REST

Part 2 – HTTP Methods

REST

- REST is an architectural style and as such it is abstract.
- The best-known implementation of REST principles today is the Web

  i.e. the application protocol HTTP, the naming standard URI and the Internet Media Types.
- These topics are now discussed with respect to how they implement the constraints of REST.

HTTP

- HTTP is a text-based application protocol that has ubiquitous deployment on the Web. The latest version is 1.1 [HTTP 1.1].

- “RESTful HTTP” is defined as “*using HTTP as intended and as described by the REST principles*” [Tilkov 08b].

HTTP methods

- HTTP specifies various methods as part of the uniform interface constraint.
- These methods have well-defined semantics as per the HTTP specification. One is restricted to manipulating resources via these methods only.

HTTP GET

- To retrieve a resource representation, one issues a GET to the representation’s URI. The server sends back a representation in the response entity-body.
- The client can issue a “conditional GET” by including certain request header fields. For example, the client can include the request header *If-Modified-Since* stating the date/time of the latest representation of the requested resource that the client currently has.
- If the server determines that the resource has not changed since the date/time the client provided i.e. the client has an up-to-date version, the server will return a *304 Not Modified* response with no message body, thereby reducing unnecessary network usage [Vinoski 08c].

HTTP HEAD

- "*The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response*" [HTTP 1.1]

  i.e. HEAD gives you exactly what GET would give you, but without the entity-body.

- HEAD is used to retrieve metadata (headers) about the resource without downloading the possibly enormous entity-body.

- A client can use HEAD to test for the existence of a resource and for finding out information about the resource (e.g. does it support an XML representation) without downloading the entire representation.

HTTP OPTIONS

- The OPTIONS method lets the client find out what methods are supported by a resource. The server responds with the Allow header

  e.g. Allow: HEAD, GET (a read-only resource).

HTTP DELETE

- To delete a resource representation, one issues a DELETE to the representation’s URI.

- If the request was successful, the server can either send back a
  - 200 OK message with further status information in the entity-body or a
  - 204 No Content if there is no entity-body returned i.e. success but no entity-body.

HTTP PUT

- "*The PUT method requests that the enclosed entity be stored under the supplied Request-URI*" [HTTP 1.1].
- If the request-URI does not exist, then the server creates the resource with the given URI and the content (of the entity-body) in the message.
- If the request-URI does exist, then the server updates the resource identified by the URI with the content in the message.
- Thus, PUT is used to create/update a resource when the client is in charge of the resource URI to be used. E.g. The client knows the database ID of the resource. [Ruby and Richardson 07].

HTTP POST

- POST can be used in different situations [Ruby and Richardson 07]:
    - To create new resources underneath a parent resource. For example, a client may send a request to create a customer by passing the customer details to a */customers* resource; the server assigns a customer number (e.g. based on some internal database identifier) “2561” to the customer, stores the details in a database (resource state) and sends a 201 Created response with the new URI /customers/2561 in the location header back to the client.
    - Compared with PUT, the server works out the Database ID..

HTTP method properties

- Some of the methods have certain useful properties:
  - **Safety**
    - A method is considered “safe”, if making the request once is the same as making it 10 times or not making it at all. In Mathematics, multiplying by 1 is safe because 4 = 4*1 = 4*1*1*1 [Ruby and Richardson 07].
    - This property applies to the retrieval methods HEAD and GET. Essentially, side-effects (if any) have not been user driven. This does not mean that HEAD or GET cannot have side-effects e.g. log files on the server are often updated and hit counters incremented based on retrieval requests. “*The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them*” [HTTP 1.1].

HTTP method properties

- **Idempotence**
  - A method is considered "idempotent", if making the request once is the same as making it 10 times i.e. "*the second and subsequent requests leave the resource state in exactly the same state as the first request did*" [Ruby and Richardson 07].
  - In Mathematics, multiplying by 0 is idempotent as 4*0 = 4*0*0*0 = 0.
  - However, multiplying by 0 is not safe as 4*0 is not the same as 4.

HTTP method properties

- **Idempotence**
  - HEAD, GET, PUT and DELETE are all idempotent (POST is not).
  - If you DELETE a resource it is gone; if you DELETE it again it is still gone.
  - If you create a new resource with PUT, it is created; if you resend the same PUT request, the resource is still there and it has the same properties as when you created it.
  - If you use PUT to update the resource state, sending the same PUT request results in resource state which is the same as before. Note that PUT has to be programmed properly to avail of idempotence. For example, “setting x to 5” is idempotent whereas “adding 1 to x” is not.

What do RESTful HTTP WS look like?

- Let's say, for example, that we want to design a RESTful HTTP Web Service based on the following user requirements specification:
  - ability to create, read, update and delete (CRUD) a phone
  - provide a facility to retrieve and delete all phone numbers
  - ensure that clients are only dependent on one entry point URI
  - clients should be able to query a URI for its Application Programming Interface (API)

What do RESTful HTTP WS look like?

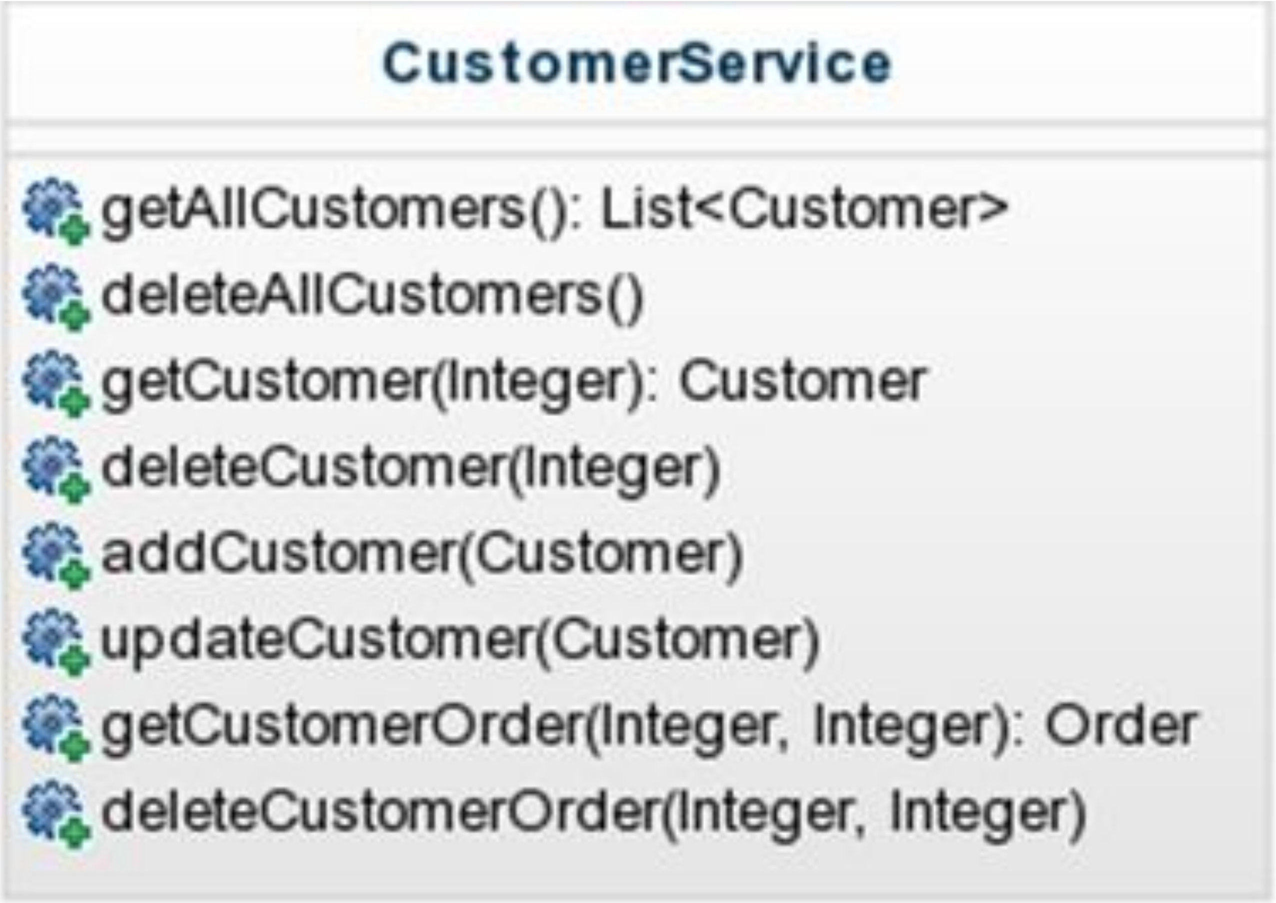
- The question is asking what will the services look like from the viewpoint of Uniform Resource Identifiers (URI’s), verbs, hypermedia etc..

- */phoneDirectory*
  - GET – retrieve all the phones
  - DELETE – delete all the phones
  - POST – add the phone passed in the entity body
  - OPTIONS – returns the verb set supported by this URI (the API)

- */phoneDirectory/{phoneNumber}*
  - GET – retrieve the phone details for {*phoneNumber*}
  - DELETE – delete the phone details for {*phoneNumber*}
  - PUT – update the phone details for {*phoneNumber*} with the phone passed in the entity body
  - OPTIONS – returns the verb set supported by this URI (the API)

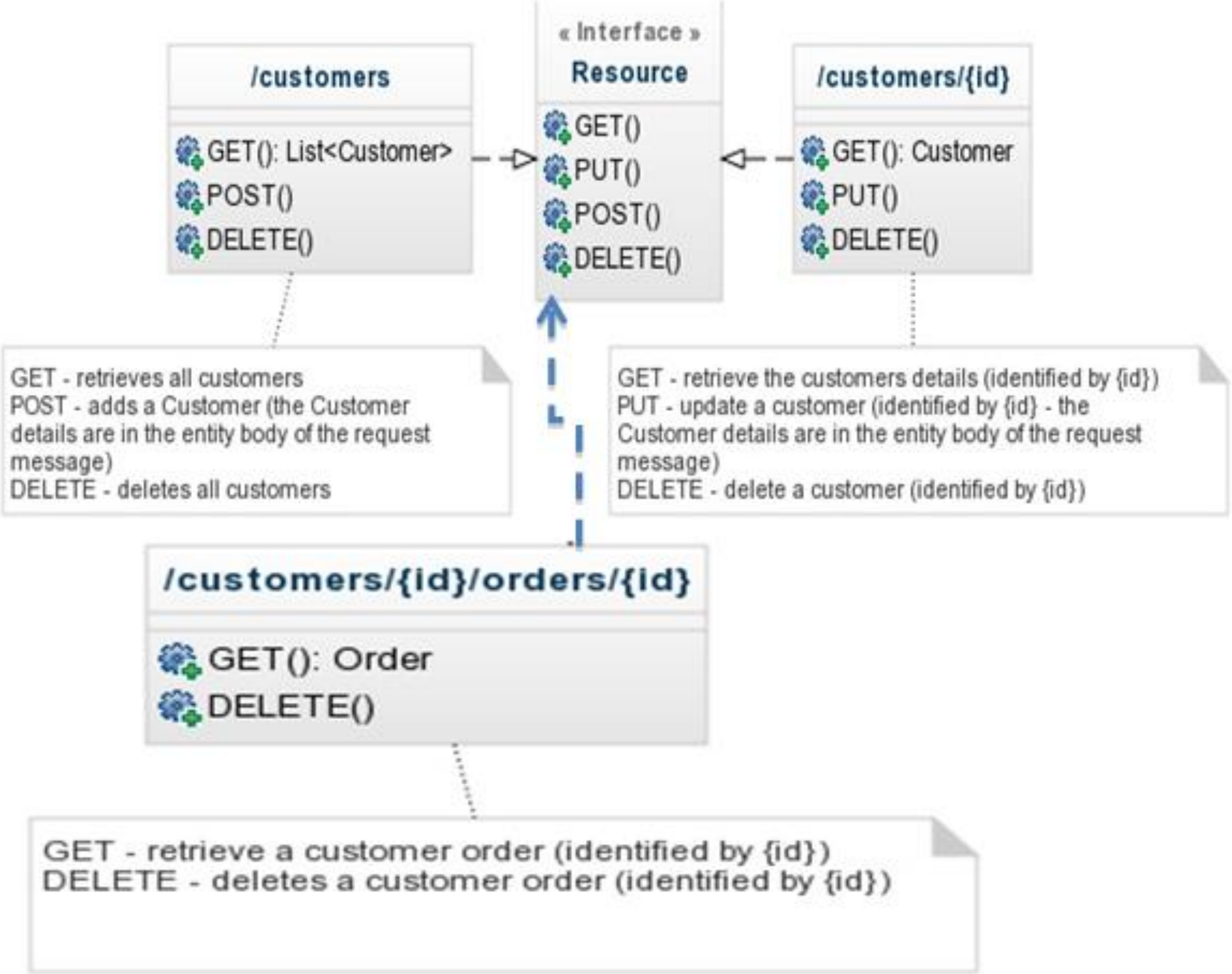
What do RESTful HTTP WS look like?

- Transform the following UML diagram of a WS-* (SOAP) Web Service into its RESTful counterpart.



CustomerService

getAllCustomers(): List<Customer>
deleteAllCustomers()
getCustomer(Integer): Customer
deleteCustomer(Integer)
addCustomer(Customer)
updateCustomer(Customer)
getCustomerOrder(Integer, Integer): Order
deleteCustomerOrder(Integer, Integer)



GET - retrieves all customers
POST - adds a Customer (the Customer details are in the entity body of the request message)
DELETE - deletes all customers

GET - retrieve the customers details (identified by {id})
PUT - update a customer (identified by {id} - the Customer details are in the entity body of the request message)
DELETE - delete a customer (identified by {id})

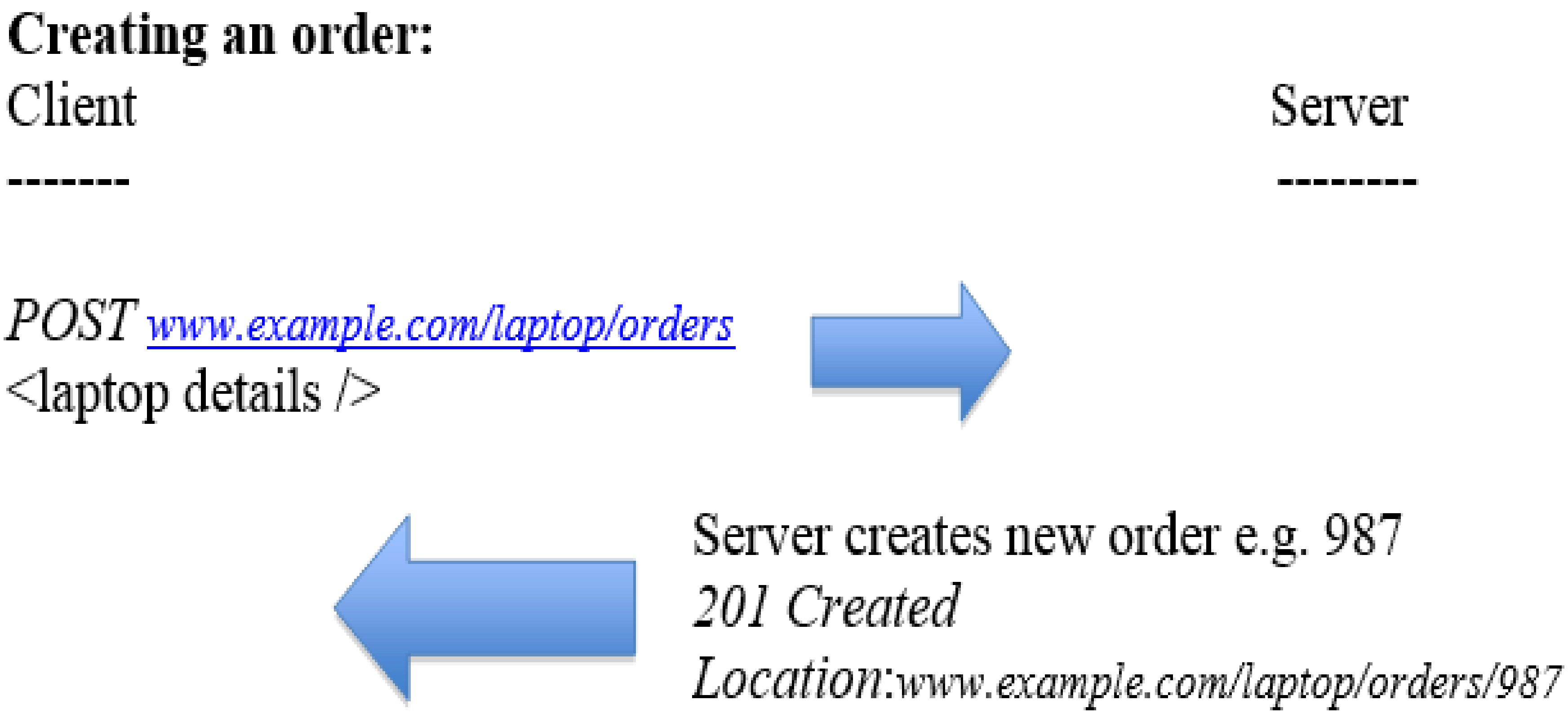
GET - retrieve a customer order (identified by {id})
DELETE - deletes a customer order (identified by {id})

A sample HTTP interaction

- How does HTTP model the following scenario?
  a) a client wishes to place an order for a laptop using the "*www.example.com/laptop/orders*" URI.

  b) subsequently, the client wishes to change the quantity on the order (from 1 to 2 say)

  c) finally, the client has changed his/her mind and wishes to cancel the order created



Creating an order:

Client

Server

POST *www.example.com/laptop/orders*

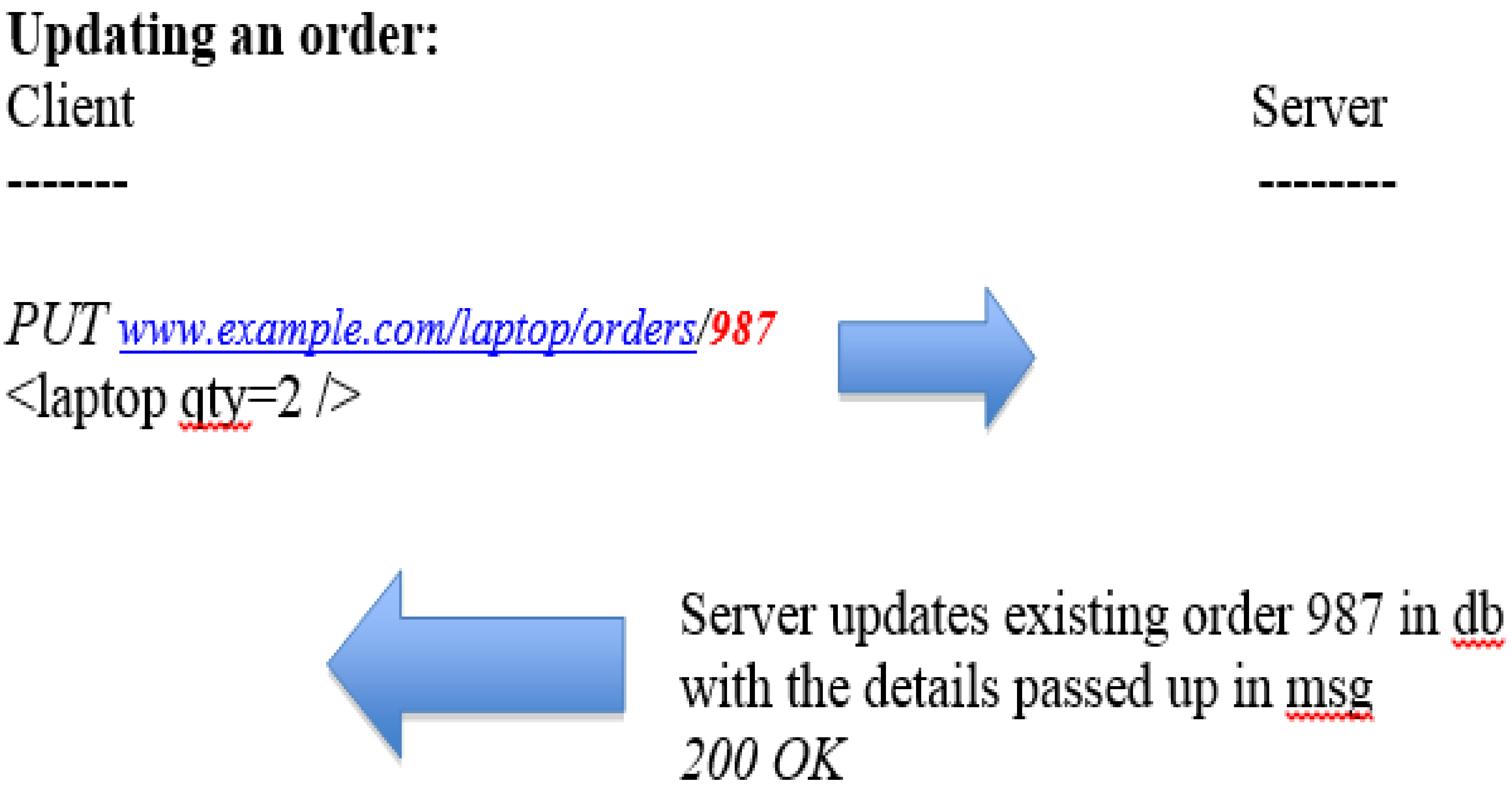
<laptop details />

Server creates new order e.g. 987

201 Created

Location:www.example.com/laptop/orders/987

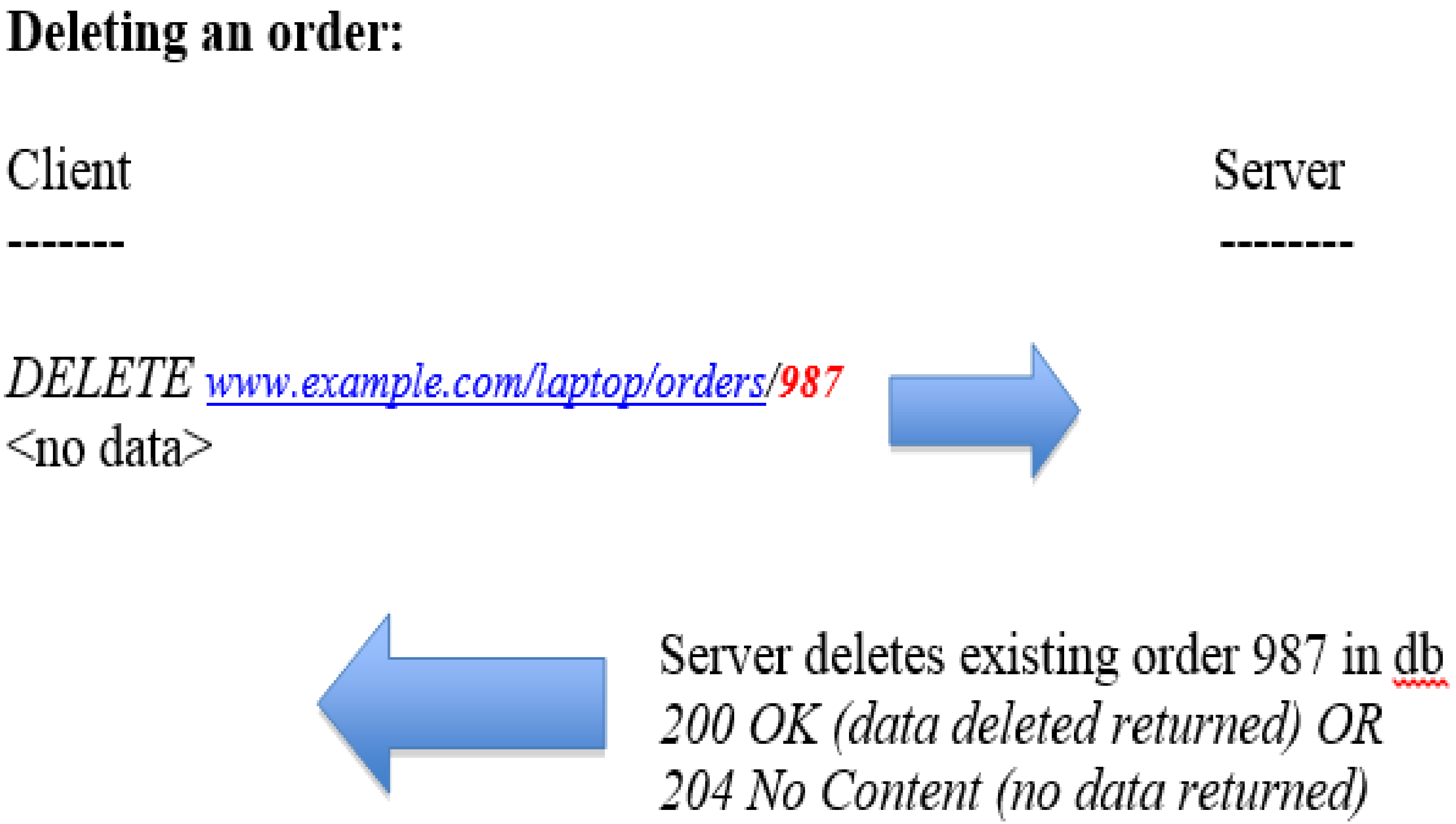
Updating an order:



Client	Server
--------	---------

*PUT www.example.com/laptop/orders/**987***
<laptop qty=2 />

Server updates existing order 987 in db
with the details passed up in msg
200 OK



Deleting an order:

Client

Server

DELETE *www.example.com/laptop/orders/**987***
<no data>

Server deletes existing order 987 in db
200 OK (data deleted returned) OR
204 No Content (no data returned)

Web Service Architectures

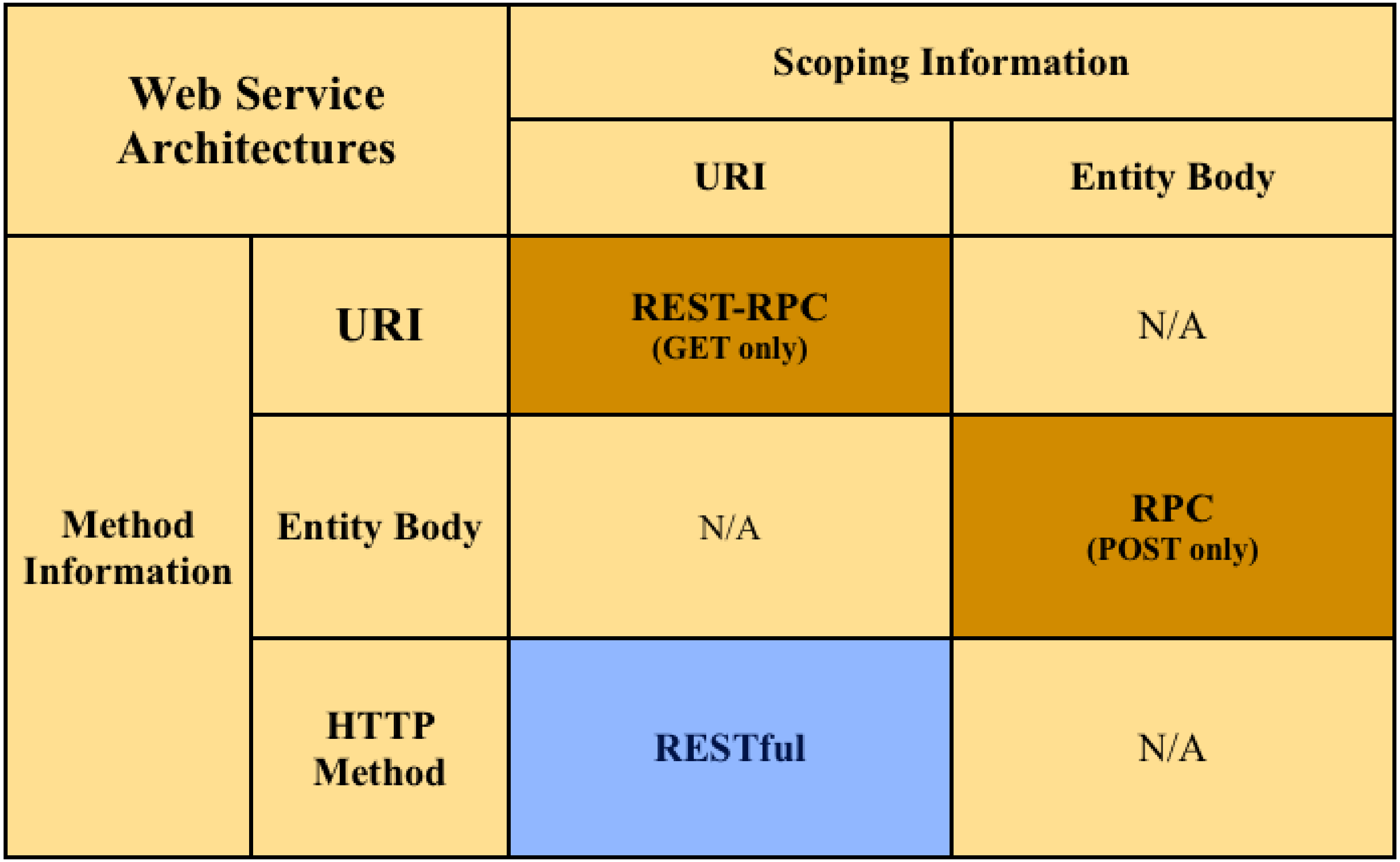
- The criteria are outlined in Ruby and Richardson’s book “RESTful Web Services” and relate to the location in the request of the *method* and *scoping information*.

- The method informs the server what operation the client wants to perform, for example: to retrieve or delete data.

- The scoping information informs the server what data the client wants to operate on, for example: customer data or order data.

- The locations of both the *method* and *scoping information* affect whether a service is considered to be RESTful, RPC or REST-RPC hybrid.

Web Service Architectures



Web Service Architectures		Scoping Information	
		URI	Entity Body
Method Information	URI	REST-RPC (GET only)	N/A
	Entity Body	N/A	RPC (POST only)
	HTTP Method	RESTful	N/A

Web Service Architectures

- RESTful
  - In a RESTful architecture, the method information is in the HTTP method and the scoping information goes into the URI.

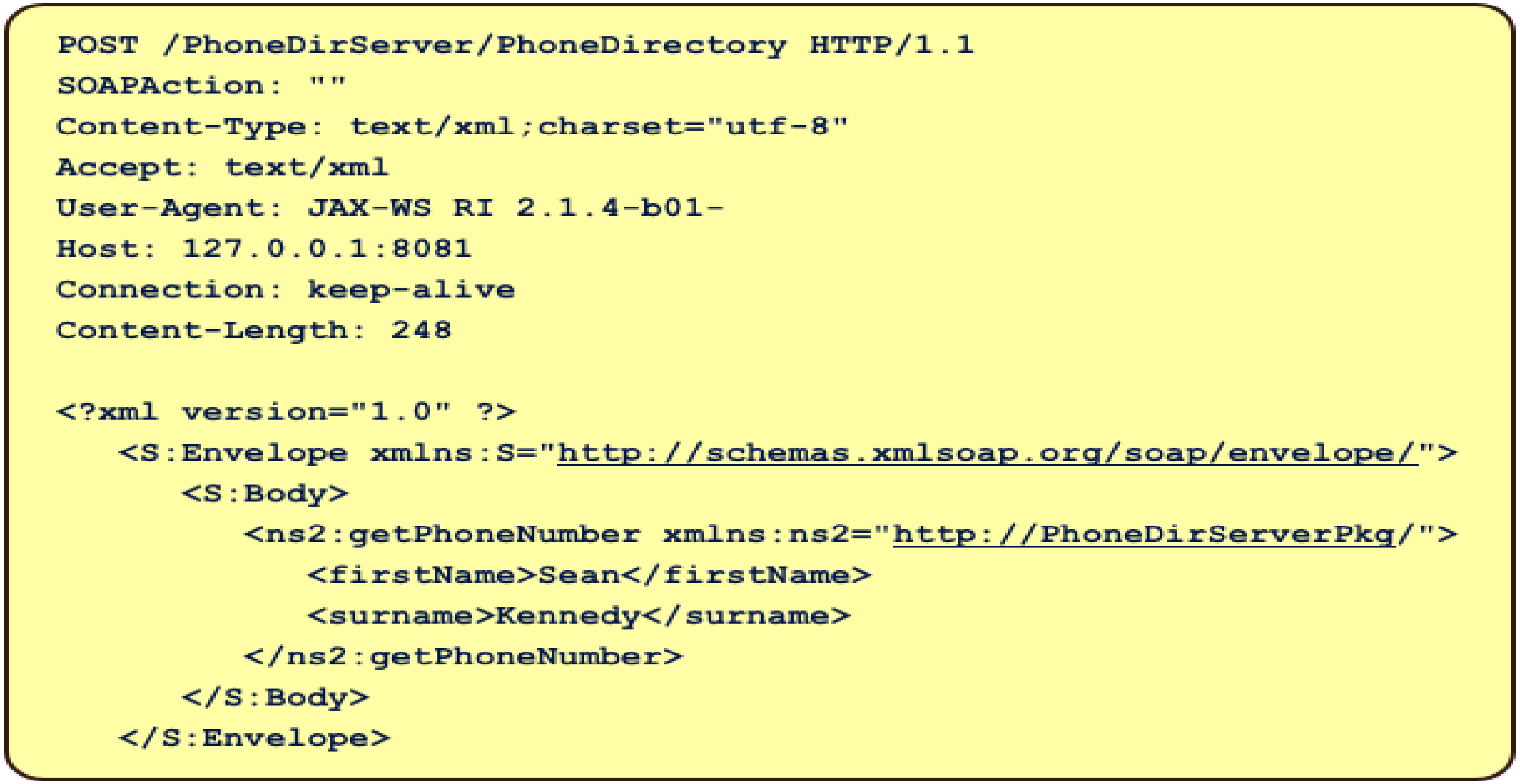
```
GET /BookServer/webresources/books HTTP/1.1
Host: 127.0.0.1:8081
Connection: keep-alive
…
```

Web Service Architectures

- RPC
  - RPC-style services encapsulate the method and scoping information in an envelope. The kinds of envelope are unimportant but SOAP is a popular example.
  - Every RPC-style service defines its own new vocabulary.
  - Only one URI is exposed (the “endpoint”) and POST is the only verb supported. The messages are parsed to elicit the Web Service to execute and its associated parameters.
  - An example of RPC-based Web Services are SOAP Web Services.

Web Service Architectures

- RPC



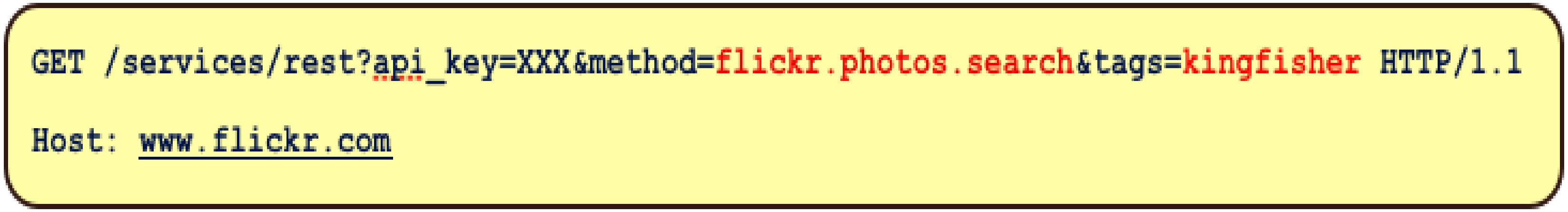
```
POST /PhoneDirServer/PhoneDirectory HTTP/1.1
SOAPAction: ""
Content-Type: text/xml;charset="utf-8"
Accept: text/xml
User-Agent: JAX-WS RI 2.1.4-b01-
Host: 127.0.0.1:8081
Connection: keep-alive
Content-Length: 248

<?xml version="1.0" ?>
   <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
      <S:Body>
         <ns2:getPhoneNumber xmlns:ns2="http://PhoneDirServerPkg/">
            <firstName>Sean</firstName>
            <surname>Kennedy</surname>
         </ns2:getPhoneNumber>
      </S:Body>
   </S:Envelope>
```

Web Service Architectures

- RPC-REST
  - RPC-REST services encapsulate both the method and scoping information *in the URI*.
  - The RPC classification is due to the fact that these services define their own vocabulary and use HTTP as an envelope format, inserting the method and scoping information wherever they please.
  - The method GET is used for *all* messages.

Web Service Architectures

- RPC-REST



```
GET /services/rest?api_key=XXX&method=flickr.photos.search&tags=kingfisher HTTP/1.1

Host: www.flickr.com
```

Note that in the example above, both the method and scoping information are passed in the URI i.e. the method is *flickr.photos.search* and the scoping information is *kingfisher*.

As GET is for retrieving, this message is in fact RESTful. Many read-only Web Services (which use GET) are in fact RESTful even though the method information is passed in the URI.

The issue arises when a client wants to modify the data set.

For example, deleting a photo on Flickr involves making a GET request to a URI that includes *flickr.photos.delete*.

In this scenario, the method semantics of GET conflict with the operation carried out.

The Flickr Web API is in fact a hybrid: RESTful when retrieving data and RPC-style when modifying the data set.