REST

REPRESENTATIONAL STATE TRANSFER

OVERVIEW OF REST, AN ARCHITECTURAL STYLE FOR DISTRIBUTED WEB BASED APPLICATIONS

Peter R. Egli
peteregli.net

Contents

1. What is „Representational State Transfer“ ? (1/3)

REST, unlike SOAP, is not a WS (web service) standard but an architectural style for web applications.

REST was devised by Roy Fielding in his doctoral dissertation:

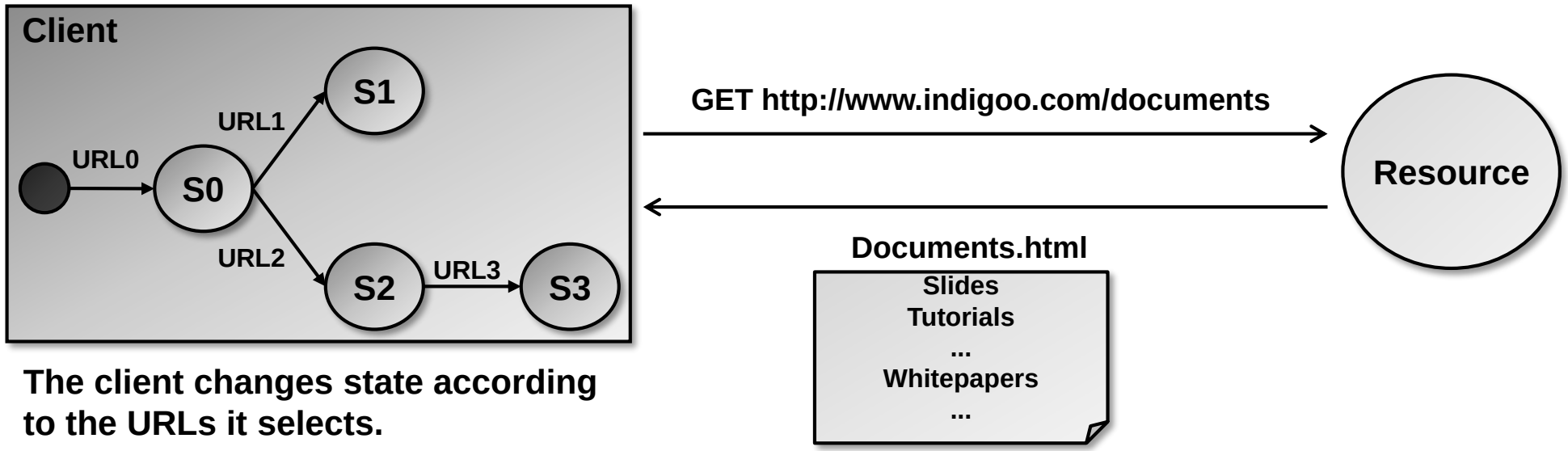
"Representation State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

- **REST is not a standard or protocol, REST is an architectural style.**
- **REST makes use of existing web standards (HTTP, URL, XML, JSON, MIME types).**
- **REST is resource oriented. Resources (pieces of information) are addressed by URIs and passed from server to client (or the other way round).**

N.B.: Roy Fielding is also author of a number of RFCs (e.g. RFC2616 HTTP 1.1, RFC3986 URI).

1. What is „Representational State Transfer“ ? (2/3)

To understand the REST principle, look at what happens in a web access of a browser:



The client changes state according to the URLs it selects.

1. The client references a web resource using a URL.
2. The web server returns a *representation* of the resource in the form of an HTML document.
3. This resource places the client into a new *state*.
4. The user clicks on a link in the resource (e.g. Documents.html) which results in another resource access.
5. The new resource places the client in a new state.

➔ The client application changes (=*transfers*) *state* with each resource *representation*.

1. What is „Representational State Transfer“ ? (3/3)

REST is based on existing web (WWW, HTTP) principles and protocols:

Resources:
Application state and functionality are abstracted into resources (everything is a resource).

Addressability of resources:
Every resource is uniquely addressable using hyperlinks.

Uniform interface for accessing resources:
All resources share a uniform interface for the transfer of state between client and resource, consisting of

- **a constrained (=limited) set of well-defined operations (GET, PUT, POST, DELETE).**
- **a constrained set of content types (text/html, text/xml etc.).**

2. Why REST?

Scalability of WWW:

The WWW has proven to be:
a. scalable (growth)
b. simple (easy to implement, easy to use)

REST rationale:

If the web is good enough for humans, it is good enough for machine-to-machine (M2M) interaction.
The concepts behind RPC-WS (SOAP, XML-RPC) are different. RPC-WS make very little use of WWW-concepts and technologies. Such WS define an XML-based interface consisting of operations that run on top of HTTP or some other transport protocol. However, the features and capabilities of HTTP are not exploited.

The motivation for REST was to create an architectural model for web services that uses the same principles that made the WWW such a success.
The goal of REST is to achieve the same scalability and simplicity.

➔ REST uses proven concepts and technologies.
➔ REST keeps things as simple as possible.

3. Architectural constraints for REST (1/2)

REST defines 6 architectural constraints that a system architecture should comply with to obtain scalability.

1. Client-server paradigm:

A client retrieves resources from a server or updates resources on a server.

- **Separation of concerns such as presentation (client) from data storage (server).**
- **Portability (UI may be ported to different platforms).**

2. Stateless:

A client request contains all information necessary for the server to understand the request.

- **No need for storing context (state) on the server.**
- **Better scalability.**

3. Cacheable:

Data (resources) need to be labeled as cacheable or non-cacheable.

- **Improve network performance.**

3. Architectural constraints for REST (2/2)

4. Uniform interface:

Uniformity of interfaces between components of a distributed application is a central feature of REST.

Uniform interface means that resources are identified in a standard (uniform) way. Resources are manipulated with standard methods.

- **Simplified architecture.**
- **Decoupling of application (service) from interface.**
- **Also called HATEOAS – Hypermedia As The Engine Of Application State**

5. Layered system:

Layers (or tiers) are aimed at the decomposition of the system functionality.

- **Encapsulation of functionality in layers (e.g. encapsulation of legacy services behind a standard interface).**
- **Decomposition of system functionality into client, server and intermediary.**

6. Code on demand:

This constraint is optional for a REST-style system. Code on demand means the dynamic download and execution of code on the client (Javascript etc.).

- **Extensibility (e.g. extension of client with scripting code downloaded from the service).**

4. REST 'protocol' (1/5)

REST is not a protocol like SOAP.
But REST defines some core characteristics that make a system REST-ful.
N.B.: REST does not define something new, it simply makes use of existing protocols and standards (HTTP, URI).

Addressing resources:
REST uses plain *URIs* (actually URLs) to address and name resources.

Access to resources:
Unlike RPC-WS where the access method (CRUD) is mapped to and smeared over SOAP messages, REST uses the available HTTP methods as a resource interface:

Create (C)	**→ HTTP POST**
Read (R)	**→ HTTP GET**
Update (U)	**→ HTTP PUT**
Delete (D)	**→ HTTP DELETE**

REST assumes the methods GET, HEAD, PUT, DELETE to be idempotent (invoking the method multiple times on a specific resource has the same effect as invoking it once) as defined in RFC2616 (page 51).

REST assumes the methods GET and HEAD to be safe (do not change the resource's state on the server, i.e. resource will not be modified or deleted) as defined in RFC2616 (page 51).

4. REST 'protocol' (2/5)

Resource representations:

REST uses standard resource representations like HTML, XML, JSON, GIF, JPEG. Commonly used representations are XML and JSON (preferable to XML if the data needs to be transferred in a more compact and readable form).

Media types:

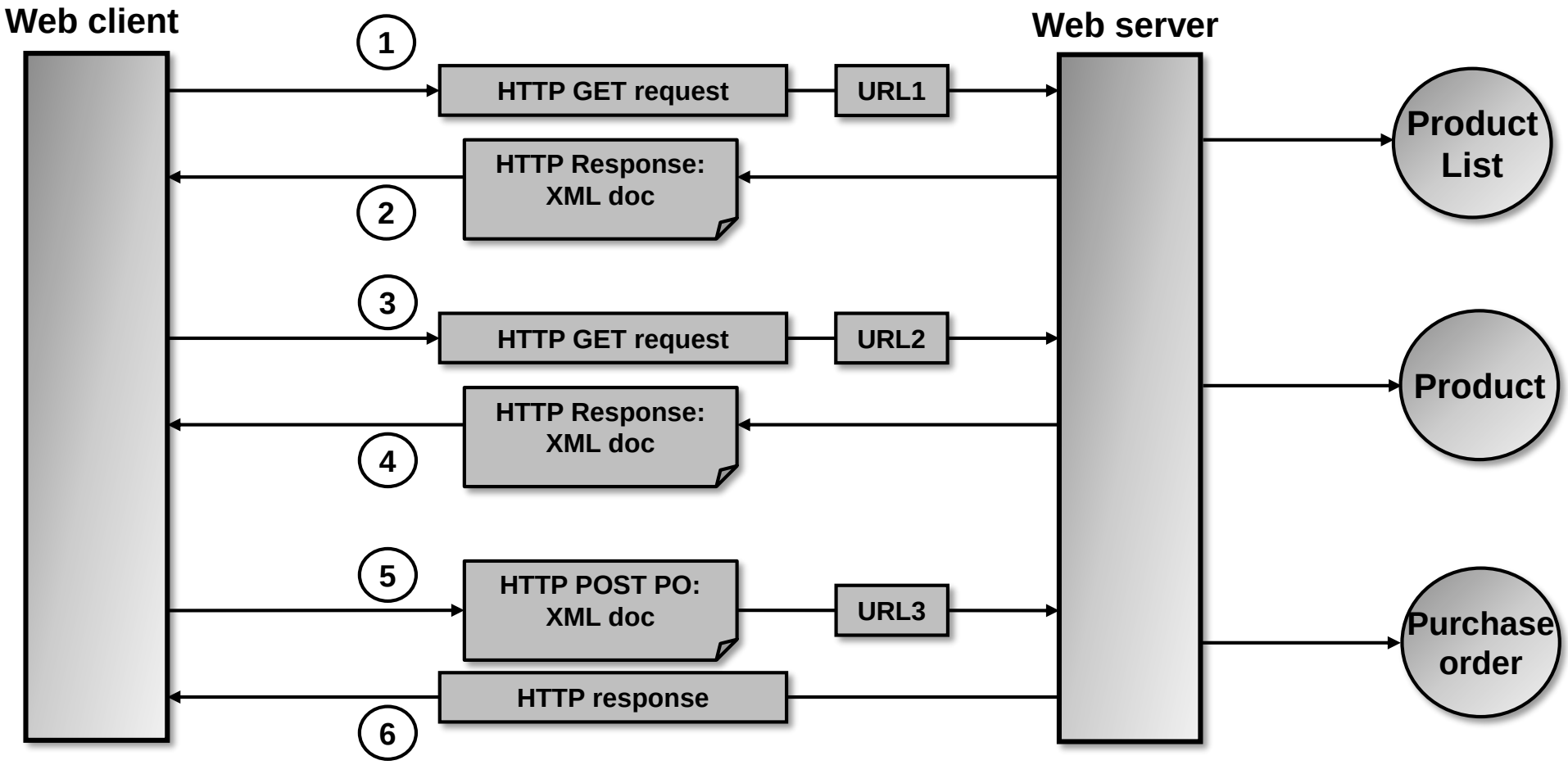
REST uses the HTTP header *Content-type* (MIME types like text/html, text/plain, text/xml, text/javascript for JSON etc.) to indicate the encoding of the resource.

State:

Application state is to be maintained on the client. The server does not have to maintain a state variable for each client (this improves scalability).
Resource state (resource creation, update, deletion), however, is maintained on the server.

4. REST 'protocol' (3/5)

Example of a REST-ful access (1/3):



4. REST 'protocol' (4/5)

Example of a REST-ful access (2/3):

1. & 2. product list request:

The client requests a product list that is available under the URL *http://www.cool-products.com/products&flavor=xml* (URL1).
The response contains the resource encoded in XML.

3. & 4. product selection:

The client application (or the user in front of the browser) selects product 00345 by placing a request for the URL *http://www.cool-products.com/products/00345&flavor=xml* (URL2).
The response contains an XML representation of the product info for product 00345.

5. & 6. placing a purchase order:

The client application places a purchase order (PO) for product 00345 by requesting the resource under the URL *http://www.cool-products.com/products/00345/PO?quantity=7* (URL3).
The purchase order contains additional information entered on the client (customer name etc.).
Therefore the request is a POST accompanied by the XML representation of the purchase order.

4. REST 'protocol' (5/5)

Example of a REST-ful access (3/3):

Products list:

```
<?xml version="1.0"?>
<p:Products xmlns:p="http://www.cool-products.com"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation=
                            "http://www.cool-products.com
                             http://www.cool-products.com/products.xsd">
    <Product id="00345" xlink:href="http://www.cool-products.com/products/00345"/>
    <Product id="00346" xlink:href="http://www.cool-products.com/products/00346"/>
    <Product id="00347" xlink:href="http://www.cool-products.com/products/00347"/>
    <Product id="00348" xlink:href="http://www.cool-products.com/products/00348"/>
</p:Products>
```

Product list contains links to get detailed information about each product (e.g. using XLink). This is a core feature of REST.

Product 00345 info:

```
<?xml version="1.0"?>
<p:Product xmlns:p="http://www.cool-products.com"
              xmlns:xlink="http://www.w3.org/1999/xlink"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation=
                          "http://www.cool-products.com
                           http://www.cool-products.com/product.xsd">
    <Product-ID>00345</Product-ID>
    <Name>Widget-A</Name>
    <Description>This product is made of disposable materials</Description>
    <Specification xlink:href="http://www.cool-products.com/products/00345/specification"/>
    <PurchaseOrder xlink:href=http://www.cool-products.com/products/00345/po/>
    <UnitCost currency="CHF">1.40</UnitCost>
    <Quantity>10</Quantity>
</p:Product>
```

Another URL is provided in the product XML response for placing purchase orders.

5. REST versus SOAP (1/5)

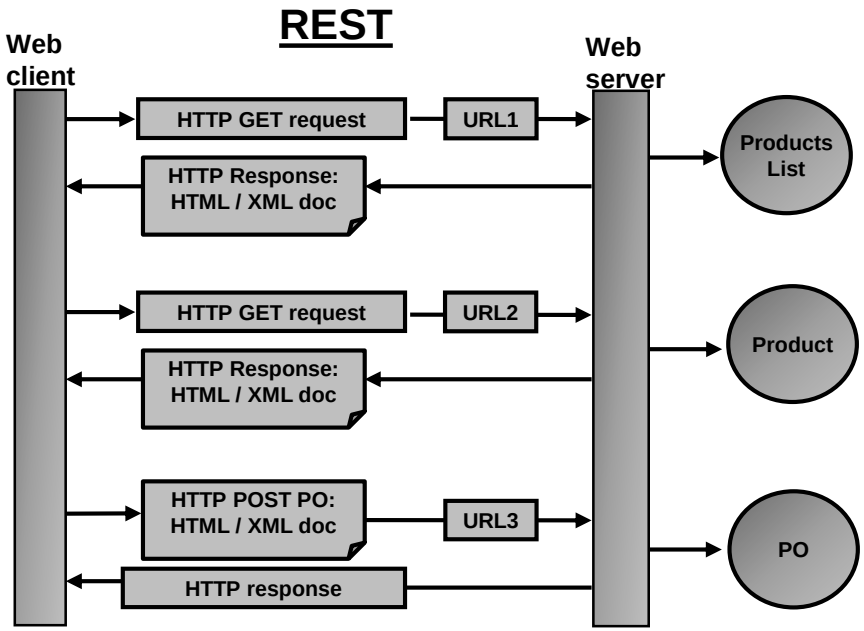
Comparison table:

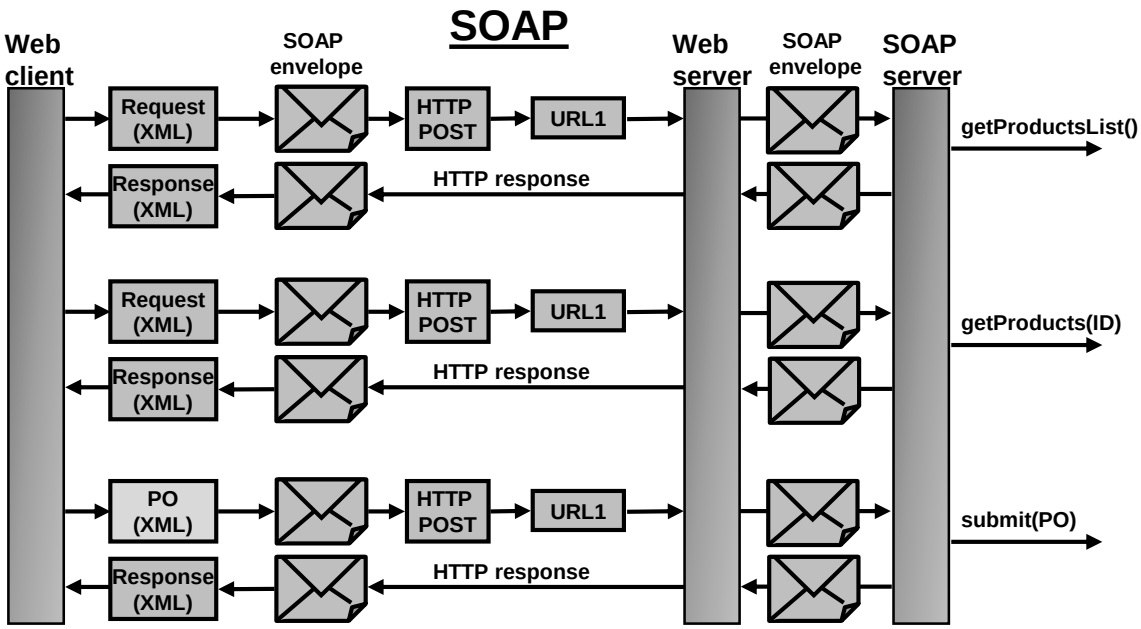
(1) See http://www.w3.org/DesignIssues/Axioms.html

Aspect	SOAP / WSDL (RPC-WS)	REST
Standard	SOAP: W3C SOAP 1.2 WSDL: W3C WSDL 2.0	No standard, makes use of existing standards like RFC2616 HTTP 1.1
Resource addressing	Indirect via SOAP operations	Every resource has its own URL
URL	Only used to address the SOAP processor	Used to address individual resources (data sets)
Error handling	SOAP fault messages	E.g. <Error> element in response, similar to SOAP <fault> element
Data presentation	XML	All encodings defined by HTTP (XML, text, JSON, JPG etc.).
Use of HTTP	Only as transport protocol (envelope)	Actions on resources (CRUD) mapped to HTTP methods (PUT, GET, POST, DELETE)
Compliance with web axioms (1)	Low (does not make much use of web features)	High (see axiom 0 and 1)
State	Stateful (every SOAP action is part of an application defined interaction)	Stateless (requests are “self-contained”, no context saved on server)
Transactions	Supported by SOAP (SOAP header transaction element)	More difficult, but possible e.g. by modelling a transaction as a resource
Registry / service discovery	UDDI / WSDL	None (maybe search engines like Google can be seen as registries of REST web services)
Method	Inside SOAP body	HTTP method
Scoping (which data?)	Inside SOAP body	Part of URL
State transitions of client application	More difficult to determine the next state (SOAP message does not contain data to do this)	Simpler, based on URL (URL ‚points‘to the next state)
Web intermediaries (proxies, caches, gateways etc.)	More difficult since proxies need to peek into SOAP messages to determine the receiver	Simpler since the receiver can be identified by the URL (simple mapping of resource with URL to a receive handler)

5. REST versus SOAP (2/5)

Comparison of architectures:



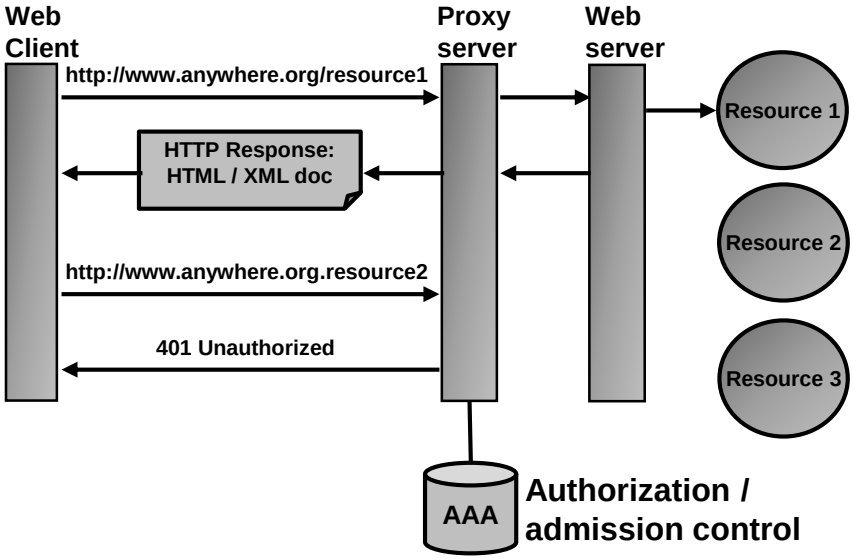


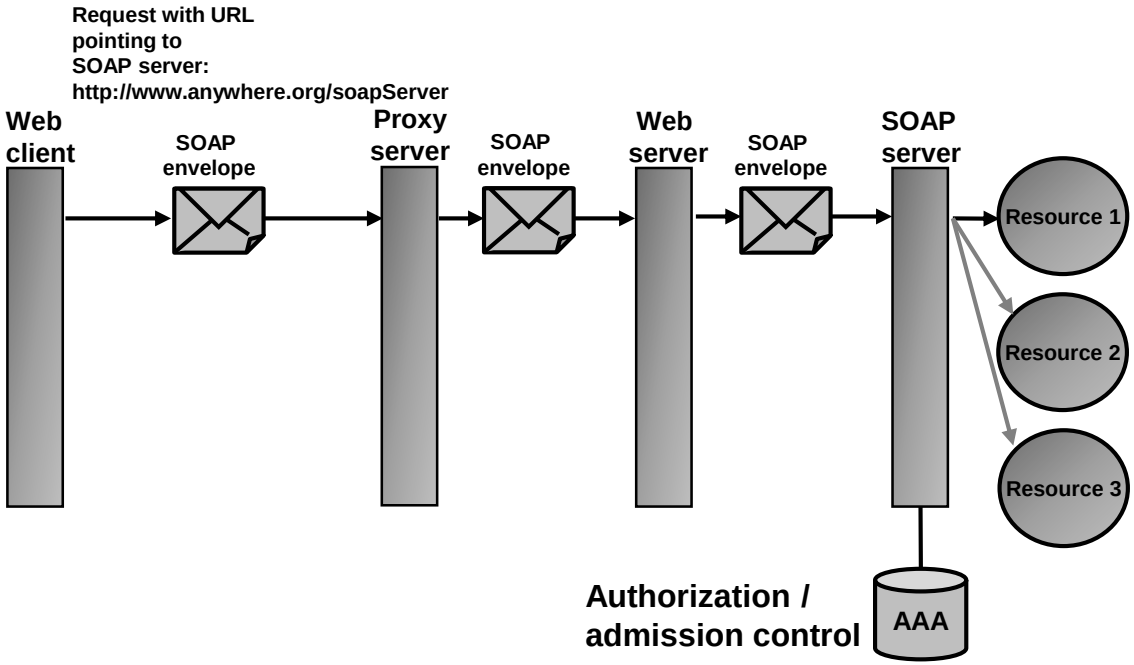
- REST uses different URLs to address resources.
- The web server directly dispatches a request to a handler (URL addresses a handler).
- REST maps the type of access to HTTP methods.

- RPC-WS (SOAP) use the same URL (URL1 in the example) for all interactions.
- The SOAP server parses the SOAP message to determine which method to invoke.
- All SOAP messages are sent with HTTP POST requests.

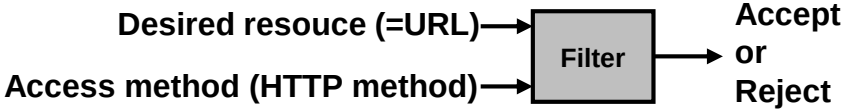
5. REST versus SOAP (3/5)

Use of proxy servers:





- The URL identifies the desired resource.
- The HTTP method identifies the desired operation (GET, PUT, POST, DELETE).
- A proxy can accept / reject access based on the resource identified by a URL and the HTTP method.



- The proxy server cannot determine the target resource from the URL. The URL only addresses the SOAP server.

- A proxy server with filtering / access control needs to look into and understand the SOAP message (semantics) to find out the target resource:

```
<SOAP-ENV:Body>
  <m:GetLastTradePrice xmlns:m="Some-URI">
   <symbol>DEF</symbol>
   </m:GetLastTradePrice>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

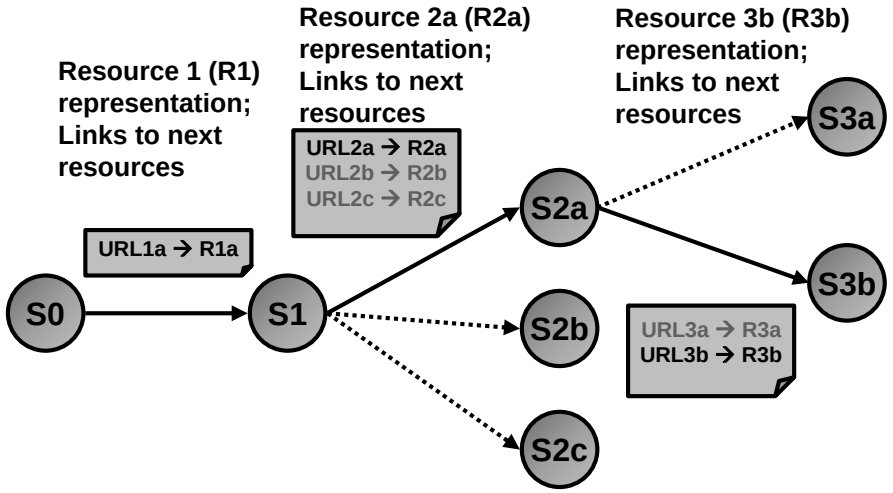
What is the resource? The proxy server must understand the semantics of the SOAP message to know which resource is being accessed.

5. REST versus SOAP (4/5)

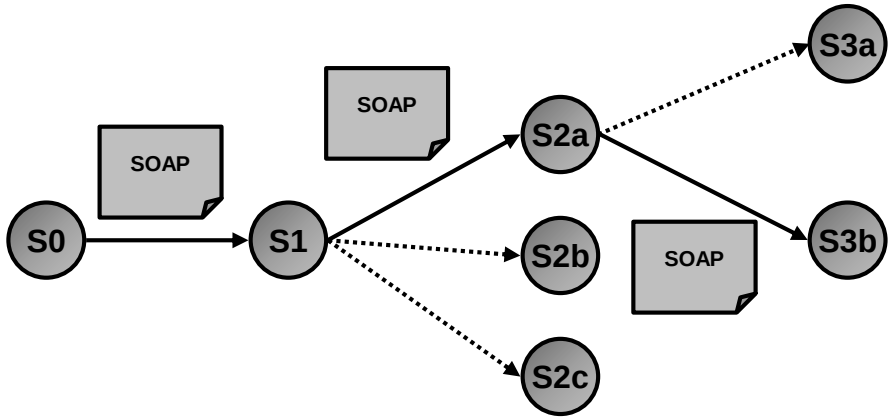
State transfer:

REST is modelled after the well-known interaction model of humans with the web (browsing). The user loads a page, reads it, follows a link to get another page. Each page puts him into another state.

REST applies this simple model and puts much control of the application into the server (links in XML responses guide through the application).



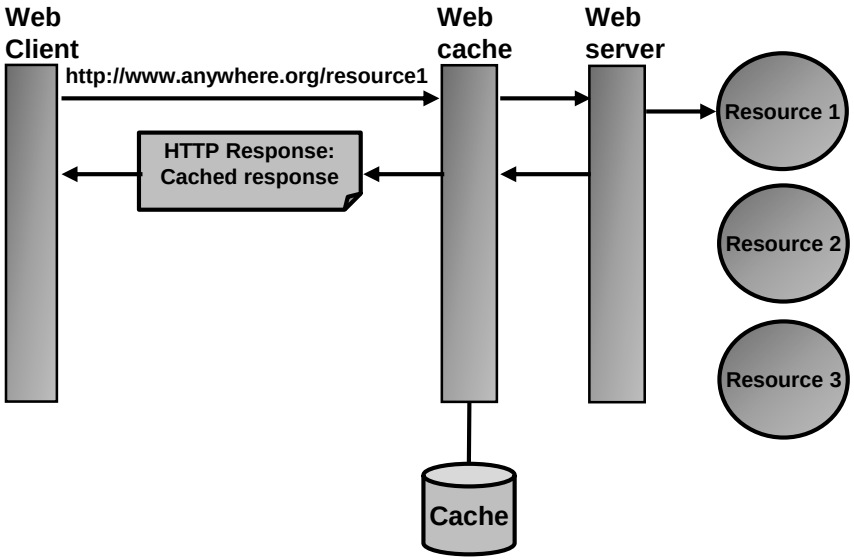
The links in the response XML document ‚points' to the next state.
Using XLink technology (hyperlink markup, see http://www.w3.org/TR/xlink/) allows adding more information about the resource linked to (XLink:role).



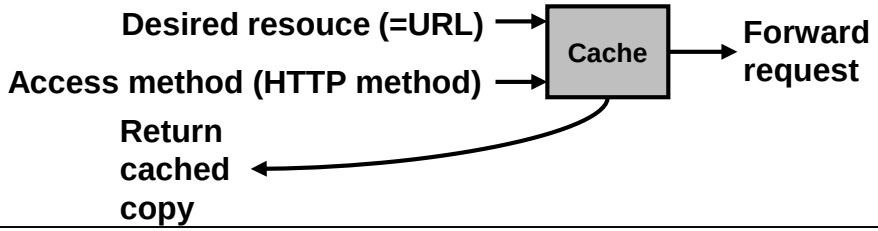
The SOAP messages contain no hyperlinks, only data. The client cannot obtain information on what to do next from the SOAP message but must get this information somewhere else (must be built in into the client application).

5. REST versus SOAP (5/5)

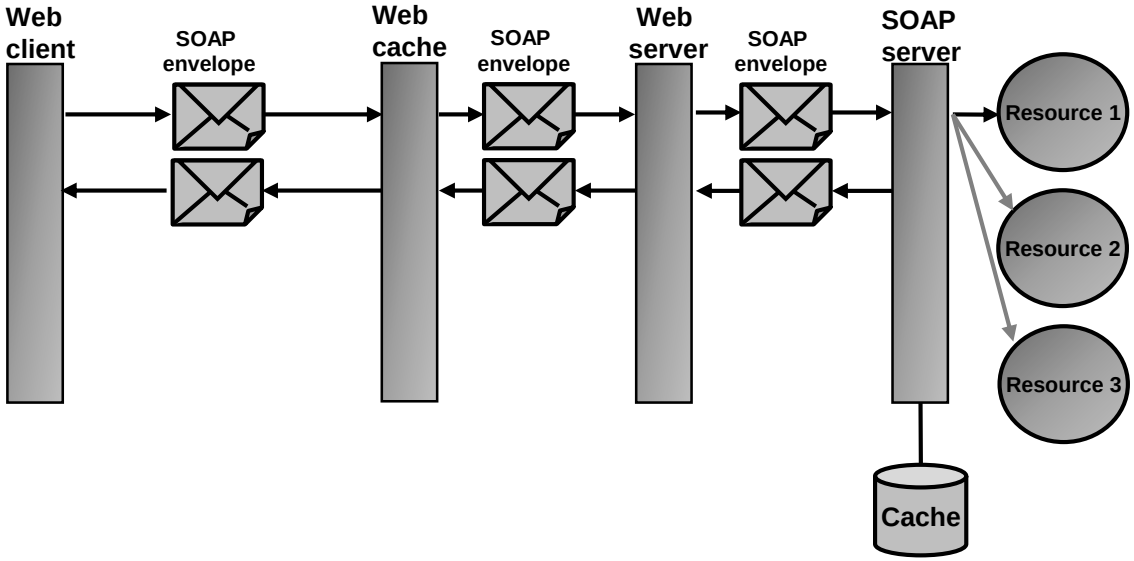
Use of caching (proxy) servers:



• Caching in HTTP is a proven technology to speed up accesses and reduce network load.
• REST may (re)-use the very same cache logic (server must place cacheability information into the HTTP header).



Request with URL pointing to SOAP server: http://www.anywhere.org/soapServer



• The cache cannot directly determine if the resource is cacheable or can be retrieved from the cache.

• It is even worse: The cache cannot even know if the client accesses a resource.

➔ Simple caching proxies can not be used with SOAP.
➔ Only the SOAP server itself can do caching; but this means that the request already consumed network bandwidth, opened an additional connection to the web server and started a new request on the SOAP server.

6. How to organize / manage the resources for REST web services

Does REST require that all resources need to be stored as individual XML files?

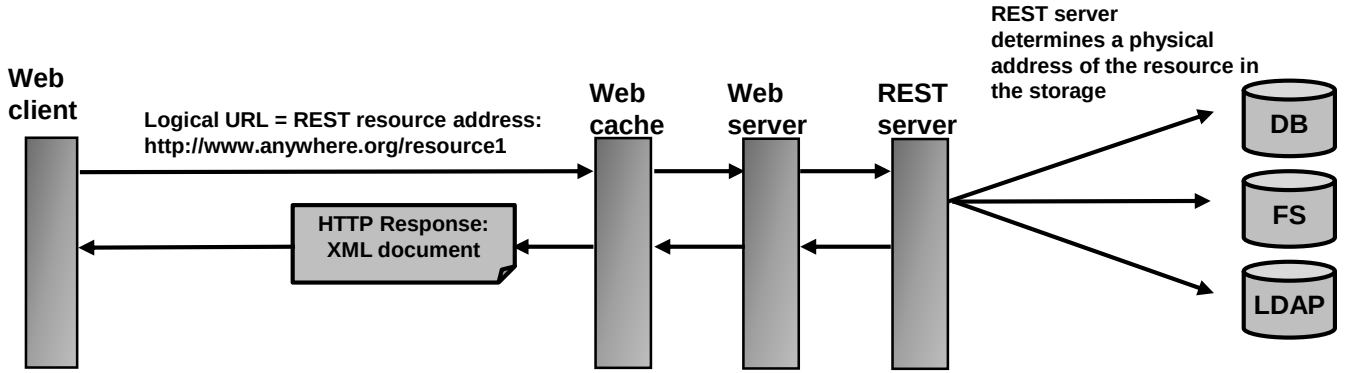
http://www.cool-products/products/000000
http://www.cool-products/products/000001
...
http://www.cool-products/products/999999

→ No!

REST uses „logical“ URLs, i.e. addresses that identify a resource. A resource is physically stored somehow in memory (DB, file, directory service etc.).

→ The underlying implementation of a resource and its storage is transparent to the REST-client.

Resources are converted to XML fragments „on-the-fly“, i.e. the REST-server retrieves the data e.g. from a DB and converts them to XML fragments.



DB Data Base
FS File System
LDAP Lightweight Directory Access Protocol

7. Additional functionality for REST-services

„Basic REST“ puts the scoping information (which data) into the HTTP URL and the method information (what to do with the data) into the HTTP method (GET, PUT, POST, DELETE).

REST may use additional features and signal these with additional fields of the HTTP header.

Function	Used HTTP functionality
Compression of transferred HTTP-body data	HTTP encoding headers: Request header: `Accept-Encoding` Response header: `Encoding`
Cache the responses (have faster access)	HTTP caching headers: Request header: `Etag`, `If-Modified-Since` Response header: `Etag`, `Last-Modified`
Authentication (basic, digest etc.)	HTTP authentication headers: Request header: `Authorization` Response header: `WWW-Authenticate`
Redirect a request to another server (e.g. service has moved)	HTTP redirect: HTTP return code 301 ‚Moved Permanently‘ with new URL REST-WS client has to avoid redirect loops (e.g. give up after 10 redirects).

8. Formal description of REST services (1/4)

Web services may use description files that formally define / describe the service they offer.

RPC-WS (SOAP): WSDL (Web Service Description Language)
REST-WS:
a. „Human-readable “ description (e.g. http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html) or
b. WADL (Web Application Description Language) or
c. WSDL (WSDL 2.0 allows to describe REST-WS)

WADL (1/2):

WADL is not an official (W3C, OASIS) standard.
WADL was invented by Sun Microsystems for the Java ecosystem.
There is a WADL specification under https://wadl.java.net/.

Critique of WADL and WSDL 2.0 for REST:

- **Formal description of service allows better machine processing (e.g. code generation of stubs).**
- **Additional layer of complexity (REST services becomes more complex).**

8. Formal description of REST services (2/4)

b. WADL (2/2):

WADL example 1 for a REST-WS: The ever-popular stock service

```
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://research.sun.com/wadl/2006/10 wadl.xsd"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:ex="http://www.example.org/types"
     xmlns="http://research.sun.com/wadl/2006/10">

     <grammars>
        <include href="ticker.xsd"/>
     </grammars>

     <resources base="http://www.example.org/services/">
        <resource path="getStockQuote">
           <method name="GET">
                <request>
                   <param name="symbol" style="query" type="xsd:string"/>
                </request>
                <response>
                   <representation mediaType="application/xml"
                       element="ex:quoteResponse"/>
                   <fault status="400" mediaType="application/xml"
                       element="ex:error"/>
                </response>
           </method>
        </resource>
     </resources>
</application>
```

Source: http://www.ajaxonomy.com/2008/xml/web-services-part-2-wsdl-and-wadl

WADL Example 2: Yahoo news search service (see http://weblogs.java.net/blog/mhadley/archive/2005/05/introducing_wad.html)

8. Formal description of REST services (3/4)

c. WSDL example for REST-WS (1/2):

```
<wsdl:description xmlns:wsdl="http://www.w3.org/ns/wsdl"
   targetNamespace="http://www.bookstore.org/booklist/wsdl"
   xmlns:tns="http://www.bookstore.org/booklist/wsdl"
   xmlns:whttp="http://www.w3.org/ns/wsdl/http"
   xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msg="http://www.bookstore.org/booklist/xsd">
  <wsdl:documentation>
     This is a WSDL 2.0 description of a sample bookstore service
     listing for obtaining book information.
  </wsdl:documentation>

  <wsdl:types>
    <xs:import namespace="http://www.bookstore.org/booklist/xsd"
       schemaLocation="booklist.xsd"/>
  </wsdl:types>

  <wsdl:interface name="BookListInterface">
    <wsdl:operation name="getBookList"
       pattern="http://www.w3.org/ns/wsdl/in-out"
       style="http://www.w3.org/ns/wsdl/style/iri"
       wsdlx:safe="true">
      <wsdl:documentation>
         This operation returns a list of books.
      </wsdl:documentation>
      <wsdl:input element="msg:getBookList"/>
      <wsdl:output element="msg:bookList"/>
    </wsdl:operation>
  </wsdl:interface>
```

Definition of book-list XML response format

Request-response messaging pattern

8. Formal description of REST services (4/4)

c. WSDL example for REST-WS (2/2):

```
 <wsdl:binding name="BookListHTTPBinding"
    type="http://www.w3.org/ns/wsdl/http"
    interface="tns:BookListInterface">
   <wsdl:documentation>
      The RESTful HTTP binding for the book list service.
   </wsdl:documentation>
   <wsdl:operation ref="tns:getBookList" whttp:method="GET"/>
 </wsdl:binding>

 <wsdl:service name="BookList" interface="tns:BookListInterface">
   <wsdl:documentation>
      The bookstore's book list service.
   </wsdl:documentation>
   <wsdl:endpoint name="BookListHTTPEndpoint"
      binding="tns:BookListHTTPBinding"
      address="http://www.bookstore.com/books/">
   </wsdl:endpoint>
 </wsdl:service>
</wsdl:description>
```

REST-WS is bound to the HTTP-protocol

The operation `getBookList` is bound to the HTTP GET method

Address of booklist

Source: https://www.ibm.com/developerworks/webservices/library/ws-restwsdl/

9. ROA – Resource Oriented Architecture (1/3)

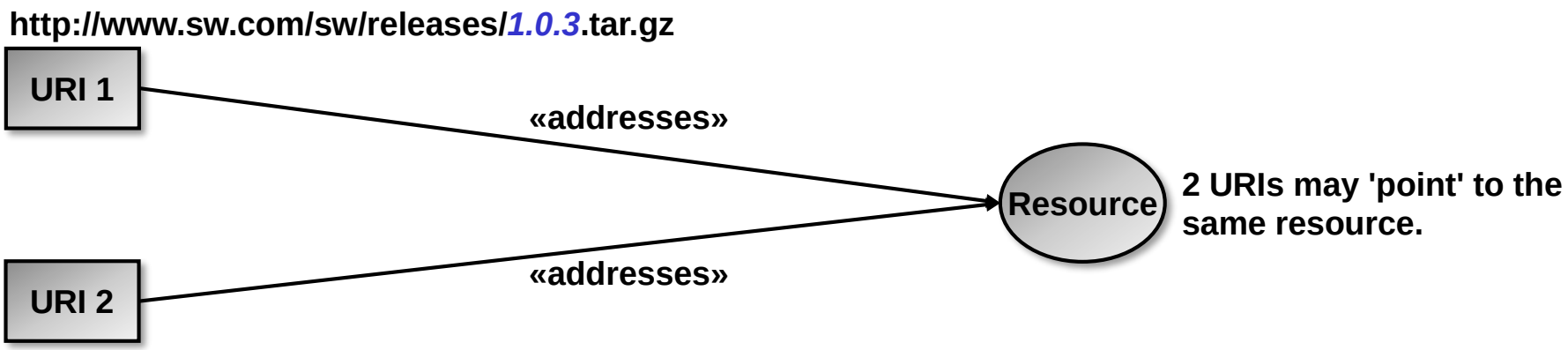
ROA is similar to SOA, but uses REST-style web services instead of SOAP.

SOA → SOAP WS:

The service as a collection of operations and message types is central to SOA.
The data (resources) are accessible indirectly via SOAP operations.

ROA → REST WS:

REST defines a set of design criteria while ROA is a set of architectural principles.
A resource is a unit of data that is worth to be addressed, accessed and processed individually (e.g. a document, a row in a DB, the result of a calculation etc.).
A resource is identified and addressed by one or multiple URIs.



9. ROA – Resource Oriented Architecture (2/3)

ROA principles (1/2):

1. Addressability:
All data is exposed as a resource with an URI.

2. Statelessness:
The context of access (e.g. result page of search) should not be stored as a cookie (cookies are unRESTful). Instead, the context / state should be modelled as an URI as well.
Example:
`/search?q=resource+oriented+architecture&start=50` **(page is part of URI).**
N.B.: HTTP is stateless (unlike e.g. FTP).

3. Uniform interface:
Map request methods uniformly to HTTP methods (GET, PUT, DELETE, POST).
A uniform interface should be safe and idempotent:
GET, HEAD → Safe (resources and server state are not changed).
GET, HEAD, PUT, DELETE → Idempotent (method can be called multiple times).

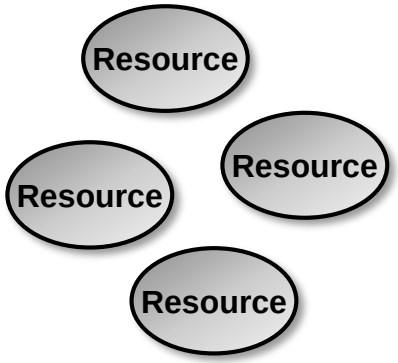
9. ROA – Resource Oriented Architecture (3/3)

ROA principles (2/2):

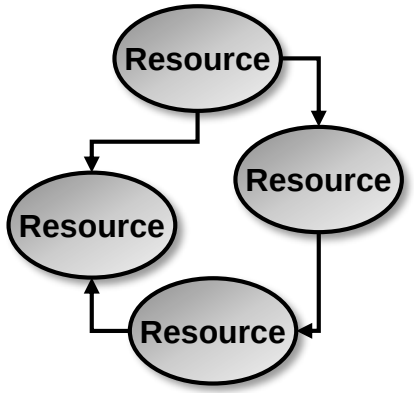
4. Connectedness:

In ROA, resources should link to each other in their representations (resources are 'connected' to each other).

RPC-style WS:
Everything is addressed through a single URI.



REST-hybrid WS:
WS with addressable but not connected resources (no links between resources).
Example: Amazon S3 WS.



Fully RESTful:
WS with addressable and connected resources.

10. Complex data queries with REST

REST allows doing more complex data (resource) queries by mapping the HTTP methods to SQL commands.

Mapping REST to SQL like queries:

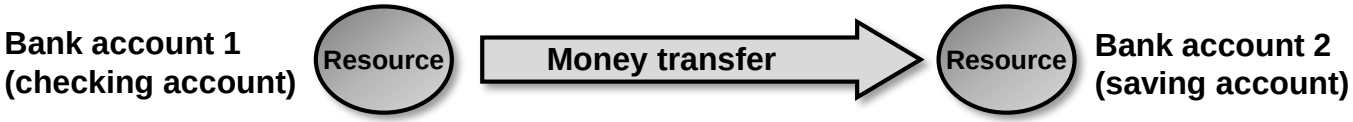
HTTP GET	→	**SQL SELECT (retrieve data)**
HTTP PUT	→	**SQL UPDATE (store data)**
HTTP DELETE	→	**SQL DELETE (delete data)**
HTTP POST	→	**SQL INSERT (create data)**

N.B.: The mapping is of course not 1:1 because SQL offers many more possibilities regarding data manipulation (select from multiple tables, join records from multiples tables etc.).
The REST interface has to be clearly defined, but must not get too complicated (otherwise it would not be RESTful anymore!).

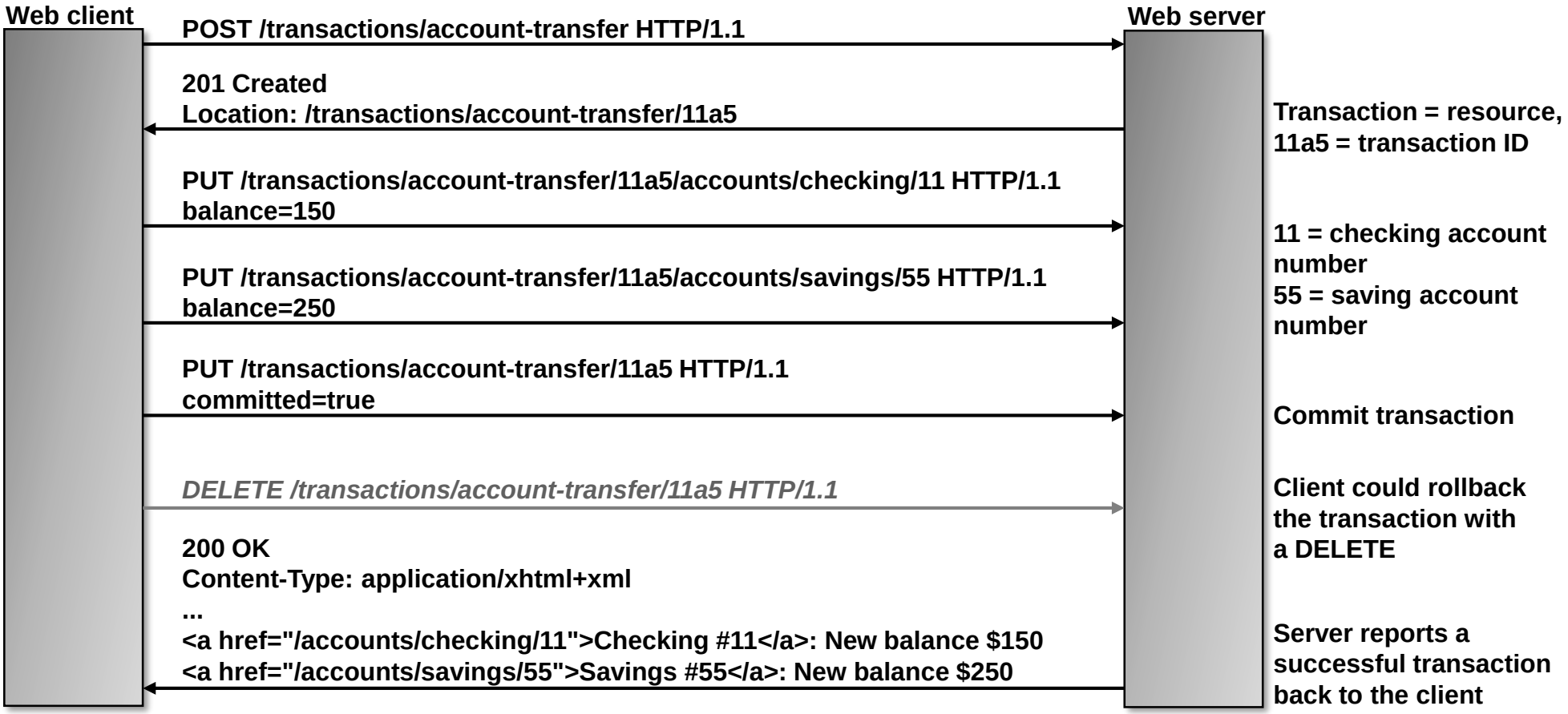
Example library for a RESTful SQL interface see http://phprestsql.sourceforge.net/.

11. Transactions with REST

Typical example that requires a transaction:



Solution with REST: Model transaction as a resource.



12. Authentication with REST (1/3)

REST web services typically make use of the defined HTTP (RFC2616) authentication mechanisms.

The HTTP Authorization header is extensible so any authentication mechanism is possible, even proprietary schemes as exemplified by Amazon's AWS signature (see below).

HTTP client authentication mechanisms (non-exhaustive list):

a. Basic authentication (RFC2617)
b. Digest access authentication (RFC2617)
c. Proprietary (e.g. Amazon AWS signature)

HTTP server and mutual authentication:

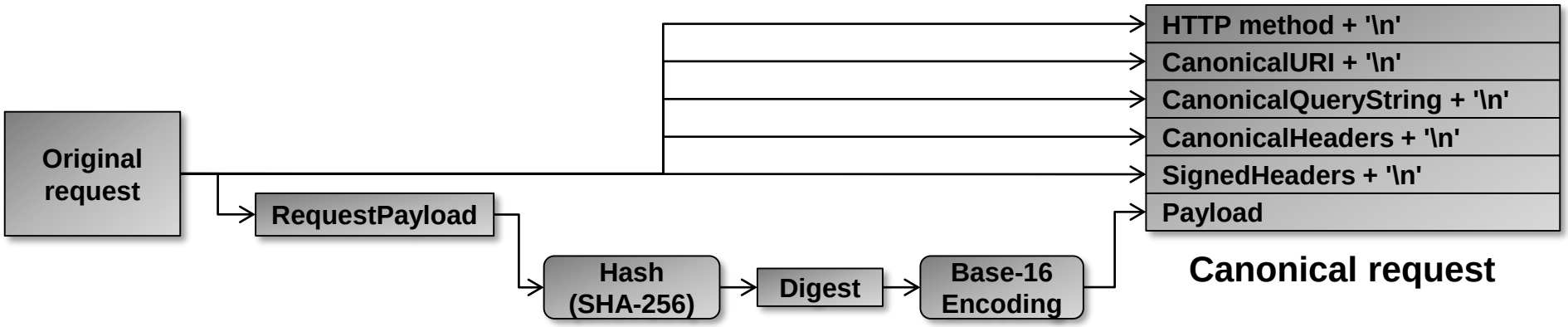
Digest access authentication (RFC2617) provides some degree of server authentication (server authenticates itself to the client).
However, for true mutual authentication and security, HTTPs (TLS) is a better choice.

12. Authentication with REST (2/3)

Amazon AWS authentication / digital signature creation (AWS signature version 4):

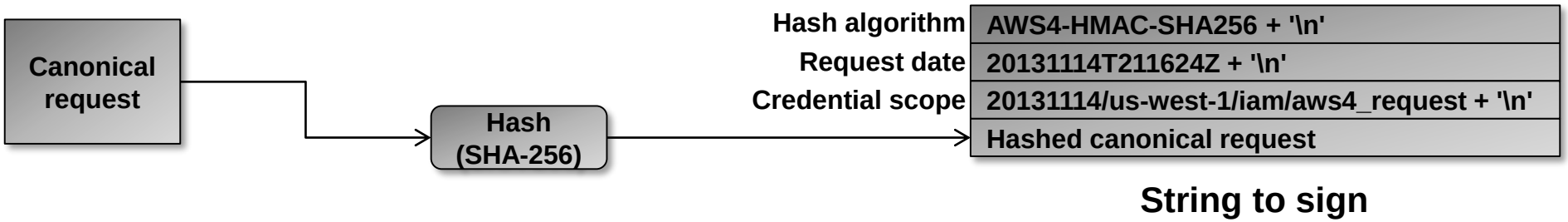
Step 1:

Normalize request (canonical request) so that AWS "sees" the same request and calculates the same signature.



Step 2:

Create string to sign.

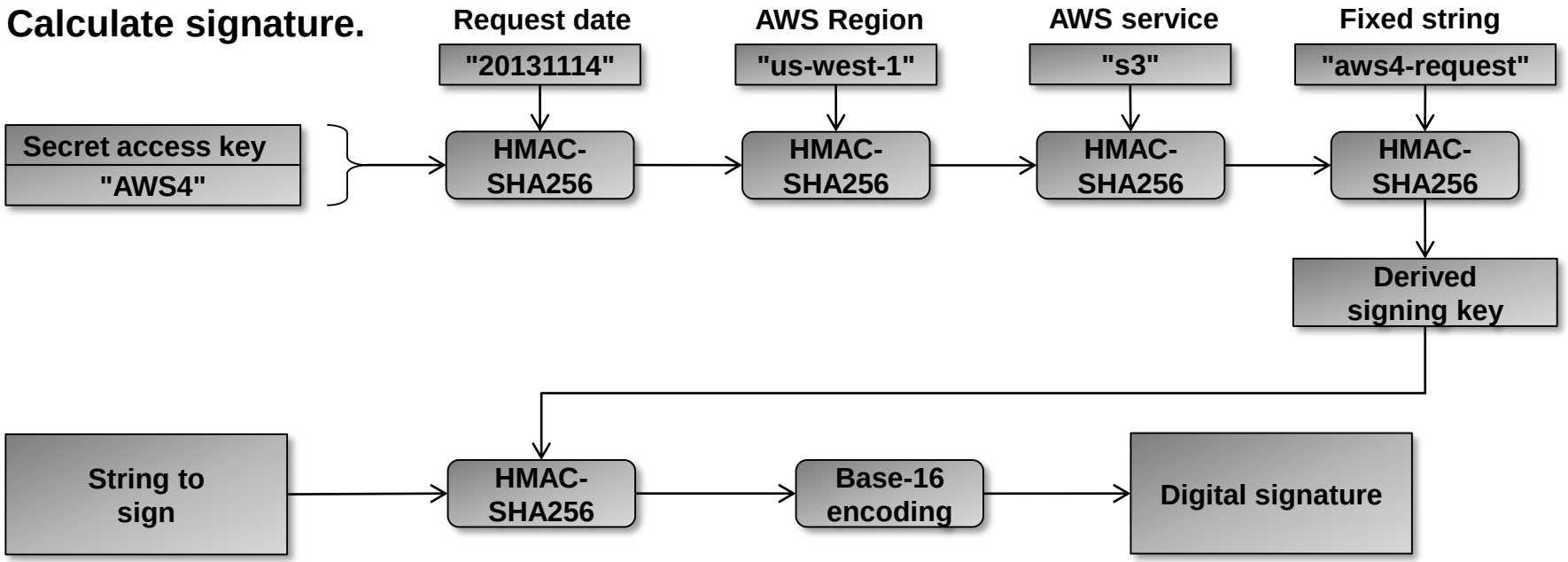


12. Authentication with REST (3/3)

Amazon AWS authentication / digital signature creation (AWS signature version 4):

Step 3:

Calculate signature.



Sample AWS request:

```
POST http://s3.amazonaws.com/ HTTP/1.1
Authorization: AWS4-HMAC-SHA256 Credential=AKIDEXAMPLE/20110909/us-east-1/s3/aws4_request,
SignedHeaders=content-type;host;x-amz-date,
Signature=ced6826de92d2bdeed8f846f0bf508e8559e98e4b0199114b84c54174deb456c
host: s3.amazonaws.com
Content-type: application/x-www-form-urlencoded; charset=utf-8
x-amz-date: 20110909T233600Z
```

13. Examples of REST services

1. Amazon S3 service (http://aws.amazon.com/s3/)

2. Atom publishing protocol (news feeds, see RFC5023)

3. Flickr API (see https://www.flickr.com/services/api/)

4. Yahoo web search (https://developer.yahoo.com/)

5. DynDNS IP address updates (http://dyn.com)

6. Google search API (see http://googlesystem.blogspot.ch/2008/04/google-search-rest-api.html).
N.B.: Google deprecated the SOAP API, see also groups.google.com.