

REST-based Web Services (II)

Helen Paik

School of Computer Science and Engineering
University of New South Wales

Resources:

- RESTful Web Services by L. Richardson and S. Ruby, O'Reilly
- Building Web Services the REST Way, By Roger L. Costello – <http://www.xfront.com/REST-Web-Services.html>
- ICSOC 2008 Summer School session on REST-based Services – article: How to get a cup of coffee by Jim Webber (<http://www.infoq.com/articles/webber-rest-workflow>)

Acknowledgement: The slides are adapted from 'RESTful Web Services' chapters 4, COMP9322 lectures in the previous sessions prepared by Dr. Adnene Guabtni and Dr. Sherif Sakr.

22 April 2010

Brief Recap of REST principles

Resources:

- Resource: Any *thing* (noun) that is worthy of being given a unique ID (URI) and be accessible via client
- Resources are something the server is responsible for managing
- Resources must have representations to be 'transmitted' to client

e.g., resources in the starbucks example: order, payment (represented in XML)

Brief Recap of REST principles

Uniform Interface: Uniform 'verbs' that go with the resources (noun)

Given a resource (coffee order): a representation in XML

```
<order xmlns="urn:starbucks">  
  <drink>latte</drink>  
</order>
```

Resources and Uniform Interface

POST /starbucks/orders

returns: location: /starbucks/orders/order?id=1234

GET /starbucks/orders/order?id=1234

PUT /starbucks/orders/order?id=1234

DELETE /starbucks/orders/order?id=1234

Brief Recap of REST principles

Connectedness/Links: Resources may contain links to other resources

e.g., Order resource is linked to Payment resource

In response to POSTing an order

201 Created

Location: /starbucks/orders/order?id=1234

Content-Type: application/xml

Content-Length: ...

```
<order xmlns="urn:starbucks">
  <drink>latte</drink>
  <link rel="payment" href="/starbucks/payments/order?id=1234"
        type="application/xml"/>
</order>
```

Both forward/backward links, when possible (e.g., order having 'cancel/delete' link)

REST - Hypermedia

■ Getting the list of parts

GET <http://www.parts-depot.com/parts> HTTP/1.1

HTTP/1.1 200 OK

```
<?xml version="1.0"?>
```

```
<p:Parts xmlns:p="http://www.parts-depot.com"
```

```
  xmlns:xlink="http://www.w3.org/1999/xlink">
```

```
  <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
```

```
  <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
```

```
  <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
```

```
  <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
```

```
</p:Parts>
```

Web Server

REST - Hypermedia

■ Getting the details of a specific part

GET <http://www.parts-depot.com/parts/00345> HTTP/1.1

HTTP/1.1 200 OK

```
<?xml version="1.0"?>
```

```
<p:Part xmlns:p="http://www.parts-depot.com"
```

```
  xmlns:xlink="http://www.w3.org/1999/xlink">
```

```
<Part-ID>00345</Part-ID>
```

```
<Name>Widget-A</Name>
```

```
<Description>This part is used within the trap assembly</Description>
```

```
<Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
```

```
<UnitCost currency="USD">0.10</UnitCost>
```

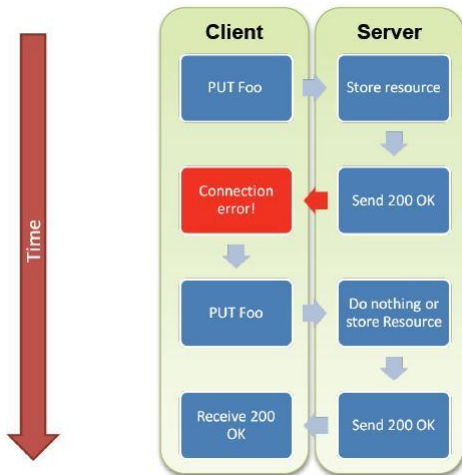
```
<Quantity>10</Quantity>
```

```
<Order href="http://.../Orders/" />
```

```
</p:Part>
```

Web Server

REST Safety, Idempotence



Popular Resource Representation Formats

XML (RSS, ATOM, ...)

- Standard textual syntax for semi-structured data
- Managed within several standard tools like XML Schema, DOM, SAX, XPath, XSLT, XQuery, ...

```
<animals>
  <dog>
    <name>Rufus</name>
    <breed>labrador</breed>
  </dog>
  <dog>
    <name>Marty</name>
    <breed>whippet</breed>
  </dog>
  <cat name="Matilda"/>
</animals>
```


Popular Resource Representation Formats

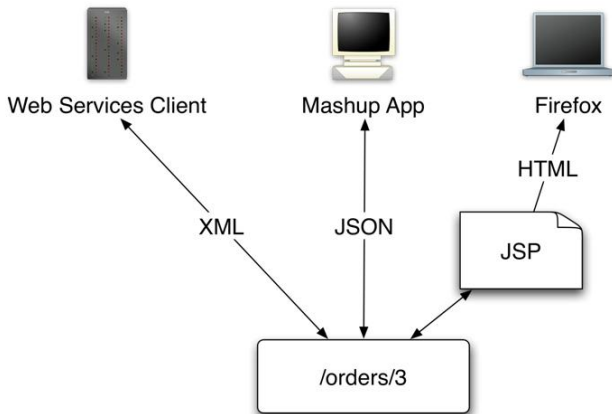
JSON (JavaScript Object Notation)

- Data interchange format derived from the JavaScript programming language for representing simple data structures (name/value pairs, arrays and other objects)
- Popularised by AJAX (Asynchronous JavaScript + XML).

```
{  animals:{
    dog:[
      {    name:'Rufus',
          breed:'labrador'  },
      {    name:'Marty',
          breed:'whippet'   }
    ],
    cat:{  name:'Matilda'   }
  }
}
```

One API for All

One resource, multiple representations (it is the client's choice):



Building REST Web services

In theory: REST does not require a specific client or server-side framework in order to write your Web services. All you need is a client or server that supports the HTTP protocol.

- choose a language of your choice
- you do not need a big server infrastructure (HTTP/Web servers are enough)

e.g., In Java: You'd use servlets and override `doGet()`, `doPost()`, `doPUT()` and `doDelete()`

- URLs contains: servlet path + path info (all you need to process a request in REST)
- You could use a third-party library for generating specific content type (CSV, JSON or XML, etc.) or use Strings concatenations for simple responses.

Building REST Web services

e.g., A quick look at a simple servlet that supports GET (and XML, CSV and JSON resource formats)

- A servlet is mapped to `/places/sydney/*`
- GET returns some interesting places in Sydney
- Default representation is XML, but other format can be added in the path info: JSON and CSV
- Uses `http://www.JSON.org/java/index.html` for the JSON encoding

Building REST Web services

```
public class SydneyServlet extends HttpServlet {  
  
    private static final String[] places = {  
        "Harbour Bridge",  
        "Circular Quay",  
        "Opera House",  
        "Hyde Park",  
        "Darling Harbour",  
        "Bondi Beach",  
        "Coogee Beach"  
    };  
  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws  
ServletException, IOException {  
        if ("/json".equals(req.getPathInfo())) {  
            jsonReply(resp);  
        } else if ("/csv".equals(req.getPathInfo())) {  
            csvReply(resp);  
        } else {  
            xmlReply(resp);  
        }  
    }  
}
```

Building REST Web services (Implementing an XML reply)

```
private void xmlReply(HttpServletResponse resp) throws ServletException, IOException {  
    resp.setContentType("text/xml");
```

```
    Document doc = DocumentFactory.getInstance().createDocument();
```

```
    Element root = doc.addElement("cityplaces");
```

```
    root.addElement("city").setText("Sydney");
```

```
    Element list = root.addElement("places");
```

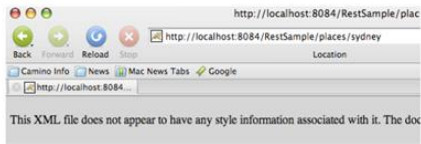
```
    for (int i = 0; i < places.length; ++i) {
```

```
        list.addElement("place").setText(places[i]);
```

```
    }
```

```
    resp.getWriter().write(doc.asXML());
```

```
}
```

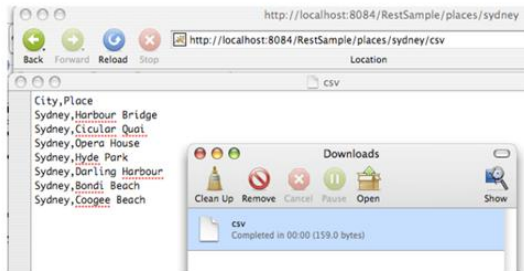


```
- <cityplaces>  
  <city>Sydney</city>  
  - <places>  
    <place>Harbour Bridge</place>  
    <place>Circular Quay</place>  
    <place>Opera House</place>  
    <place>Hyde Park</place>  
    <place>Darling Harbour</place>  
    <place>Bondi Beach</place>  
    <place>Coogee Beach</place>  
  </places>  
</cityplaces>
```

Building REST Web services (Implementing a CSV reply)

```
private void csvReply(HttpServletResponse resp) throws  
ServletException, IOException {  
    resp.setContentType("text/csv");
```

```
    Writer writer = resp.getWriter();  
    writer.write("City,Place\r\n");  
    for (int i = 0; i < places.length; ++i) {  
        writer.write("Sydney,");  
        writer.write(places[i]);  
        writer.write("\r\n");  
    }  
}
```

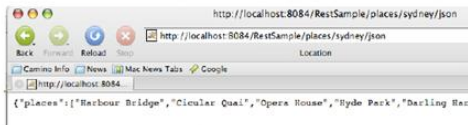


Building REST Web services (Implementing a JSON reply)

```
private void jsonReply(HttpServletResponse resp) throws ServletException,
IOException {
    resp.setContentType("text/json");

    JSONObject root = new JSONObject();
    root.put("city", "Sydney");
    JSONArray list = new JSONArray();
    for (int i = 0; i < places.length; ++i) {
        list.add(places[i]);
    }
    root.put("places", list);

    resp.getWriter().write(root.toString());
}
```



Building REST services

In reality ... it would still be nice to have some support (a framework)

Having a framework can help reduce boilerplate-type coding

In Java, standard specification is: JAX-RS

(<http://jcp.org/en/jsr/detail?id=311>)

- **Apache CXF**
- **Jersey** - the JAX-RS Reference Implementation from Sun.
- **RESTEasy** - JBoss's JAX-RS project.
- **Restlet** - probably the first REST framework, which existed prior to JAX-RS.

Building REST services: JAX-RS

<http://java.dzone.com/articles/putting-java-rest>

- JAX-RS uses Java annotations
- Annotate your class with:
 - `@Path`: to indicate the relative URI path you are interested in,
 - `@GET`, `@POST`, `@PUT`, `@DELETE`, or `@HEAD`: to indicate which HTTP method you want dispatched to a particular method.

```
01. @Path("/orders")
02. public class OrderEntryService {
03. @GET
04. public String getOrders() {...}
05. }
```

Given "GET `http://xxx.xxx.xx/orders`", JAX-RS would dispatch the HTTP request to `getOrders()` and return the content `getOrders()` produces.

Building REST services: JAX-RS

We may want to limit the size of the result set returned.

“GET http://xxx.xxx.xx/orders?size=50”

To extract this information from the HTTP request, use a `@QueryParam` annotation:

```
01. @Path("/orders")
02. public class OrderEntryService {
03. @GET
04. public String getOrders(@QueryParam("size")
05.                          @DefaultValue("50") int size)
06. { ... method body ... }
09. }
```

`@QueryParam` will pull the “size” query parameter from the incoming URL

The `@DefaultValue` injects the default value of “50”.

Building REST services: JAX-RS

Exposing a single order as a resource:

“GET http://xxx.xxx.xx/orders/1234”

Using Path Parameters:

```
01. @Path("/orders")
02. public class OrderEntryService {
03.     @GET
04.     @Path("{id}")
05.     public String getOrder(@PathParam("id") int orderId) {
06.         ... method body ...
07.     }
08. }
```

To use `@PathParam` or `@QueryParam`: let's not put resource identity within a query parameter when it really belongs as part of the URI path itself.

Building REST services: JAX-RS

Content-Type: the service produces HTTP response, so the content type must be set first.

For example, let's say our `getOrders()` method actually returns an XML string.:

```
01. @Path("/orders")
02. public class OrderEntryService {
03.     @GET
04.     @Path("{id}")
05.     @Produces("application/xml")
06.     public String getOrder(@PathParam("id") int orderId)
07.     {... method body }
08. }
```

Building REST services: JAX-RS

JAX-RS understands the Accept header and will use it when dispatching.

Content Negotiation:

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
the client prefers: html or xhtml (q is 1 by default), raw XML second, and any other content type third

```
01. @Path("/orders")
02. public class OrderEntryService {
03.     @GET
04.     @Path("{id}")
05.     @Produces("application/xml")
06.     public String getOrder(@PathParam("id") int orderId) {...}
07.     @GET
08.     @Path("{id}")
09.     @Produces("text/html")
10.     public String getOrderHtml(@PathParam("id") int orderId) {...}
11. }
```

“GET http://.../orders/1234” with the above Accept would match getOrderHtml().

Building REST services: JAX-RS

Content Marshalling: code to convert a list of orders to an XML string that the client can consume.

The JAX-RS specification has some required built-in marshallers (e.g., JAXB annotated classes - JBoss RESTEasy also has providers for JSON)

```
@XmlRootElement(name="order")
public class Order {
    @XmlElement(name="id")
    int id;

    @XmlElement(name="customer-id")
    int customerId;

    @XmlElement("order-entries")

    List<OrderEntry> entries; ...
}
```

```
@Path("/orders")
public class OrderEntryService {
    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Order
        getOrder(@PathParam("id") int orderId)
}
```

JAX-RS sees the Content-Type (application/xml) and the JAXB annotated Order class and will automatically use JAXB to write the Order object to the HTTP output

Building REST services: JAX-RS

Response Codes and Custom Responses: HTTP spec defines what HTTP response codes should be on a successful request. (e.g., GET = 200, OK; PUT = 201, CREATED)

JAX-RS returns the same default response codes.

Sometimes, however, you need to specify your own response codes (or add specific headers or cookies to your HTTP response).

```
@Path("/orders")
public class OrderEntryService {
    @GET
    @Path("{id}")
    public Response getOrder(@PathParam("id") int orderId)
    {
        Order order = ...;
        ResponseBuilder builder = Response.ok(order);
        builder.expires(...some date in the future);
        return builder.build();
    }
}
```


Building REST services: JAX-RS

Exception Handling: RuntimeException/WebApplicationException class

It can take an HTTP status code or a Response object as one of its constructor parameters.

```
@Path("/orders")
public class OrderEntryService {
    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Order getOrder(@PathParam("id") int orderId) {
        Order order = ...;
        if (order == null) {
            ResponseBuilder builder = Response.status(Status.NOT_FOUND);
            builder.type("text/html");
            builder.entity("<h3>Order Not Found</h3>");
            throw new WebApplicationException(builder.build());
        }
        return order;
    }
}
```

REST and scalability

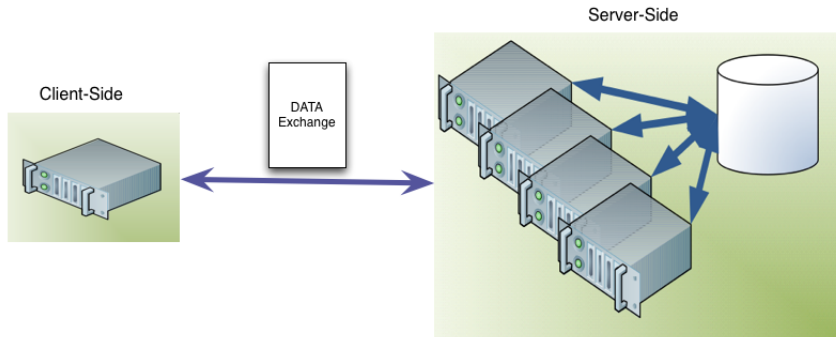
Being Stateless:

- Every action happens in isolation: This is a good thing!
 - In between requests the server knows nothing about you
 - Keeps the interaction protocol simpler
 - Makes recovery, scalability, failover much simpler too

REST and scalability

Scalable Deployment Configuration

- Deploy services onto many servers
- Services are stateless – No sessions!
- Servers share only back-end data



REST and scalability

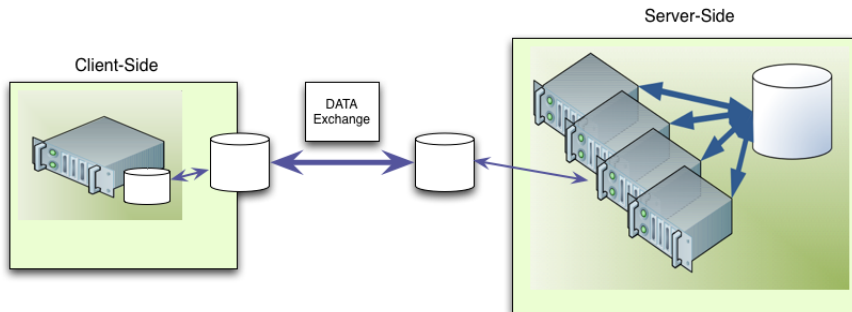
Scaling Vertically... without servers

- The most expensive round-trip:
 - From client
 - Across network
 - Through servers
 - Across network again
 - To database
 - And all the way back!
- The Web tries to short-circuit this: By determining early if there is any actual work to do! – Caching

REST and scalability

Caching in a Scalable Deployment

- Cache (reverse proxy) in front of server farm: Avoid hitting the server
- Proxy at client domain: Avoid leaving the LAN
- Local cache with client: Avoid using the network



REST and scalability

Being work-shy is a good thing!

- Provide guard clauses in requests so that servers can determine easily if there's any work to be done
- If-Modified-Since, If-None-Match/ETag
- Web infrastructure uses these to determine if its worth performing the request
- And often it isn't, So an existing representation can be returned

REST and scalability

Retrieving a Resource Representation

Request

```
GET /transactions/debit/1234 HTTP 1.1
Host: bank.example.org
Accept: application/xml
If-Modified-Since: 2010-21-04T15:00:34Z
If-None-Match: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Response

```
200 OK
Content-Type: application/xml
Content-Length: ...
Last-Modified: 2010-21-04T15:10:32Z
Etag: abbb4828-93ba-567b-6a33-33d374bcad39
<t:debit xmlns:t="http://bank.example.com">
<t:sourceAccount>12345678</t:sourceAccount>
<t:destAccount>987654321</t:destAccount>
<t:amount>299.00</t:amount>
<t:currency>GBP</t:currency>
</t:debit>
```

REST and scalability

Not Retrieving a Resource Representation

Request

```
GET /transactions/debit/1234 HTTP 1.1
Host: bank.example.org
Accept: application/xml
If-Modified-Since: 2010-21-04T15:00:34Z
If-None-Match: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Response

```
200 OK
Content-Type: application/xml
Content-Length: ...
Last-Modified: 2010-21-04T15:00:34Z
Etag: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Client's representation of the resource is already up-to-date

REST and security

REST lacks a well-articulated security model... by virtue of its grassroots origins, [it] suffers from a security just-do-it-like-the-web nonchalance that's certainly done it no favors... The REST style owes much of its popularity to being simple and fast to implement, particularly when faced with the interest-crushing complexity and tooling demands of the WS- stack. It's reasonable to think that in the enthusiastic dash to cross the working application finish line, that security is conveniently de-emphasized or forgotten altogether.*

<http://kscottmorrison.com/2010/03/04/rest-security-does-exist~Nyou-just-need-to-apply-it/>

REST and security

So, REST relies on HTTP for security too ...

HTTP Basic or HTTP Digest or SSL (certificate-based) for authentication. SSL for encryption and digital signatures.

HTTP Basic and Digest Authentication: IETF RFC 2617

- Have been around since 1996 (Basic)/1997 (Digest)
- Pros:
 - Respects Web architecture:
 - stateless design (retransmit credentials)
 - headers and status codes are well understood: Does not prohibit caching (set Cache-Control to public)
- Cons:
 - Basic Auth must be used with SSL/TLS (plaintext password)
 - Only one-way authentication (client to server)

REST and security

HTTP Basic Auth Example:

- 1 Initial HTTP request to protected resource

```
GET /index.html HTTP/1.1
```

```
Host: example.org
```

- 2 Server responds with

```
HTTP/1.1 401 Unauthorized
```

```
WWW-Authenticate: Basic realm='MyRealm'
```

- 3 Client resubmits request

```
GET /index.html HTTP/1.1
```

```
Host: example.org
```

```
Authorization: Basic Qm9iCnBhc3N3b3JkCg==
```

Further requests with same or deeper path can include the additional Authorization header preemptively

REST and security

SSL / TLS

- “Strong” server authentication, confidentiality and integrity protection
- The only feasible way to secure against man-in-the-middle attacks
- Not cache friendly, even using ‘null’ encryption mode
- Performance vs. security – becomes difficult

REST and security

OAuth:

- Web-focused access delegation protocol
- Give other Web-based services access to some of your protected data without disclosing your credentials
- Simple protocol based on HTTP redirection, cryptographic hashes and digital signatures
- Extends HTTP Authentication as the spec allows:
 - Makes use of the same headers and status codes
 - These are understood by browsers and programmatic clients

REST and security

Why OAuth?

Find people you know on Facebook

Your friends on Facebook are the same friends, acquaintances and family members that you communicate with in the real world. You can use any of the tools on this page to find more friends.



Find People You Email

[Upload Contact File](#)

Searching your email address book is the fastest and most effective way to find your friends on Facebook.

Your Email:

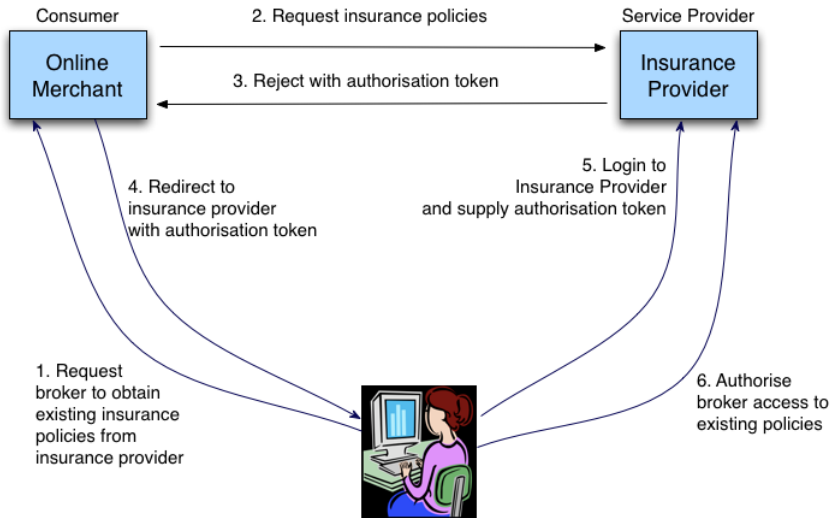
Password:

Find Friends

We won't store your password or contact anyone without your permission.

REST and security

OAuth Workflow:



REST and security

OAuth:

- 1 Alice (the User) has accounts on both the insurance broker and provider's Web sites
- 2 The insurance broker (Consumer) has registered itself at the insurance company and has a Consumer Key and Secret
- 3 Alice logs in to the broker and requests it to obtain her existing policies from the provider
- 4 Broker request to Insurance Provider:
GET /alice/policies HTTP 1.1
Host: insurance.org
- 5 Insurance provider's response:
401 Unauthorized
WWW-Authenticate: OAuth realm="http://insurance.org/"


REST and security

OAuth:

- 1 Broker requests authorisation token from Provider:
POST /request_token
oauth_consumer_key=abc&oauth_nonce=39kg&oauth_ ...
- 2 Provider sends authorisation token in response body:
200 Success
oauth_token=xyz&oauth_token_secret=abc
- 3 Broker redirects Alice to Provider in response to her request:
302 Redirect
Location: http://insurance.org/authorise?oauth_token=xyz&oauth_callback=http%3A%2F%2Fbroker.org&...
- 4 Alice logs in to Insurance Provider using her credentials at that site (the Broker never sees these) and authorises the Broker to access her existing policies for a defined period of time.

REST and security

OAuth:

- 1 Insurance Provider redirects Alice to the callback URL:
302 Redirect
Location: `http://broker.org/token_ready?oauth_token=xyz`
- 2 Broker knows Alice approved, it asks Provider for Access Token:
GET `/accesstoken?oauth_consumer_key=abc&oauth_token=xyz`
Host: `insurance.org`
- 3 The Insurance Provider sends back the Access Token:
200 Success
`oauth_token=zxcvb`
- 4 Broker creates hash or signature using access token, nonce, timestamp, Consumer Key and Secret (and more):
GET `/alice/policies` HTTP 1.1
Host: `insurance.org`
Authorization: OAuth realm=`‘‘http://insurance.org/’’`,
`oauth_signature=‘‘...’’`, `oauth_consumer_key=‘‘abc’’`, 

REST and security

(Not a real example - but PayPal works via similar workflow anyway)

`http://wiki.oauth.net/0Auth%20for%20Payment%20flow`

Positive aspects of REST

- Faster response time (caching)
- Reduced server load (caching)
- Improved server scalability and easier load-balancing (no session state)
- Client software can be reused (uniform interface)
- Can be implemented with any server-side technology
- HTTP client libraries are widespread
- Easy to serve different types of content such as images, videos, xl-sheets,..., etc
- The "web" in "Web services" is for real!

Negative aspects of REST

- No interface description language like IDL or WSDL
 - need to “read” the doc to understand how to interact ...
 - e.g., typical example
`http://wiki.developers.facebook.com/index.php/API`
- Low tooling support for automation (but frameworks are coming along)
- Most web browsers don't know PUT and DELETE
- Not complex enough for vendors to make expensive “enterprise” products :-)

When can I use REST?

- For Web Services
 - build your web service using the REST style
 - alternative to some of WS-*, not a replacement for WS-*
- Clients interfacing to public REST APIs
 - e.g. Amazon S3 REST API, Google Data APIs
 - Many other public APIs have a REST like interface
- From Rich Internet Applications (RIAs)?
 - client sends AJAX requests to a REST interface using a JavaScript library e.g. jQuery, or a framework like JavaFX or Silverlight
 - response (JSON, XML etc) is displayed on the client

REST + SOAP

Wrapping REST request/response with SOAP envelopes:

For example, the following request

`http://127.0.0.1:8080/axis2/services/version/getVersion`

will be converted to the following SOAP Message for processing by Axis2

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <axis2:getVersion xmlns:axis2="http://ws.apache.org/goGetWith... />
  </soapenv:Body>
</soapenv:Envelope>
```

`http://ws.apache.org/axis2/0_94/rest-ws.html`

REST + SOAP

GET <http://www.parts-depot.com/parts/00345> HTTP/1.1

```
<?xml version="1.0"?>
...
<soap:Envelope ...>
<soap:Body>
<PartDetails xmlns=" ...>
  <Part-ID>00345</Part-ID>
  <Name>Widget-A</Name>
  <Description>This part is used within the frap assembly</Description>
  <Spec xlink:href="http://www.parts-depot.com/parts/00345/spec"/>
  <UnitCost currency="USD">0.10</UnitCost>
  <Quantity>10</Quantity>
  <Order href="http://.../Orders/" />
</PartDetails>
</soap:Body>
</soap:Envelope>
```


REST + SOAP

Placing an order:

```
POST http://www.parts-depot.com/orders HTTP/1.1
Content-type: application/soap+xml
```

```
<soap:Envelope>
<soap:Body>
  <order xmlns="...">
    <part href="http://.../00345" />
    <orderDetails> ... </orderDetails>
  </order>
</soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-type: application/soap+xml

<soap:Envelope>
<soap:Body>
  <confirmation xmlns="...">
    <order href="http://.../ordno?=1234"/>
  </confirmation>
</soap:Body>
</soap:Envelope>
```

Web Server

REST vs SOAP - Technology

- User-driven interaction via rich Web client application
- Few operations on many resources
- URI - consistent naming for resources
- focus on Scalability and performance
- Distributed hypermedia systems
- Composition= mashups
- Orchestrated, reliable event/message flows
- Many operations (WSDL interface) on few resources
- Lack of standard naming mechanism
- focus on design of integrated enterprise applications
- composition = business processes

REST vs SOAP - Protocol

- XML in – XML out (with POST)
- URI in – XML out (with GET)
- Self-Describing” XML
- HTTP only
- HTTP/SSL is enough – no need for more standards
- HTTP is an application protocol
- Synchronous
- Do-it-yourself when it comes to “reliable message delivery”, “distributed transactions”
- SOAP in – SOAP out (with POST)
- Strong Typing (XML Schema)
- “Transport independent”
- Heterogeneity in QoS needs.
- Different protocols may be used
- HTTP as a transport protocol
- Synchronous and Asynchronous
- Foundation for the whole WS* advanced protocol stack

REST vs. SOAP Design Methodology

- Identify resources to be exposed as services (e.g., book catalog, purchase order)
- Define “nice” URLs to address them
- Distinguish read-only and side-effects free resources (GET) from modifiable resources (POST, PUT, DELETE)
- Relationships (e.g., containment, reference) between resources correspond to hyperlinks that can be followed to get more details
- Implement and deploy on Web server
- List what are the service operations (the “verbs”) into the service’s WSDL document
- Define a data model for the content of the messages exchanged by the service (XML Schema data types)
- Choose an appropriate transport protocol to bind the operation messages and define the corresponding QoS, security, transactional policies
- Implement and deploy on the Web service container (note the corresponding SOAP engine endpoint)

REST Resources

- **RESTWiki -**

<http://internet.conveyor.com/RESTwiki/moin.cgi/>

- **REST mailing list -**

<http://groups.yahoo.com/group/rest-discuss/>

- Roy Fielding's Dissertation http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf