

Java server side code
JAX-WS Runtime
WSDL Operation "getPortResource"
JAX-WS Runtime
WSDL Operation "getPortResource
Java RMI code (client side)
Java client
@WebService() Java class with Java method "getPortResource()"
Invokes Java method "getPortResource()"

Figure 1: JAX-WS SOA

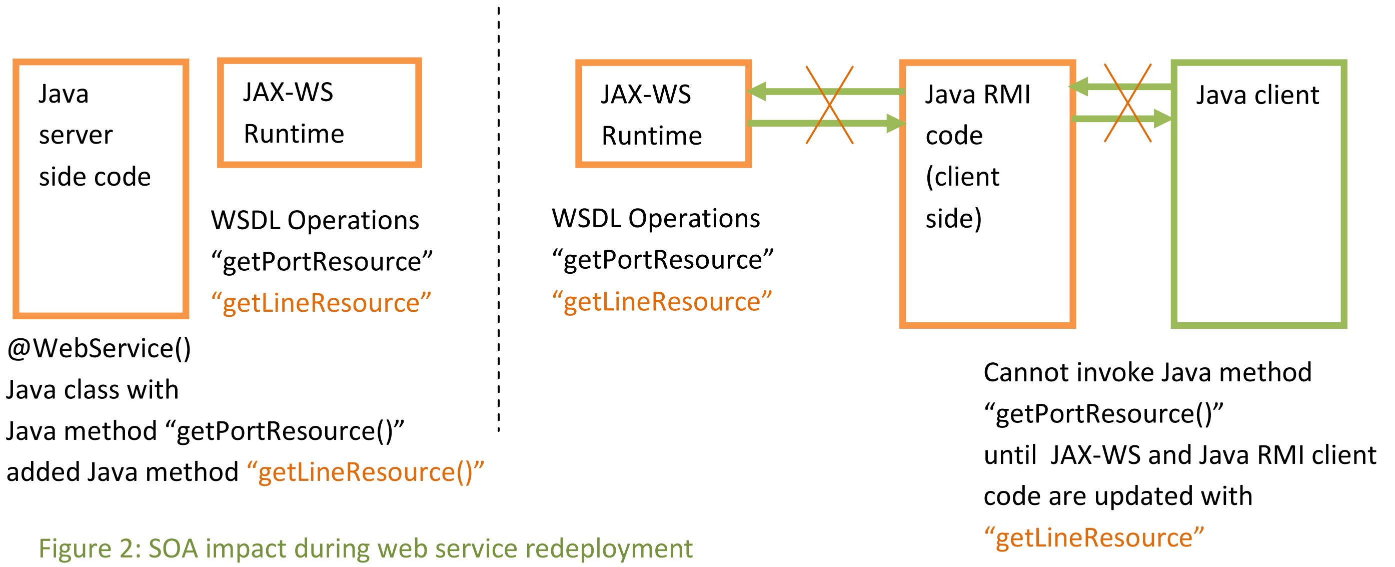

Java server side code
JAX-WS Runtime
WSDL Operations "getPortResource" "getLineResource"
JAX-WS Runtime
WSDL Operations "getPortResource" "getLineResource"
Java RMI code (client side)
Java client
@WebService() Java class with Java method "getPortResource()" added Java method "getLineResource()"
Cannot invoke Java method "getPortResource()" until JAX-WS and Java RMI client code are updated with "getLineResource"

Figure 2: SOA impact during web service redeployment

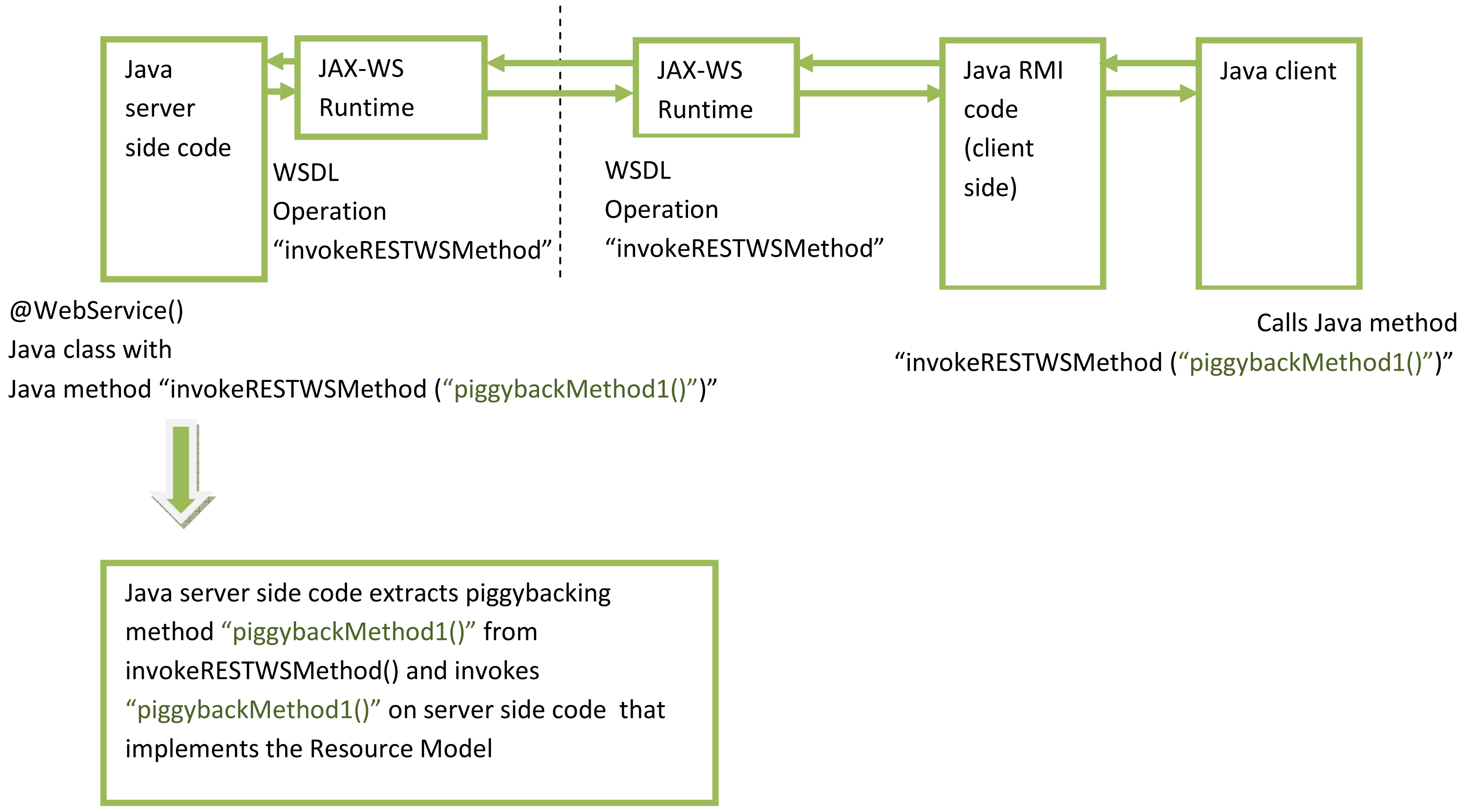
🟧 Indicates components needing update with getLineResource

Presenting Iaqua, the “Fluid as Aqua Interface”

What if we could avoid updating the WSDL and redeploying JAX-WS and yet be able to introduce a new Resource to an existing ROA web service? Can it reduce impact to existing ROA web service clients and business continuity, and at the same time, let clients seamlessly see the new Resource?

The key to this is in injecting methods corresponding to a Resource “dynamically” to the web service. These methods are not mapped to a WSDL Operation. Rather, a single WSDL Operation is present, and it allows to “piggy-back” the dynamic methods. Because the WSDL (and web service interface) remains unchanged as these methods come and go dynamically, the web service’s JAX-WS redeployment becomes unnecessary, and impact to existing clients is reduced.

Figure 3 below shows the idea of a single WSDL Operation that allows dynamically added methods to piggyback.


Java server side code
JAX-WS Runtime
WSDL Operation “invokeRESTWSMethod”
JAX-WS Runtime
WSDL Operation “invokeRESTWSMethod”
Java RMI code (client side)
Java client
@WebService() Java class with Java method “invokeRESTWSMethod (“piggybackMethod1()”)”
Calls Java method “invokeRESTWSMethod (“piggybackMethod1()”)”

Java server side code extracts piggybacking method “piggybackMethod1()” from invokeRESTWSMethod() and invokes “piggybackMethod1()” on server side code that implements the Resource Model

Figure 3: Using a single WSDL Operation to piggyback multiple methods

As seen in figure 3, the Java server side code extracts the piggybacking method from the Web Service method (WSDL Operation) and then invokes the extracted method on the Resource. Consistent with OO, the extracted method that is invoked on the Resource is implemented by the Resource itself.

Figure 4 below shows how the impact to ROA components and business continuity is reduced while releasing addition of a new method or modification of an existing method into the web service.

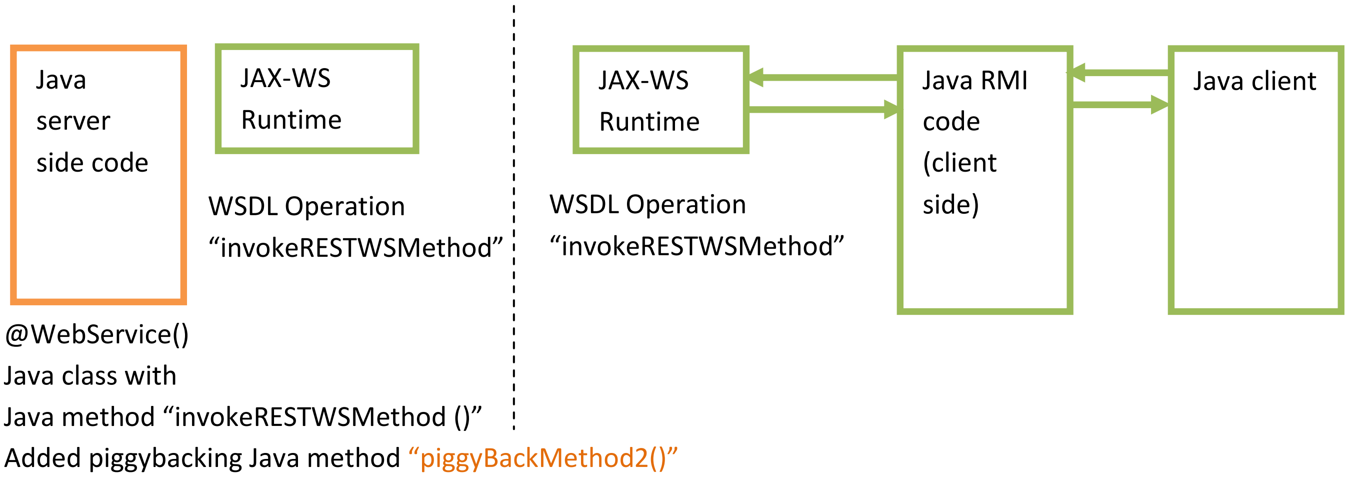

Java server side code
JAX-WS Runtime
WSDL Operation "invokeRESTWSMethod"
JAX-WS Runtime
WSDL Operation "invokeRESTWSMethod"
Java RMI code (client side)
Java client
@WebService()
Java class with
Java method "invokeRESTWSMethod ()"
Added piggybacking Java method "piggyBackMethod2()"

Figure 4: SOA impact during web service redeployment with piggybacking

🟧 Indicates components needing update with piggyBackMethod2()

Only the Java server side code is impacted due to addition of new method or update of an existing one. Benefits of avoiding impact to JAX-WS and Java RMI components (see figure 4 above) accrue from not having to re-generate/re-deploy and regression test these components. This not only saves cost but also accelerates overall availability of the updated web service. Benefits are proportional to the volume of clients integrating the web service.

Java technologies at work

In our example implementation, the Server side code is implemented using J2EE. Our Application Server is Apache Geronimo. Resources are modelled as regular Java Objects (we call them IJOB or Iaqua Java Object). The dynamic availability of Resources (and their methods) is achieved by using a "bean" pattern. A bean in this context is The Geronimo GBean facility. It is a class or facility that can have a lifecycle managed by the Geronimo J2EE container. This allows us to 'configure' GBeans on the fly with IJOBs that represent Resources. A GBean becomes the provider of a Resource Model, and thus, we can instantiate a GBean and invoke all methods implemented by the IJOB on the GBean, just as we would the IJOB itself. This GBean is exposed through the J2EE container to the web service which defines a single WSDL Operation "invokeRESTWSMethod". Every GBean's method call from the client piggybacks invokeRESTWSMethod. This method is then extracted and invoked on the GBean at the server side, as shown in figure 3.

Since the WSDL is transparent to the piggybacked methods themselves, there is no interface specification in the WSDL for these methods. It is left to the Java client and Java server sides to agree on the method signatures for each Resource. This is accomplished by specifying the Resource entity, its methods and applicable Data Types in XML format. A client parses these specifications and creates following mappings to integrate the rest of its own application:

Iaqua (XML namespace) Data types to Java data types

Java data types for method arguments and return type

Method names (Remember, these are Methods that are exposed by a Resource. Call to these methods will piggyback invokeRESTWSMethod web service method)

Resource name: Note that in Iaqua XML files, a Resource is referred to as 'entity'. Remember, these are the Resource Models which implement Methods.

Once these mappings are done, the client specifies Method Name, Arguments and Return Parameter as string literals. Then the client invokes a Method by supplying these parameters to invokeRESTWSMethod ("piggybacking"). The server side code then converts the string literals for Arguments to the appropriate Java data types before invoking the Method on the Resource GBean. Similarly, the server side converts all Method Return values to string literals and passes back to the client. The client then converts the string literal to a value of appropriate java data type. The java data types are correctly interpreted by both the client as well as server by agreeing on the XML specification for every Resource entity, its methods and applicable Data Types. There is no involvement of SOAP in this mechanism except facilitating the invokeRESTWSMethod invocation with String arguments. Because of this, there is no impact to JAX-WS/SOAP/WSDL aspects as Resources and their Methods are added, modified or removed on the fly.