RESTful Web Services

Prof. (Dr.) Vipul K Dabhi
Associate Professor,
Department of Information Technology,
D. D. University, Nadiad

Note: The major content of this presentation is taken from Presentation on REST by Roger L. Costello, Timothy D. Kehoe

Outline of Presentation

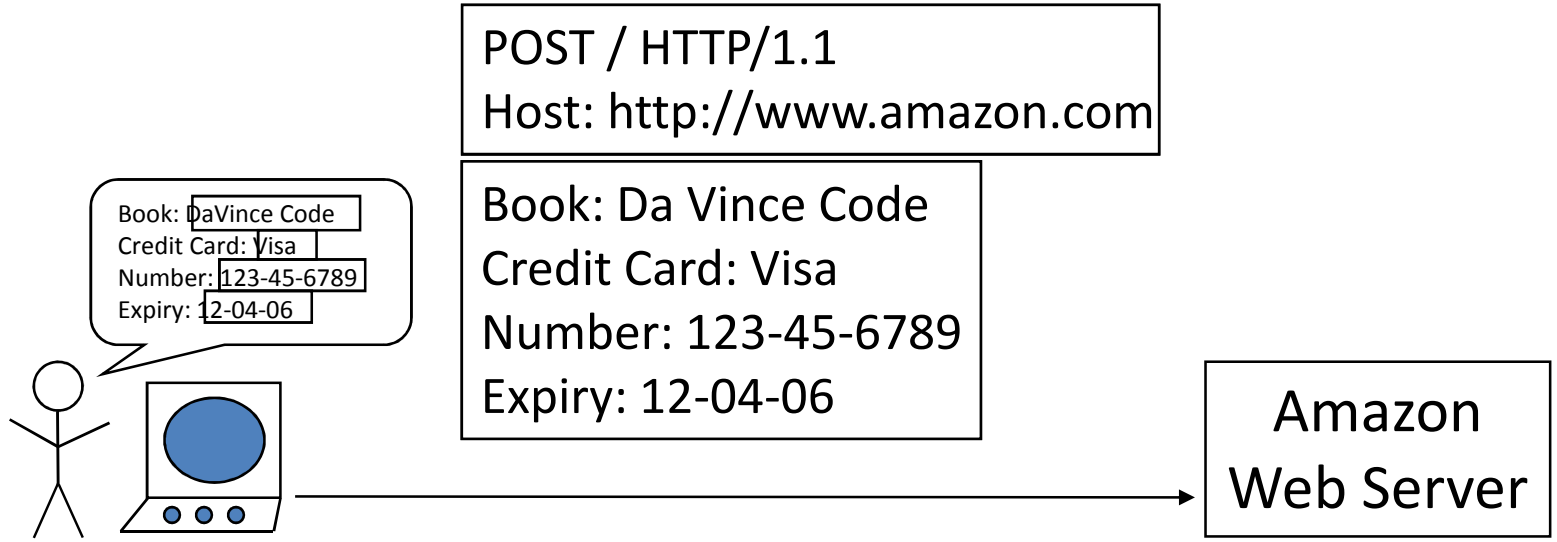
- Web Basics
  - HTTP Methods: GET, POST, PUT, DELETE
  - Idempotent / Non- Idempotent HTTP Methods
- REST Architecture
  - REST Services with Example
- SOAP Vs REST Services
  - Examples
  - Comparisons: Proxy Server, Intermediaries, Caching, State Transition, Generic Interface
- REST Best Practices
- Richardson Maturity Model
- JAX-RS
- Implementation of REST Web Service in NetBeans using Jersey

Web Basics: Retrieving Information using HTTP GET

- The user types in at his browser: http://www.amazon.com
- The browser software creates an HTTP header (no payload)
  - The HTTP header identifies:
    - The desired action: GET ("get me resource")
    - The target machine (www.amazon.com)

Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Web Basics: Updating Information using HTTP POST



- The user fills in the Web page's form
- The browser software creates an HTTP header with a payload comprised of the form data
    - The HTTP header identifies:
        - The desired action: POST ("here's some update info")
        - The target machine (amazon.com)
    - The payload contains:
        - The data being POSTed (the form data)

Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Terminology: Header and Payload

Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Web Basics: Simple Set of Operations, via the HTTP API

- HTTP provides a simple set of operations. Amazingly, all Web exchanges are done using this simple HTTP API:
  - GET = "give me some info" (Retrieve)
  - POST = "here's some new info" (Create)
  - PUT = "here's some update info" (Update)
  - DELETE = "delete some info" (Delete)
- The HTTP API is CRUD (Create, Retrieve, Update, and Delete)

Classification of HTTP Methods

- Classification of common HTTP Methods (GET, PUT, POST, DELETE)
  - GET is read-only method
  - PUT, POST and DELETE are write methods
- As GET is read-only method, no matter how many times you do GET request, nothing changes on Server side.
  - It is safe to make duplicate GET requests
- Since PUT, POST and DELETE are used to write to server, you can not call these methods multiple times.

Classification of HTTP Methods

- Repeatable and Non-repeatable operations
  - GET is repeatable
  - **What about POST, DELETE and PUT?**
    - PUT and DELETE are repeatable
      - Multiple delete calls to the same resource won't have unwanted side effects
      - PUT call (updation of information) won't have side effects
    - POST is not repeatable
      - Multiple POST requests have side effect: create duplicates data (creates a new resource) on server.
- Idempotent HTTP Methods :
  - Safely Repeatable HTTP Methods: GET, PUT, DELETE
- Non-Idempotent HTTP Method:
  - Can not be repeated safely: POST

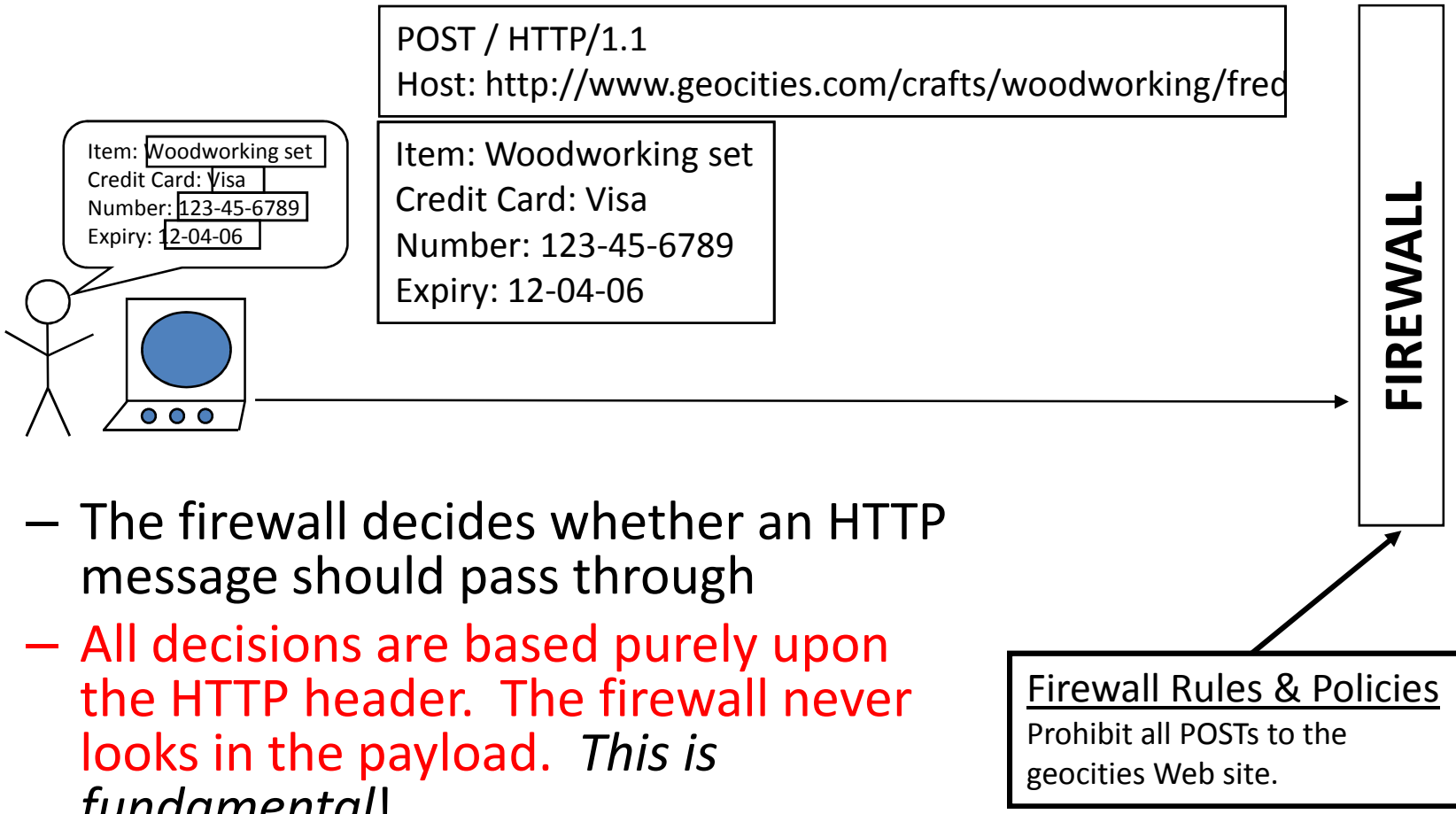
Idempotence is the property of operations in computer science that can be applied multiple times without changing the result beyond the initial application.

Classification of HTTP Methods

- HTTP specifications require GET, PUT & DELETE methods to always be idempotent.
  - If clients make request with one of these methods, they don't have to worry on making duplicate requests. But if they are making POST requests, they can't safely make duplicate requests.
  - The resource creation should always be via POST method because resource creation is non-idempotent operation.
- Every browser has "REFRESH" button
  - Resubmits the last HTTP request made by browser
  - If the last request is idempotent request (For ex. GET), the browser goes ahead and makes request
  - But if it was POST request and you press "REFRESH" button, browser warns you (You have already submitted data before. Are you sure to resubmit?).

Web Component: Firewalls

(Firewall: "Should I allow POSTing to geocities?")



- The firewall decides whether an HTTP message should pass through
- All decisions are based purely upon the HTTP header. The firewall never looks in the payload. *This is fundamental*!
- This message is rejected!

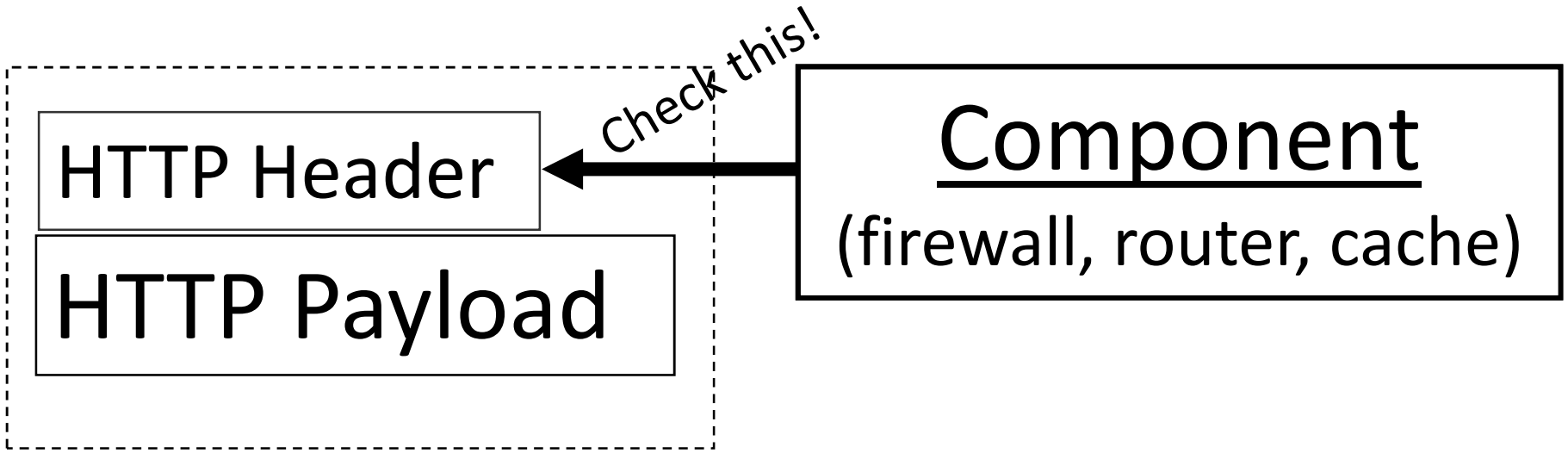
Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Basic Web Components: Firewalls, Routers, Caches

- The Web is comprised of some basic components:
  - **Firewalls**: these components decide what HTTP messages get out, and what get in.
    - These components enforce **Web security**.
  - **Routers**: these components decide where to send HTTP messages.
    - These components manage **Web scaling**.
  - **Caches**: these components decide if a saved copy can be used.
    - These components increase **Web speed**.

- All these components base their decisions and actions purely upon information in the HTTP header.

Web Components Operate using only Information found in the HTTP Header!



Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Letter Analogy

- Why do Web components base their decisions solely on information in the HTTP header?
- My company has a receiving warehouse. All letters and packages go there first, and from there they are distributed.
- No one in the receiving warehouse may look inside any letter or package. All decisions about what to do with letters and packages must be made purely by looking at the addressing on the outside. Any attempt to peek inside of letters and packages is a violation of law.

REST URI

- For Web Application URIs do not matter. The user remembers the URI of home page and navigate through different pages using links
  - Get an account details of id 1234
  - /getAccount.do? id=1234
- If you are writing REST API, the consumers have to be aware of URI (The consumers are developers who has to write code to make HTTP call to URIs).
- Imagine Static Website
  - Site
    - Contents
      - Index.html
      - Site-map.html
  - Every page has a unique URI (website.com/site/index.html).
  - In REST world, every entity has a unique URI. Drop the word ".html" (website.com/site/index)
  - Think of resource and create unique URI for it.

REST URI

- REST APIs use Uniform Resource Identifiers (URIs) to address resources.
- RESTful URI should refer to a resource that is a thing (noun) instead of referring to an action (verb).
- A resource can be a singleton or a collection.
  - For ex., “customers” is a collection resource and “customer” is a singleton resource (in a banking domain).
  - We can identify “customers” collection resource using the URI /customers”.
  - We can identify a single “customer” resource using the URI “/customers/{customerId}”.
- A resource may contain sub-collection resources also. For example, sub-collection resource “accounts” of a particular “customer” can be identified using the URI“/customers/{customerId}/accounts”.
- Similarly, a singleton resource “account” inside the sub-collection resource “accounts” can be identified as follows: “/customers/{customerId}/accounts/{accountId}”

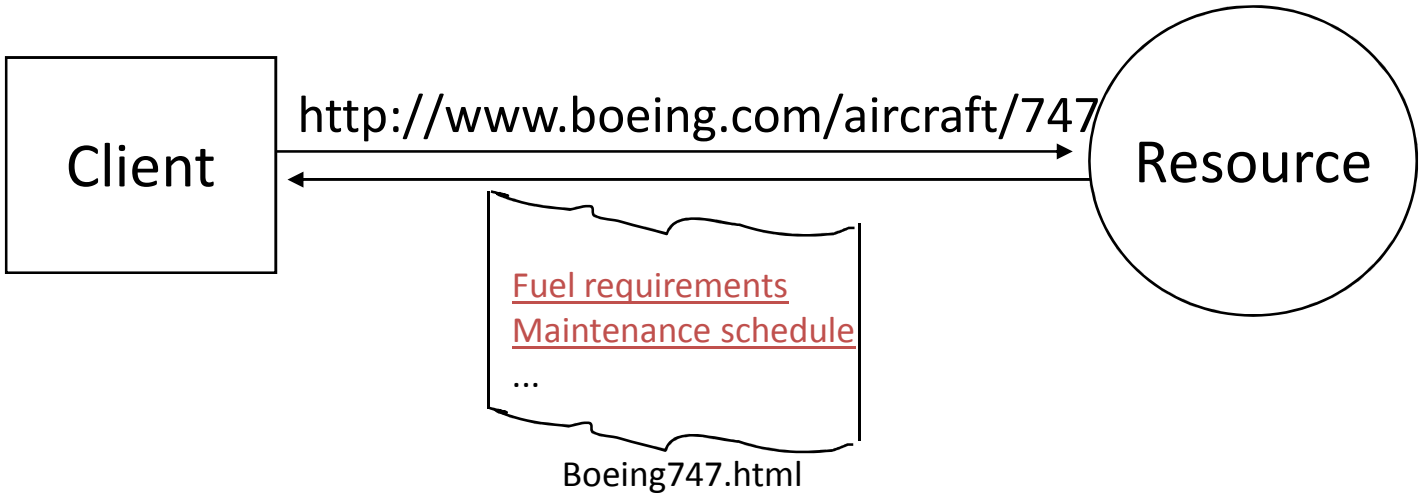
REST

What is REST?

" REST " was coined by Roy Fielding
in his Ph.D. dissertation [1] to describe a design pattern for implementing networked systems.

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Why is it called "Representational State Transfer? "



The Client references a Web resource using a URL.
A **representation** of the resource is returned (in this case as an HTML document).
The representation (*e.g.,* Boeing747.html) places the client in a new **state**.
When the client selects a hyperlink in Boeing747.html, it accesses another resource.
The new representation places the client application into yet another state.
Thus, the client application **transfers** state with each resource representation.

Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Representational State Transfer

"Representational 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."

- Roy Fielding

Motivation for REST

The motivation for developing REST was to create a design pattern for how the Web should work, such that it could serve as the guiding framework for the Web standards and designing Web services.

REST - Not a Standard

- REST is not a standard
  - You will not see the W3C putting out a REST specification.
  - You will not see IBM or Microsoft or Sun selling a REST developer's toolkit.
- REST is just a **design pattern**
  - You can't bottle up a pattern.
  - You can only understand it and design your Web services to it.
- It is an approach for creating Web Services.
- REST does prescribe the **use** of standards:
  - HTTP
  - URL
  - XML/HTML/GIF/JPEG/*etc.* **(Resource Representations)**
  - text/xml, text/html, image/gif, image/jpeg, *etc.* (Resource Types, MIME Types)
- To understand the REST design pattern, let's look at an example (learn by example).

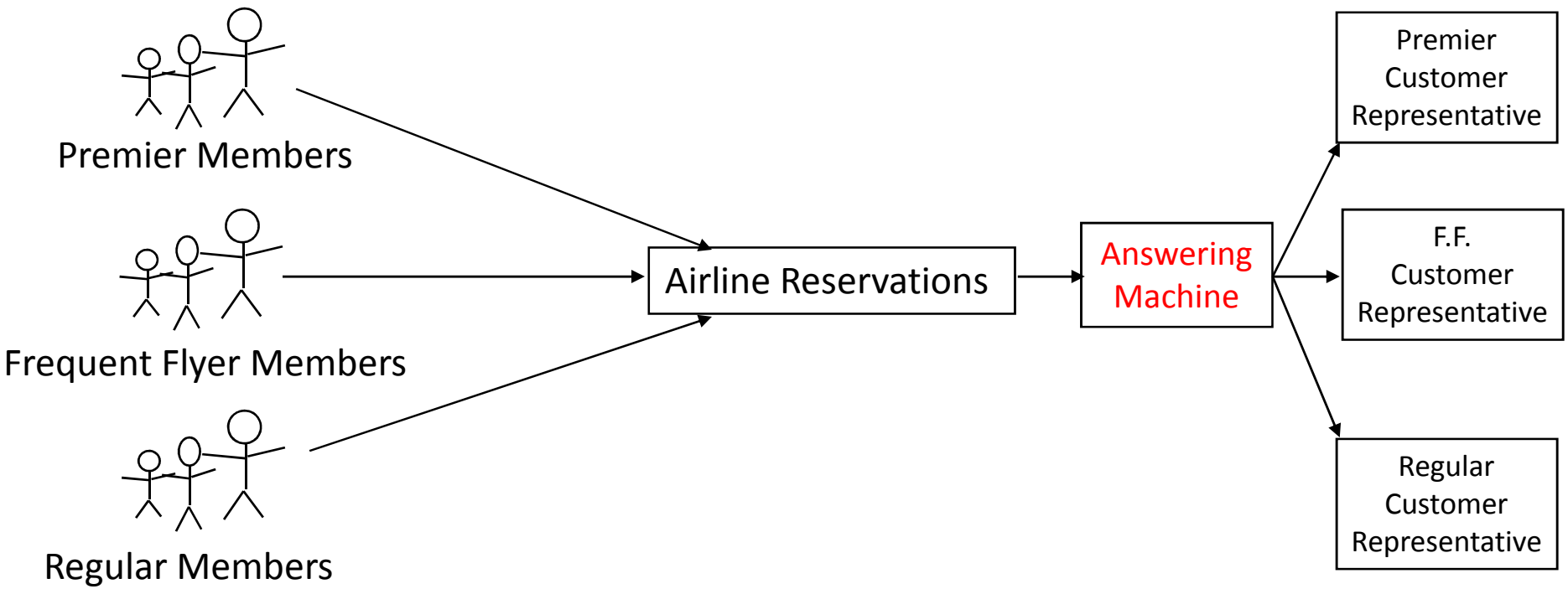
Example: Airline Reservation Service

- Suppose that an airline wants to create a telephone reservation system for customers to call in and make flight reservations.
- The airline wants to ensure that its premier members get immediate service, its frequent flyer members get expedited service and all others get regular service.
- There are two main approaches to implementing the reservation service...

Approach 1: "Press 1 for Premier, Press 2 for..."

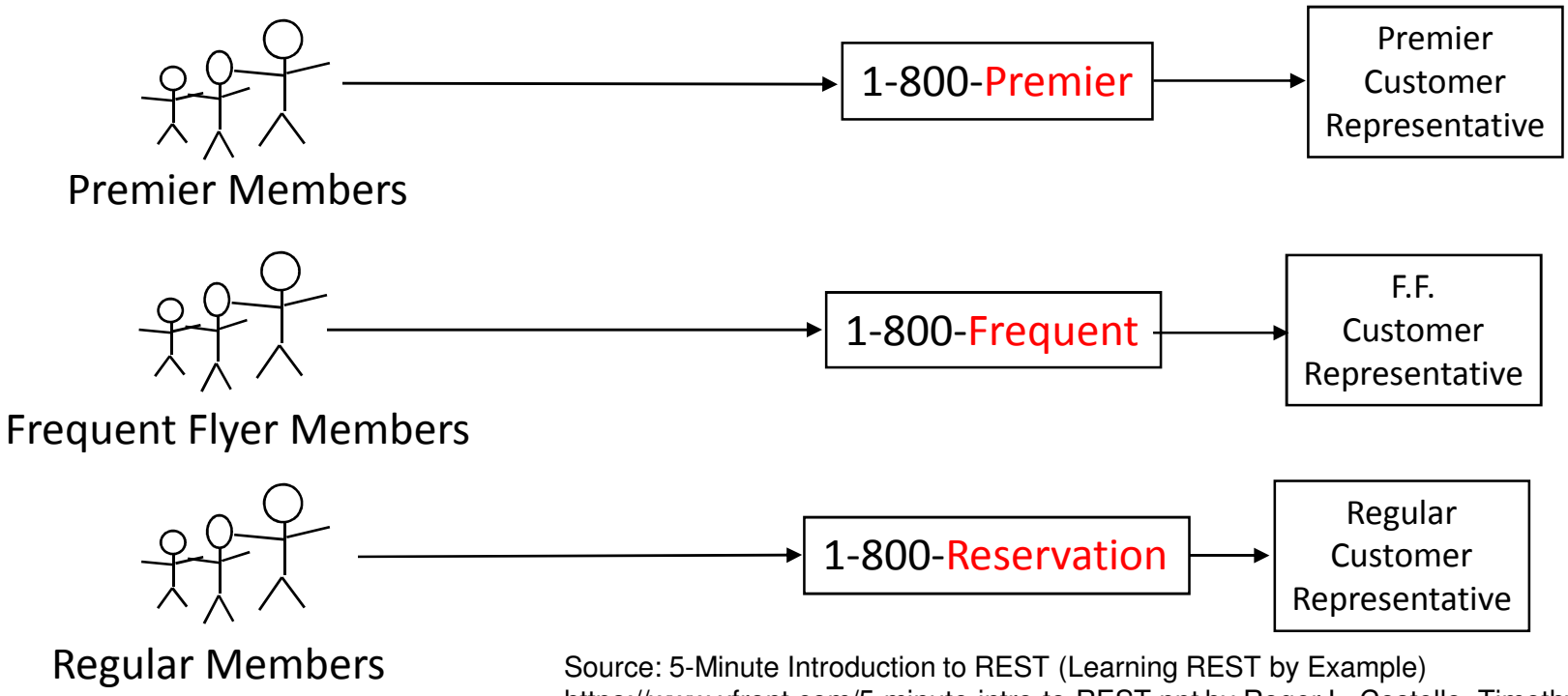
The airline provides a single telephone number.

Upon entry into the system a customer encounters an automated message, "Press 1 if you are a premier member, press 2 if you are a frequent flyer, press 3 for all others."



Approach 2 : Telephone Numbers are Cheap! Use Them!

The airline provides several telephone numbers - one number for premier members, a different number for frequent flyers, and still another for regular customers.



Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

Discussion

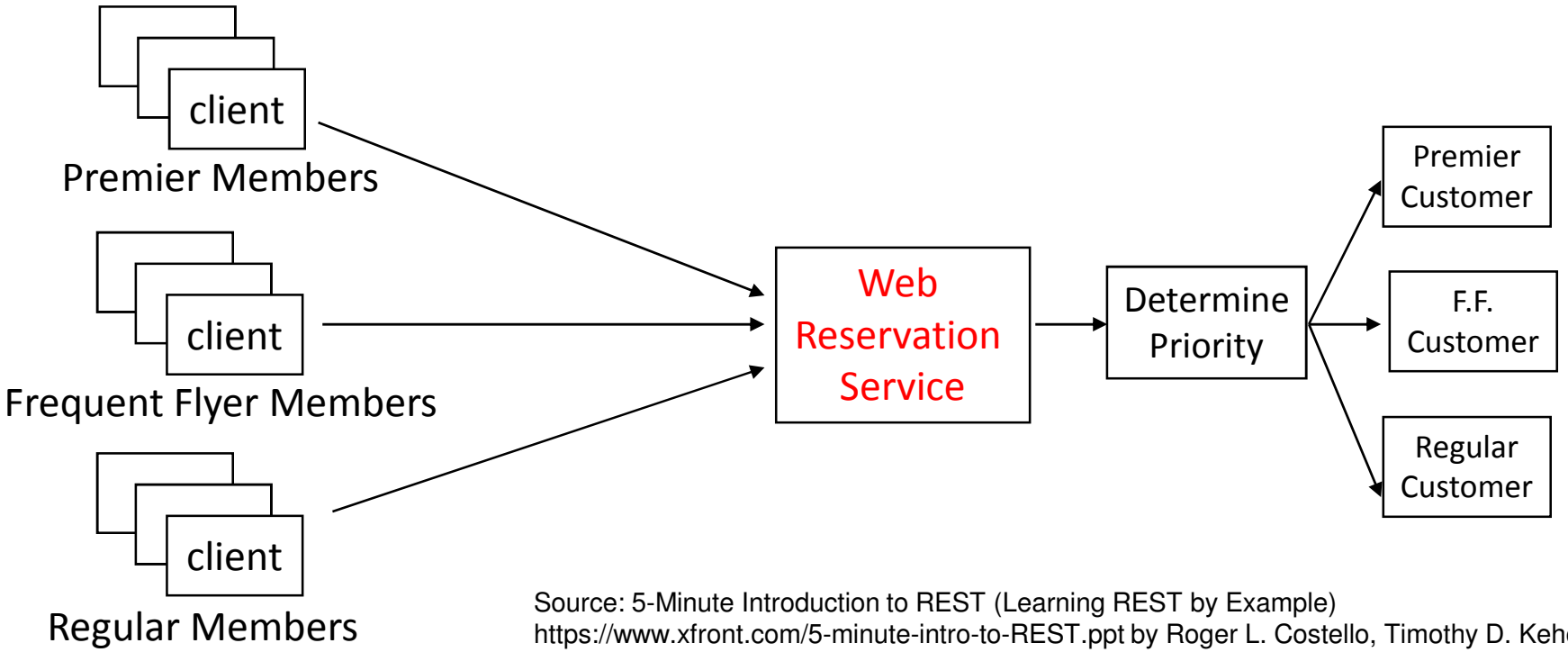
- In Approach 1 the answering machine introduces an extra delay, which is particularly annoying to premier members. (Doesn't everyone hate those answering systems)
- With Approach 2 there is no intermediate step. Premier members get instant pickup from a customer service representative. Others may have to wait for an operator.

Web-Based Reservation Service

- Suppose now the airline wants to provide a Web reservation service for customers to make flight reservations through the Web.
- Just as with the telephone service, the airline wants to ensure that its premier members get immediate service, its frequent flyer members get expedited service, all others get regular service.
- There are two main approaches to implementing the Web reservation service. The approaches are analogous to the telephone service...

Approach 1: One-Stop Shopping

The airline provides a single URL. The Web service is responsible for examining incoming client requests to determine their priority and process them accordingly.



Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

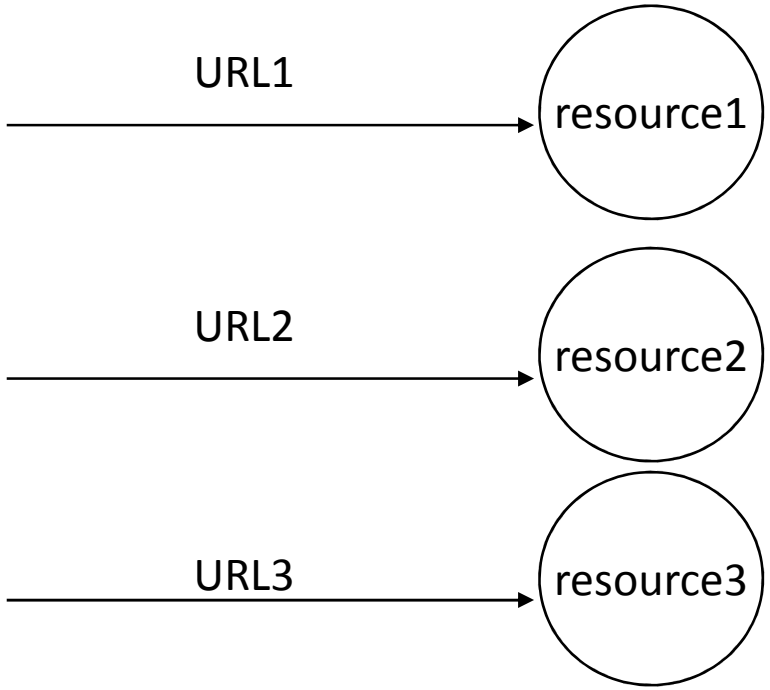
Approach 1 Disadvantages

- There is currently no industry accepted practice (rules) for expressing priorities, so rules would need to be made. The clients must learn the rule, and the Web service application must be written to understand the rule.
- This approach is based upon the incorrect assumption that a URL is "expensive" and that their use must be rationed.
- The Web service is a central point of failure. It is a bottleneck. Load balancing is a challenge.
- It violates Tim Berners-Lee Web Design, Axiom 0.

Web Design, Axiom 0

(Tim Berners-Lee, director of W3C)

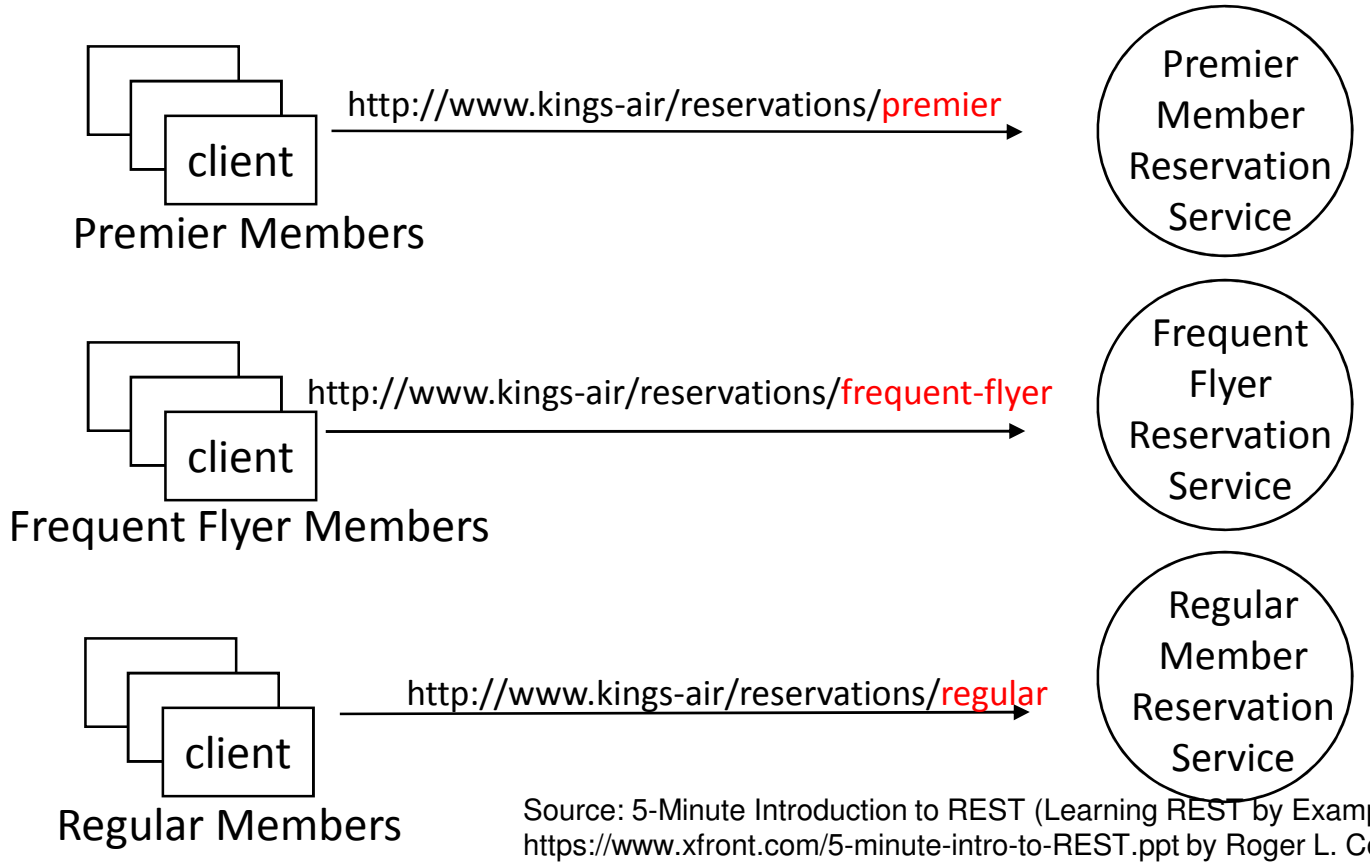
- Axiom 0: all resources on the Web must be uniquely identified with a URI.



Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

Approach 2: URLs are Cheap! Use Them!

The airline provides several URLs - one URL for premier members, a different URL for frequent flyers, and still another for regular customers.

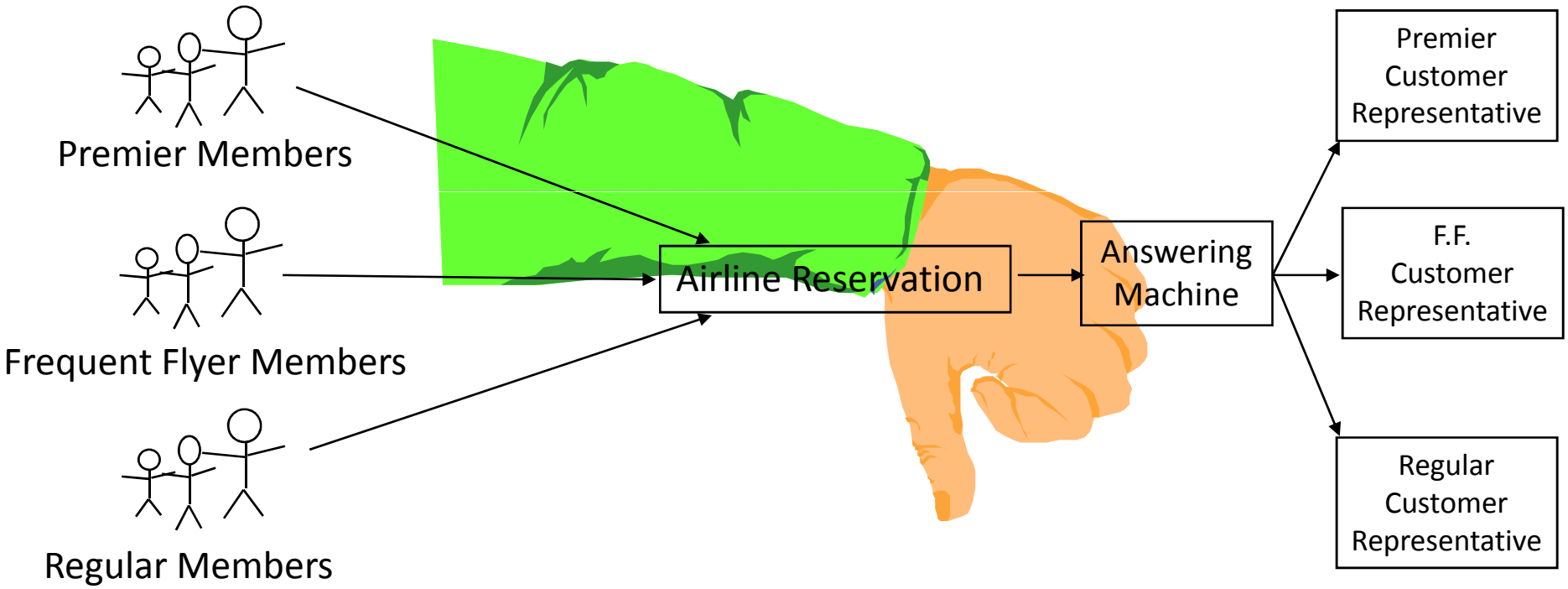


Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

Approach 2 Advantages

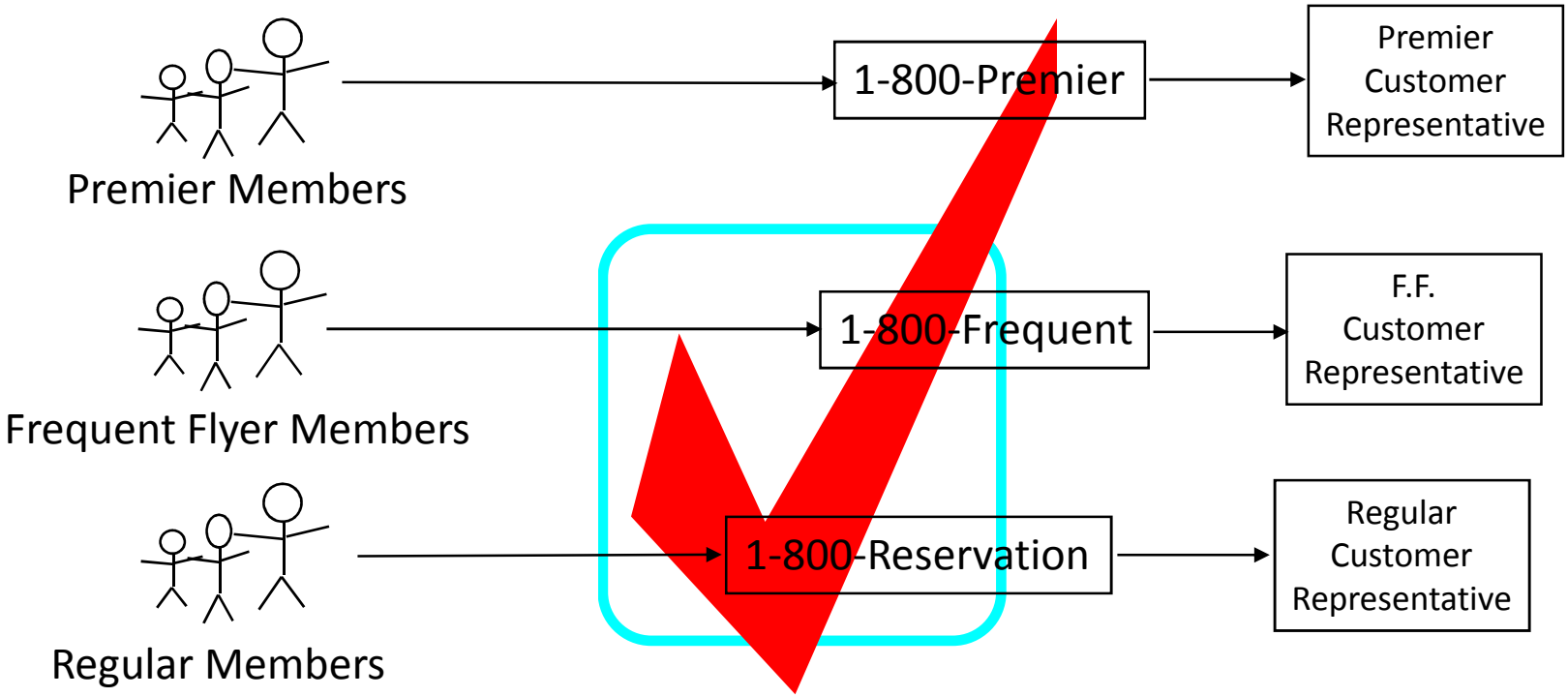
- The different URLs are discoverable by search engines and UDDI registries.
- It's easy to understand what each service does simply by examining the URL.
- There is no need to introduce rules. Priorities are elevated to the level of a URL. "What you see is what you get."
- It's easy to implement high priority - simply assign a fast machine at the premier member URL.
- There is no bottleneck. There is no central point of failure.
- Consistent with Axiom 0.

This is not the REST Design Pattern



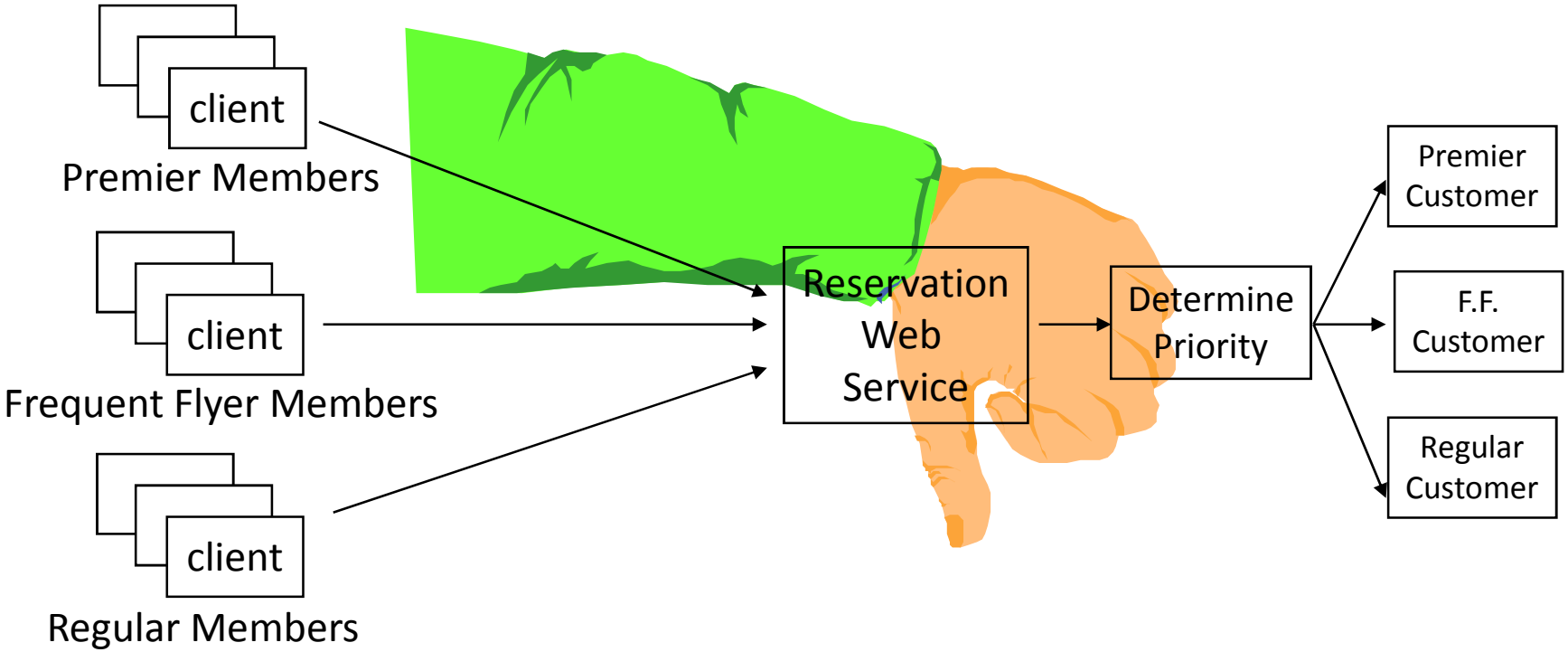
Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

This is the REST Design Pattern



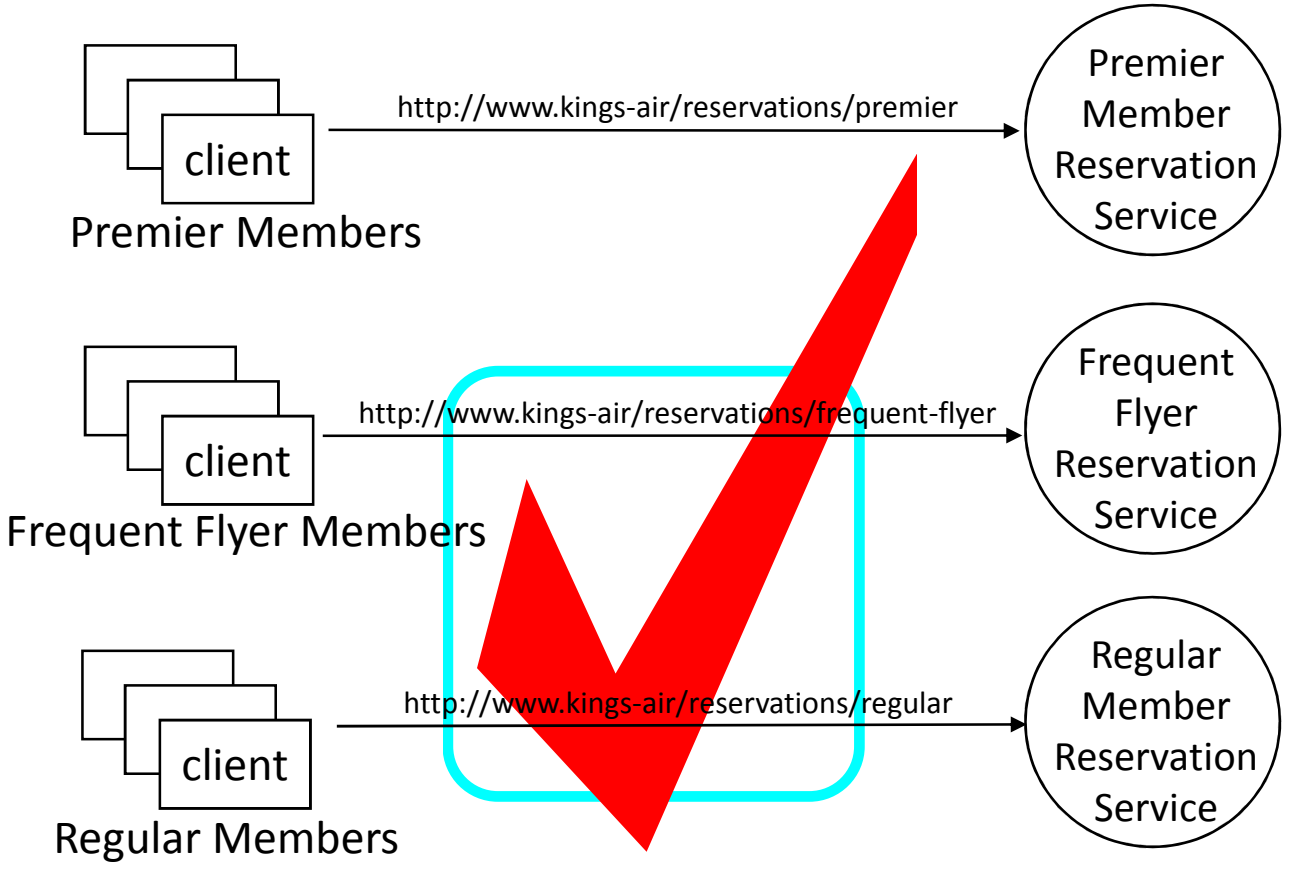
Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

This is not the REST Design Pattern



Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

This is the REST Design Pattern

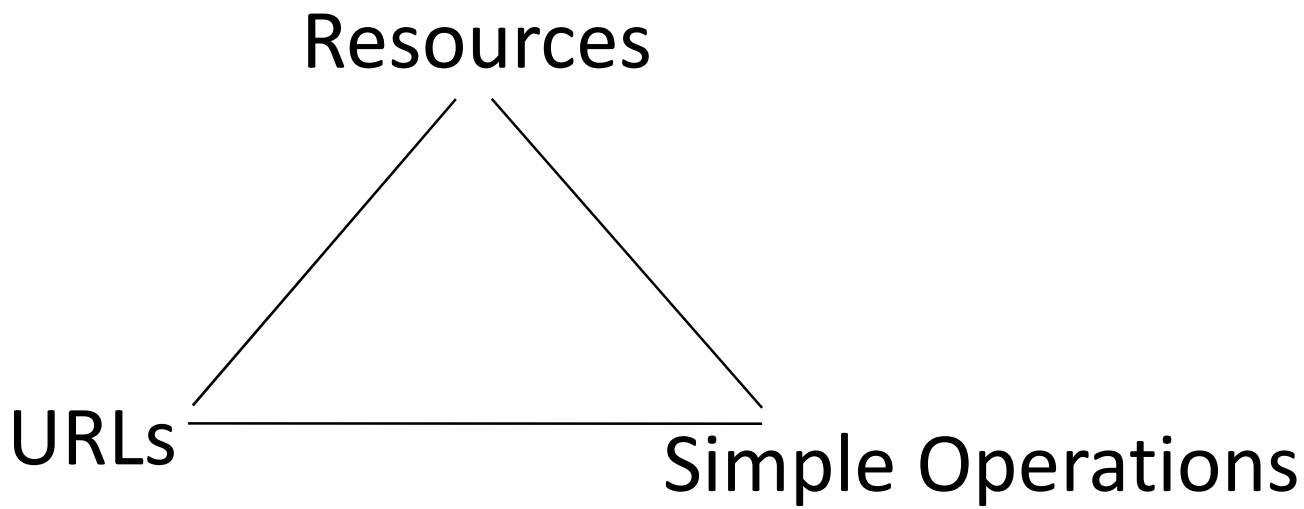


Source: 5-Minute Introduction to REST (Learning REST by Example)
https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe

Two Fundamental Aspects of the REST Design Pattern

- **Resources**
  Every distinguishable entity is a resource. A resource may be a Web site, an HTML page, an XML document, a Web service, a physical device, *etc.*
- **URLs Identify Resources**
  Every resource is uniquely identified by a URL. This is Tim Berners-Lee Web Design, Axiom 0.

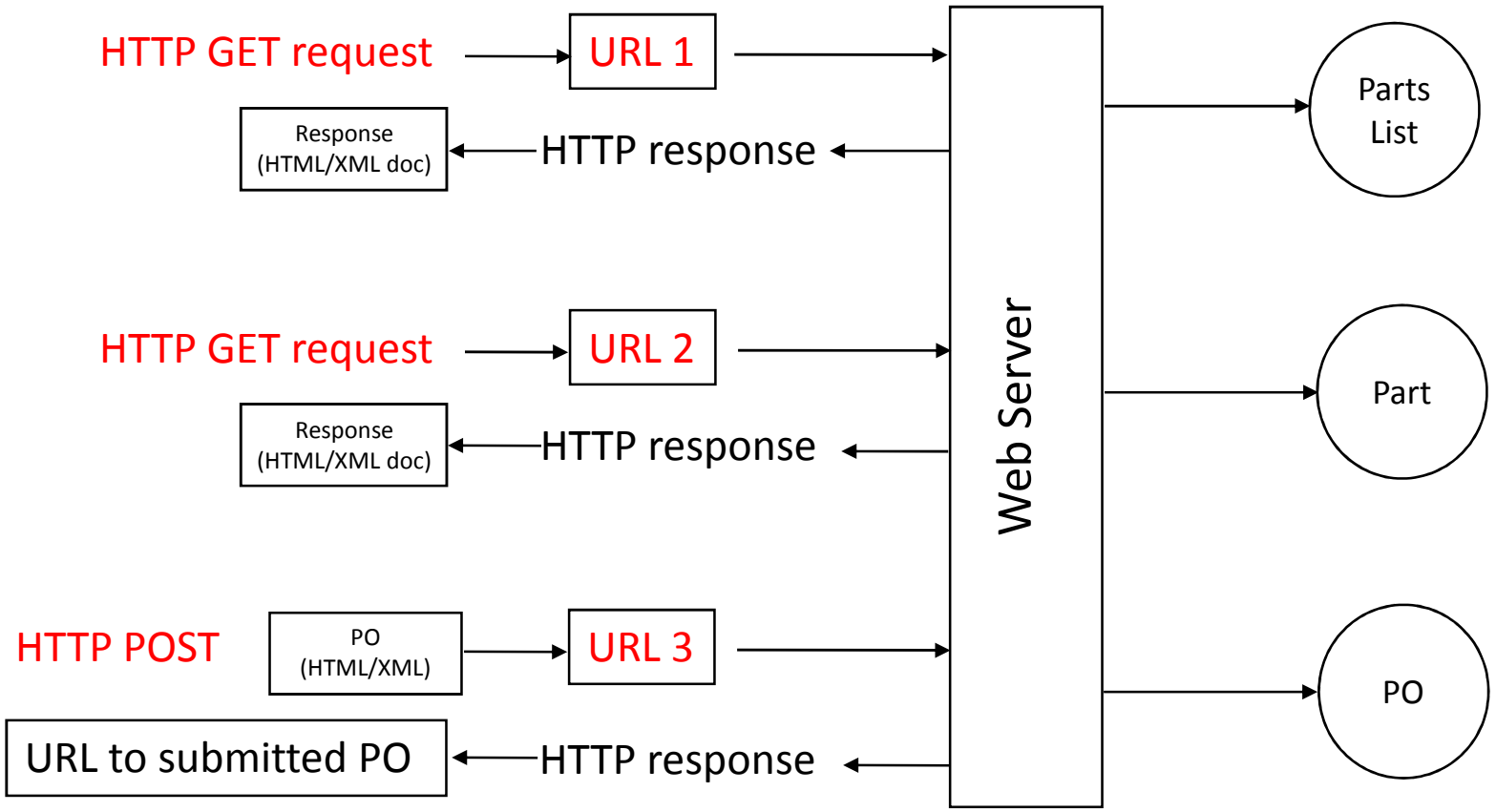
The Three Fundamental Aspects of the REST Design Pattern



Parts Depot Web Services

- Parts Depot, Inc has deployed some web services to enable its customers to:
  - get a list of parts
  - get detailed information about a particular part
  - submit a Purchase Order (PO)

The REST way of Implementing the Web Services



Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

The REST way of Implementing the Web Service

- Service: Get a list of parts
  - The web service makes available a URL to a parts list resource. Example, a client would use this URL to get the parts list:
    - http://www.parts-depot.com/parts
    - Note that **how** the web service generates the parts list is completely transparent to the client. This is *loose coupling*.
  - The web service may wish to allow the client to specify whether he/she wants the parts list as an HTML document, or as an XML document. This is how to specify that an XML document is desired:
    - http://www.parts-depot.com/parts?flavor=xml

Data Returned - Parts List

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.parts-depot.com
                   http://www.parts-depot.com/parts.xsd">
   <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>
```

Note that the parts list has links to get detailed info about each part. This is a key feature of REST. The client transfers from one state to the next by examining and choosing from among the alternative URLs in the response document.

Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

The REST way of Implementing the Web Service

- Service: Get detailed information about a particular part
  - The web service makes available a URL to each part resource. Example, here's how a client requests a specific part:
    - http://www.parts-depot.com/parts/00345?flavor=xml

Source: An Expanded Introduction to REST
https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe

Data Returned - Part

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
                "http://www.parts-depot.com
                 http://www.parts-depot.com/part.xsd">
    <Part-ID>00345</Part-ID>
    <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</p:Part>
```

Again observe how this data is linked to still more data - the specification for this part may be found by traversing the hyperlink. Each response document allows the client to drill down to get more detailed information.

HATEOAS

- HATEOAS: Hypermedia As The Engine Of Application State

- Hyper Text (Media): Text that has links to other Text. These links are called Hyper links and are used to navigate through any website (from page to page).
- What if we implement the same concept in RESTful Web service.

- The value of WEB is its linked-ness. The same is true for Programmatic Web
  - Use links to describe state transitions in Programmatic Web Services. By navigating resources you change application state. Links lead to other resources which also have links.

HATEOAS Example

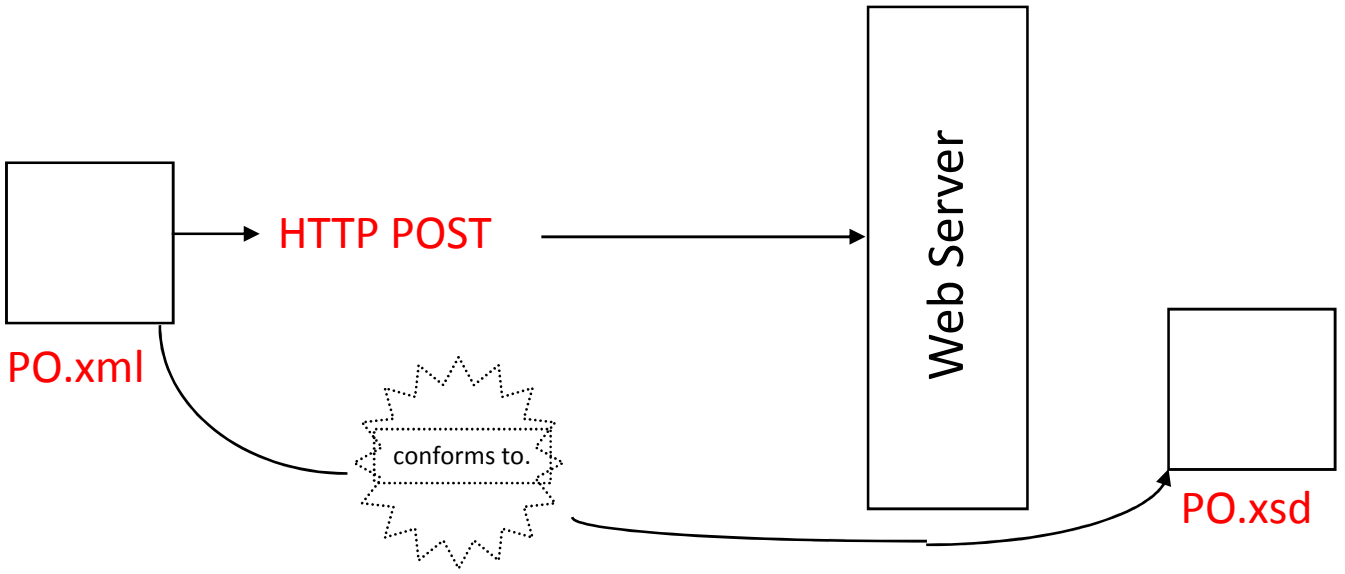
```
{
   "productId": "123",
   "productName": "Super Microwave",
   "description": "The best microwave in the world. Fact."
   "links": [{
      "rel": "self",
      "href": "http://localhost:8080/super-shop/api/products/123"
   }, {
      "rel": "details",
      "href": "http://localhost:8080/super-shop/api/products/123/details"
   }, {
      "rel": "addToCart",
      "href": "http://localhost:8080/super-shop/api/addToCart/123"
   }]
}
```

If you get a JSON response for a *product* from online shopping website, and if it was using HATEOAS it could look something like that

rel – stands for 'relationship' and explains how the link relates to the object that you requested for

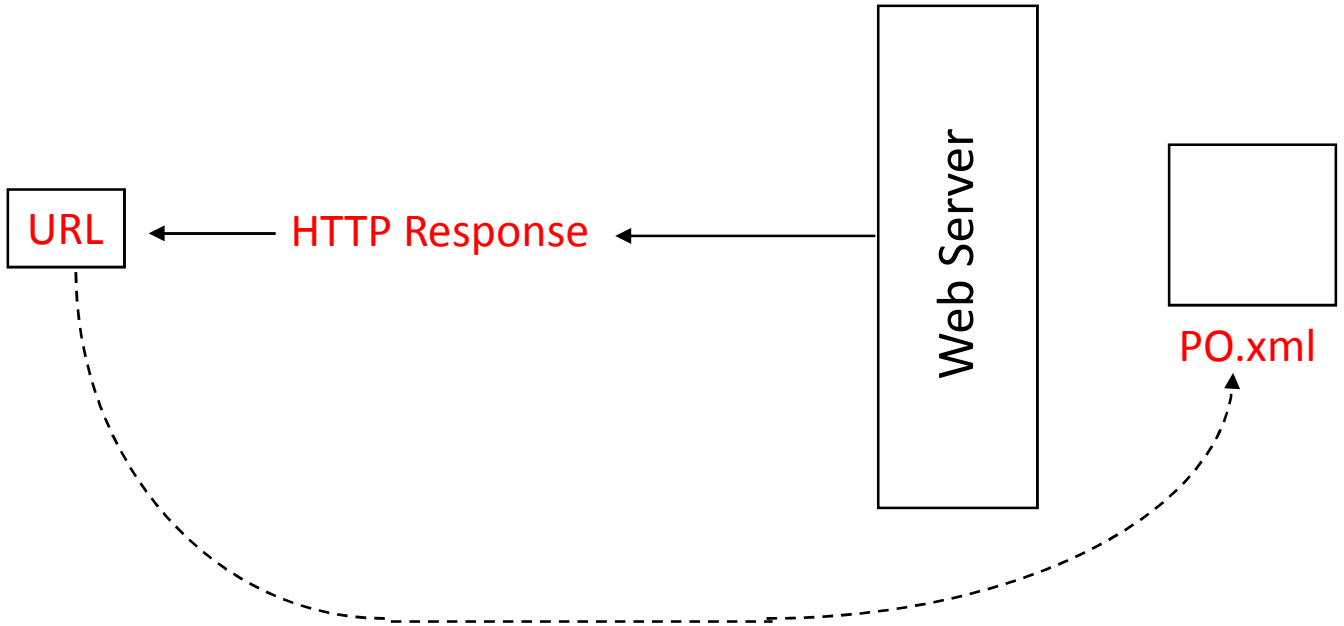
The REST way of Implementing the Web Service

- Service: Submit a Purchase Order (PO)
  - The web service makes available a URL to submit a PO. The client creates a PO XML instance document which conforms to the PO schema that Parts Depot has designed (and publicized in a WSDL document). The client submits PO.xml as the payload of an HTTP POST.



Submit PO Service (cont.)

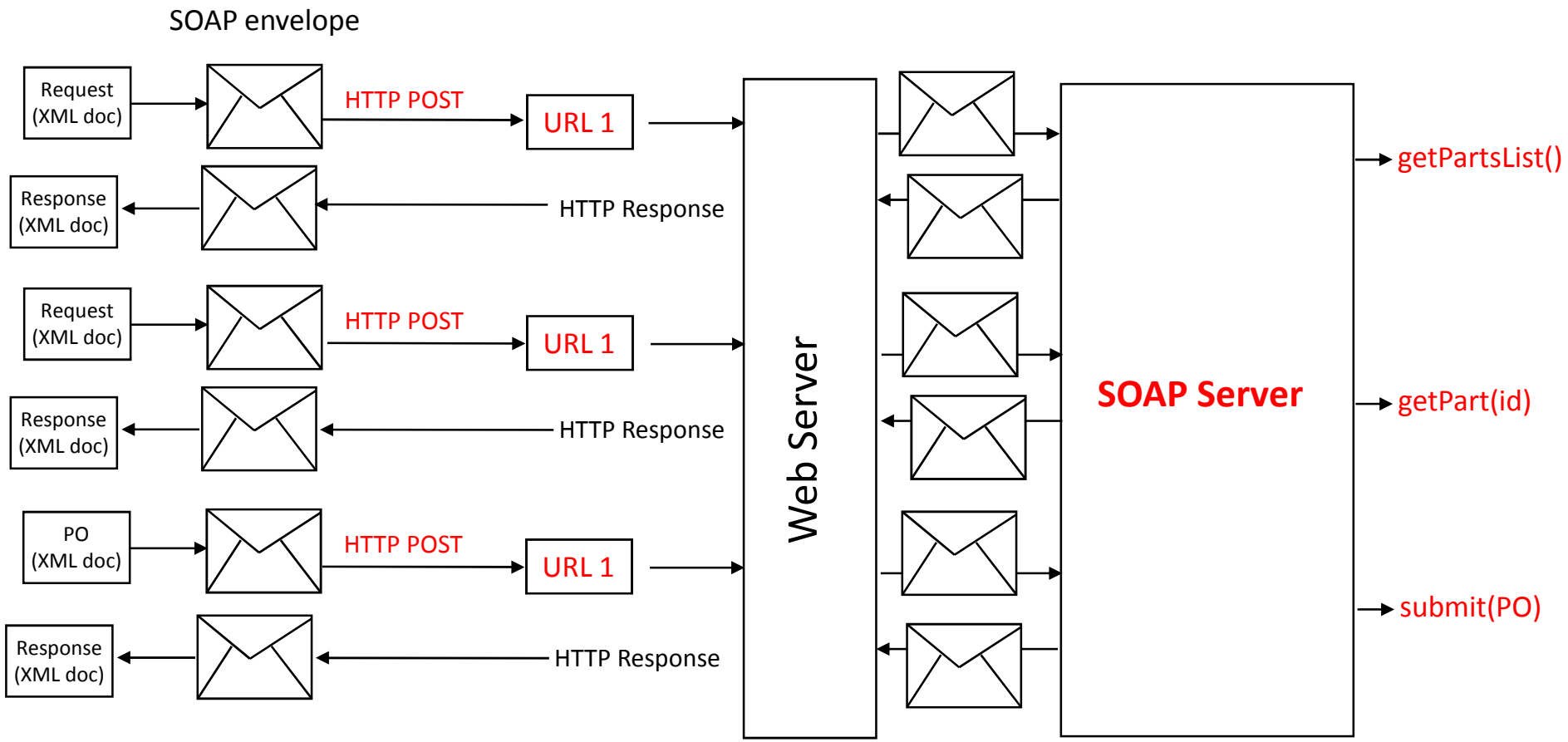
- The PO service responds to the HTTP POST with a URL (as PO.xml is a new resource available on server) to the submitted PO. Thus, the client can retrieve the PO any time thereafter.
  - The PO has become a piece of information which is shared between the client and the server. The shared information (PO) is given an address (URL) by the server and is exposed as a Web service.



Characteristics of a REST-based Network

- Stateless: each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- Cache: to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- Uniform interface: all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).
- Named resources: the system is comprised of *resources* which are *named* using a URI.
- Interconnected resource representations: the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.

Implementing the Web Services using SOAP

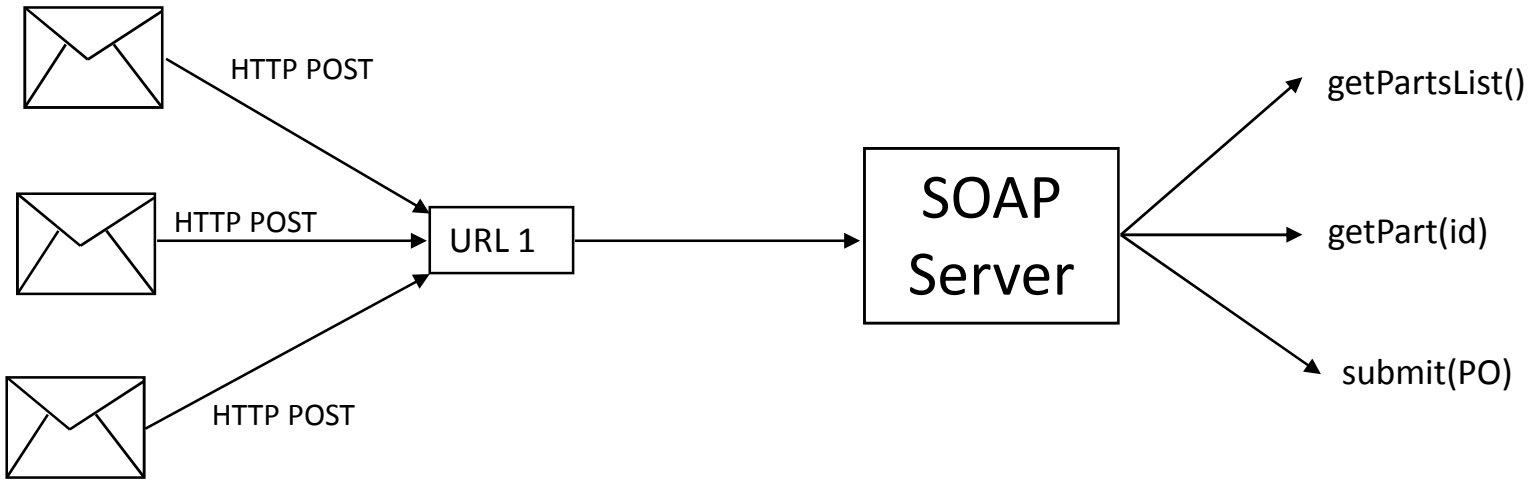


Note the use of the same URL (URL 1) for all transactions. **The SOAP Server parses the SOAP message to determine which method to invoke. All SOAP messages are sent using an HTTP POST.**

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

Note about the SOAP URI

It is not a SOAP requirement all messages be funneled to the same URL:



However, it is common among SOAP vendors to follow this practice.
For example, here is the URL for all requests when using Apache SOAP:

[host]/soap/servlet/messagerouter

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

Implementing the Web Service using SOAP

- Service: Get a list of parts
  - The client creates a SOAP document that specifies the procedure desired.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:getPartsList xmlns:p="http://www.parts-depot.com"/>

  </soap:Body>
</soap:Envelope>
```

Then the client will HTTP POST this document to the SOAP server at:
http://www.parts-depot.com/soap/servlet/messagerouter
The SOAP server takes a quick look into this document to determine what procedure to invoke.

Data Returned - Parts List

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:getPartsListResponse xmlns:p="http://www.parts-depot.com">
      <Parts>
          <Part-ID>00345<Part-ID>
          <Part-ID>00346<Part-ID>
          <Part-ID>00347<Part-ID>
          <Part-ID>00348<Part-ID>
      </Parts>
    <p:getPartsListResponse>

  </soap:Body>
</soap:Envelope>
```

Note the absence of links. Why is this? A URL that points to a SOAP service is meaningless since the URL to a SOAP service is just to the SOAP server. Thus, the URL would need to be supplemented with some indication of which method to invoke at that URL.
[Note: of course this response could contain a URL to a REST-ful service.]

Implementing the Web Service using SOAP

- Service: Get detailed information about a particular part
  - The client creates a SOAP document that specifies the procedure desired, along with the part-id parameter.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:getPart xmlns:p="http://www.parts-depot.com">
        <part-id>00345</part-id>
    </p:getPart>

  </soap:Body>
</soap:Envelope>
```

Again, the client will HTTP POST this document to the SOAP server at:

http://www.parts-depot.com/soap/servlet/messagerouter

Note that this is the same URL as was used when requesting the parts list.

The SOAP server looks into this document to determine what procedure to invoke.

Data Returned - Part

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:getPartResponse xmlns:p="http://www.parts-depot.com">
        <Part-ID>00345</Part-ID>
        <Name>Widget-A</Name>
        <Description>This part is used within the frap assembly</Description>
        <UnitCost currency="USD">0.10</UnitCost>
        <Quantity>10</Quantity>
    </p:getPartResponse>

  </soap:Body>
</soap:Envelope>
```

Again, notice the absence of links. Thus, there is nothing in the response to enable a client to "go to the next level of detail". The information about how to go to the next level of detail must be found out-of-band.

Implementing the Web Service using SOAP

- Service: Submit a Purchase Order (PO)
  - The client creates a SOAP document that contains a PO XML instance document (which conforms to the PO schema that Parts Depot has created)

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:PurchaseOrder xmlns:p="http://www.parts-depot.com">
      ...
    </p:PurchaseOrder>

  </soap:Body>
</soap:Envelope>
```

Once again, the client will HTTP POST this document to the SOAP server at:
http://www.parts-depot.com/soap/servlet/messagerouter
Note that this is the same URL as was used with the other two services.
The SOAP server looks into this document to determine what procedure to invoke.

Data Returned - PO Acknowledgment

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:PO-SubmittalResponse xmlns:p="http://www.parts-depot.com">
        <PO-ID>x-010123fjdsk390f</PO-ID>
    </p:PO-SubmittalResponse>

  </soap:Body>
</soap:Envelope>
```

Again, notice the absence of links.

Contrasting REST and SOAP

- Contrast REST and SOAP in terms of:
  1) Proxy Servers (Web intermediaries)
  2) Transitioning state in a client application
  3) Caching (i.e., performance)
  4) Web evolution (semantic Web)
  5) Generic interface (versus custom interface)
  6) Interoperability
  7) Processing the payload

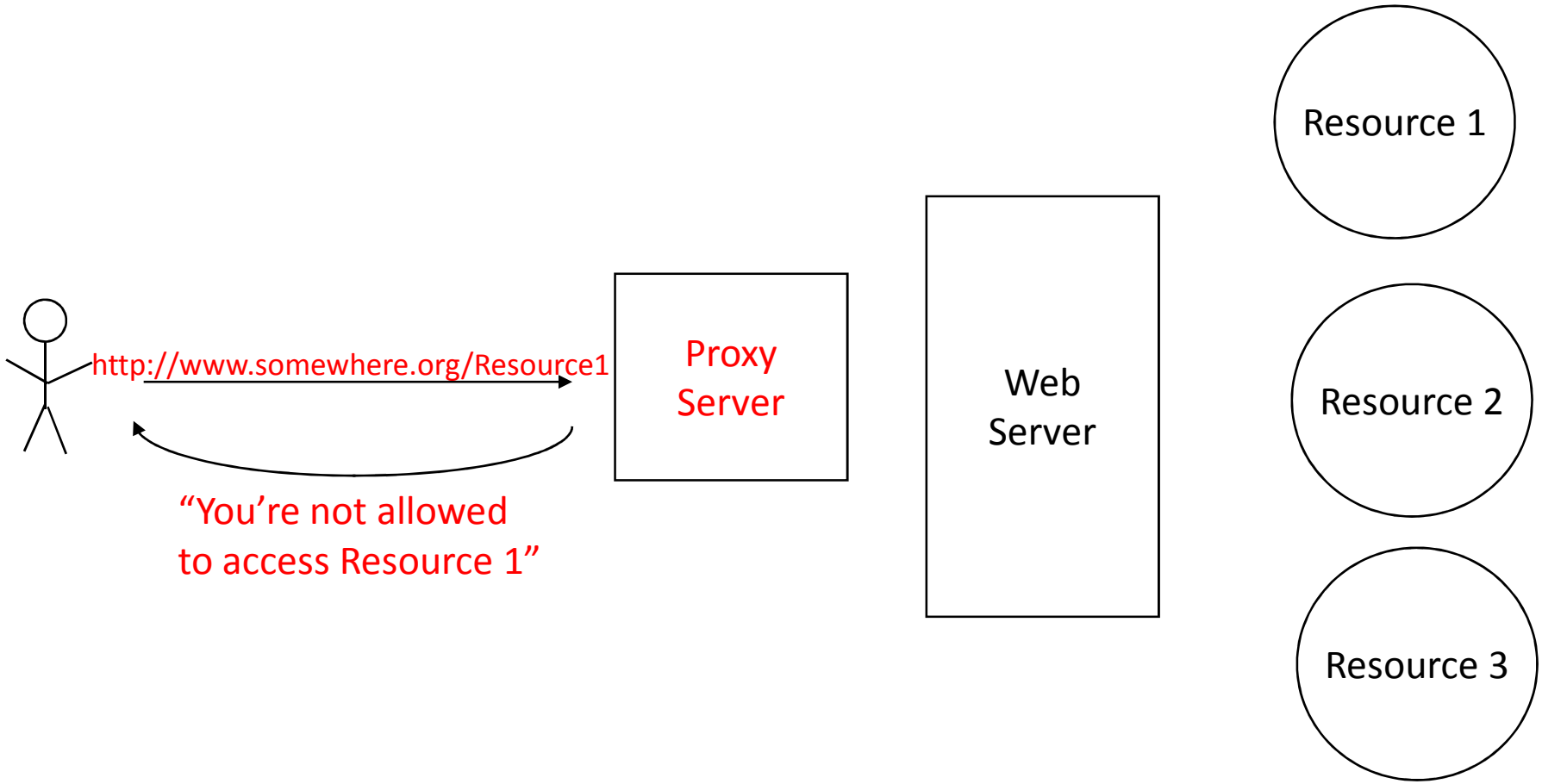
Letter Analogy

- My company has a receiving warehouse. All letters and packages first go there, and from there they are distributed.
- A SOAP Server is analogous to the receiving warehouse - the SOAP Server receives all incoming SOAP messages and then distributes each message to the appropriate application for processing.
- However, there is one big difference:
  - No one in the receiving warehouse is allowed to look inside any letter or package. All decisions about what to do with letters/packages must be made purely by looking at the addressing on the outside. Any attempt to look inside of letters/packages is a violation of Federal Law (U.S.).
  - A SOAP Server, on the other hand, is able to "peek inside" the SOAP envelope. In fact, it must do so because the actual target resource is not specified on the outside, but rather, is hidden within the envelope.
- With **REST all decisions are made based upon the URL and HTTP method.**
- Thus, REST and SOAP have a fundamental difference in this regard.

Proxy Servers

- Consider this scenario:
  - A company has deployed 3 resources - Resource 1, Resource 2, and Resource 3
  - A client wishes to access Resource 1 (get a representation of Resource 1)
  - All client requests go through a proxy server
  - The proxy server enforces the policy that access to Resource 2 and Resource 3 is allowed. However, Resource 1 is off limits (for example, suppose that Resource 1 is Hotmail, and the **client's** company policy prohibits accessing Hotmail using the company's lines).

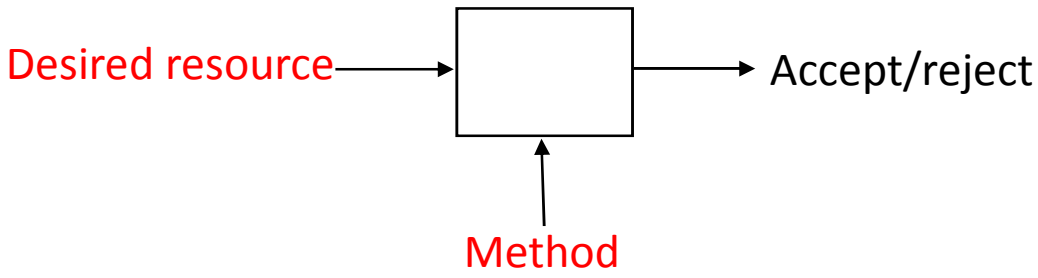
REST and Proxy Servers



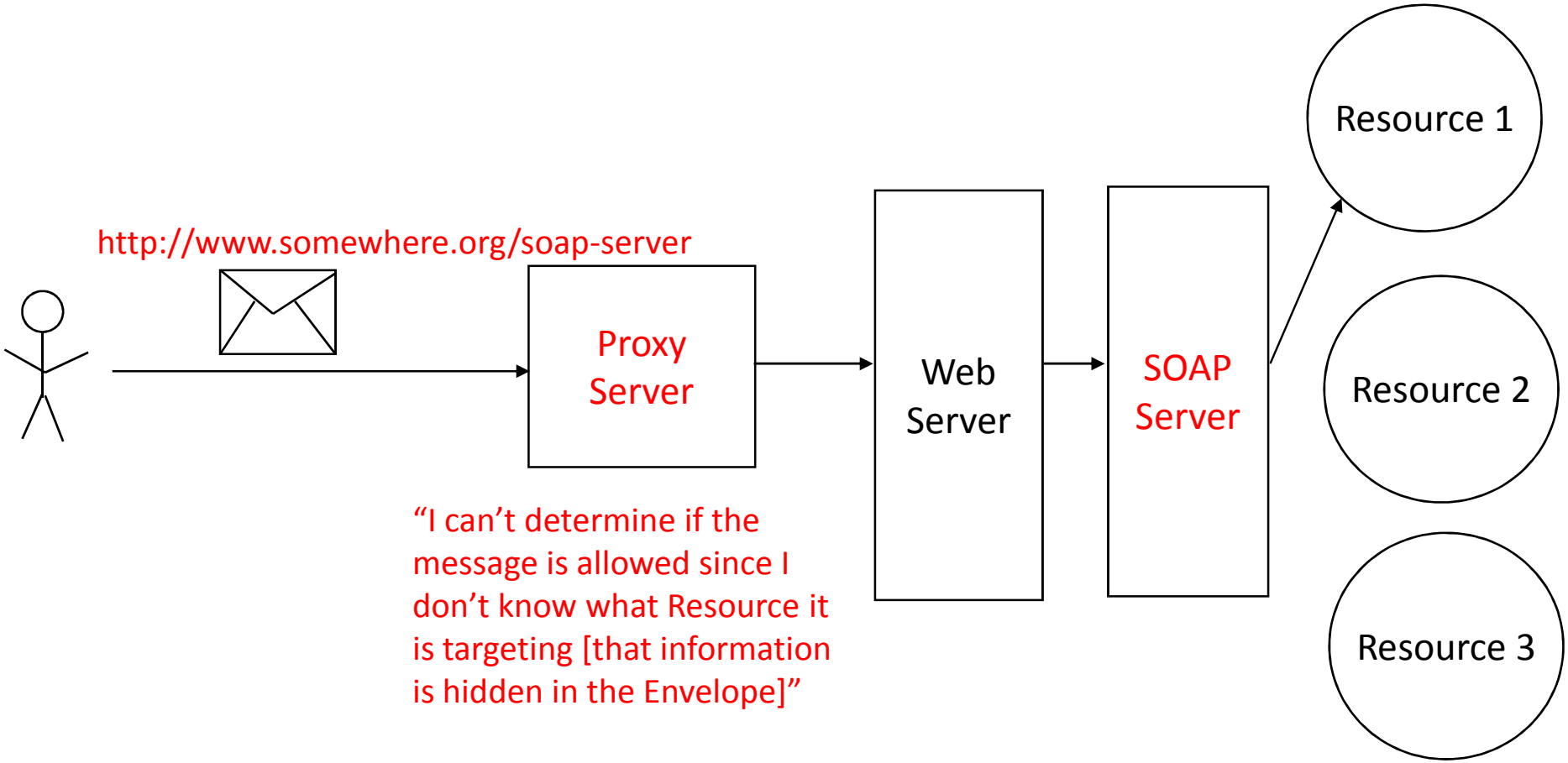
Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

REST and Proxy Servers (cont.)

- The URL identifies the resource that is desired (e.g., Resource 1)
- The HTTP method identifies the desired operation (e.g, HTTP GET)
- A proxy server can decide, based upon the identified resource, and the HTTP method whether or not to allow the operation.



SOAP and Proxy Servers



Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

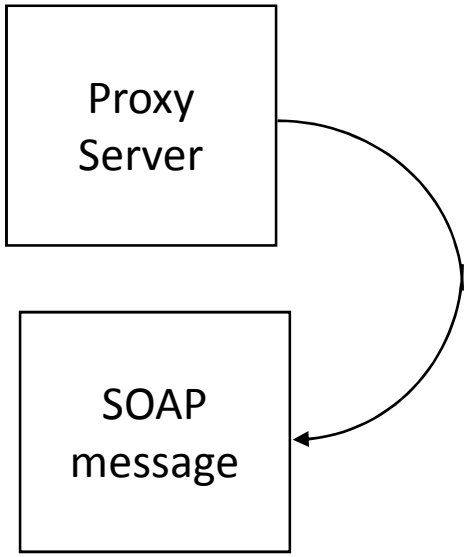
SOAP and Proxy Servers (cont.)

- The URL is not to the target resource, but rather to a SOAP server. This makes it harder, and less likely, for a proxy server to determine which resource is actually being targeted.
- The proxy server would need to look inside the SOAP message to determine which resource is being requested.
  - Further, the proxy server would need to understand the semantics of the message:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getResource xmlns="…">
      <id>R1</id>
    </getResource>
  </soap:Body>
</soap:Envelope>
```

Does this mean that the client is trying to access Resource 1? The proxy server must understand the semantics of every SOAP application!

SOAP and Proxy Servers (cont.)



"What resource is this SOAP message trying to access? What is it trying to do with the resource?"

There are 2 ways to implement the proxy server:

1. Program the proxy server to understand the semantics of each SOAP application that a client will access. This approach is not scalable - for each new SOAP application the proxy server will need to be updated.
2. If all SOAP messages are written in RDF/DAML then it may be possible for the proxy server to dynamically discover the resource/method being requested.

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

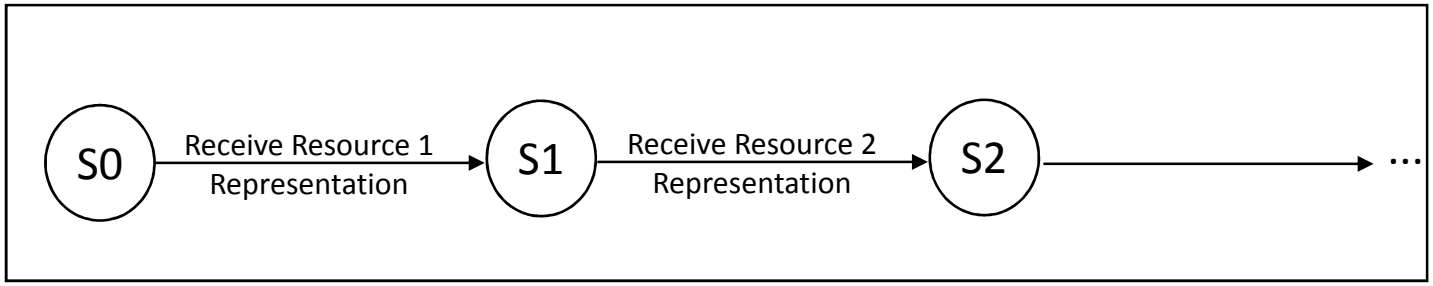
Web Intermediaries

- A proxy server is one example of a Web intermediary. Other examples are **gateways, caches**, etc. A typical Web request may be routed through multiple intermediaries.
- A Web intermediary will have a much greater chance of making reasonable decisions when a client's request shows, in the clear, the targeted resource, and the method requested is understood.
- As we have seen with SOAP both the targeted resource, as well as the requested method is nested within the SOAP envelope, thus making it much more difficult for intermediaries to do their job.

State Transitions

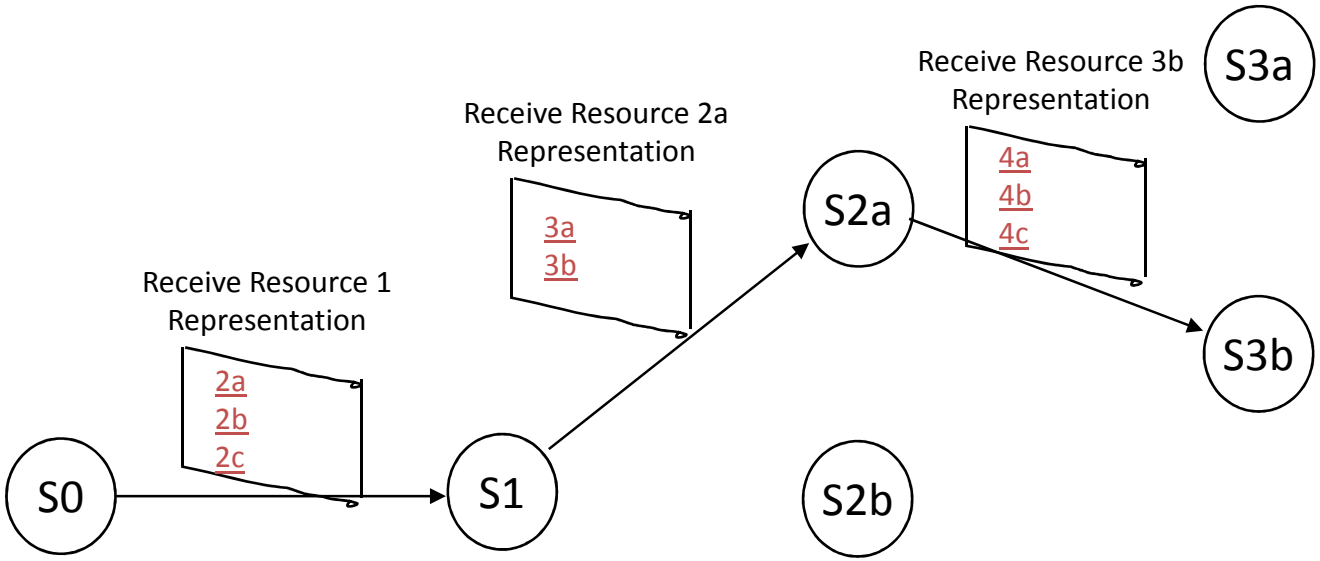
- Let us consider a client application as a state machine - each resource representation that it receives causes it to transition to the next state.

State Transition Diagram for a Client Application



Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

State Transitions in a REST-based Network



Each resource representation contains hyperlinks. Following a hyperlink takes the client to the next state. Thus, within the resource representations themselves are pointers to the next states.

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

State Transitions in a REST-based Network

Recall the Parts Depot example. The parts list resource returns this representation:

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.parts-depot.com
                   http://www.parts-depot.com/parts.xsd">
    <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>
```

This document contains hyperlinks to resources that provide detailed information about each part. When a client application follows one of the hyperlinks it receives a representation of the resource identified by the hyperlink. This transfers the client application to the next state.

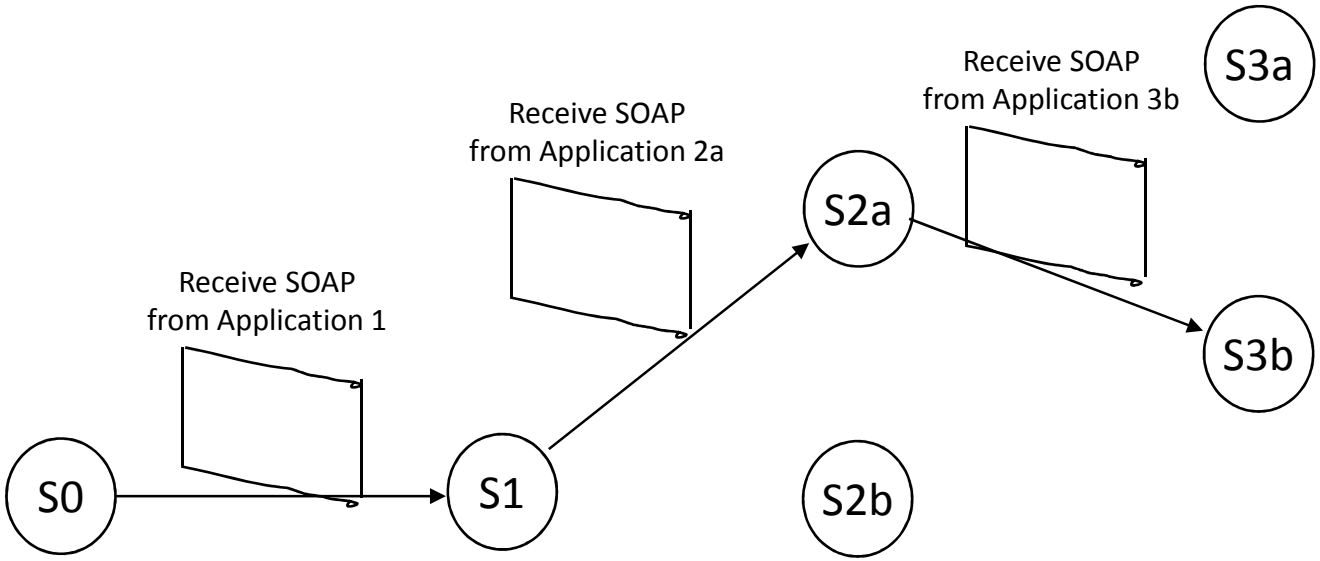
State Transitions in a REST-based Network

Question: If I am sitting at a browser then I can understand how the decision is made on which hyperlink to select (my brain decides). But if this is all being done programmatically, without human intervention, then how will the decision be made on which hyperlink to select?

Answer: The XML hyperlinking technology is XLink. With this technology in addition to providing a URL to the target resource, you can also provide data about the resource you are linking to (**using xlink:role**). If the client application is able to understand the semantics of xlink:role then it will be able to make a decision on which resource to choose next. This is ultra cool - the application is dynamically making decisions about what resources to access (it is becoming a self-propelled automata). If the client application is not able to evaluate the xlink:role attribute then the decision will need to be made out-of-band (that is, the decision must have been made when the application was written).

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

State Transitions in a SOAP-based Network



In a pure SOAP system each SOAP message will be just data, no hyperlinks. Consequently, the decision on which resource to access next must be made out-of-band.

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

State Transitions in a SOAP-based Network

Recall the Parts Depot example. The parts list resource returns this SOAP document:

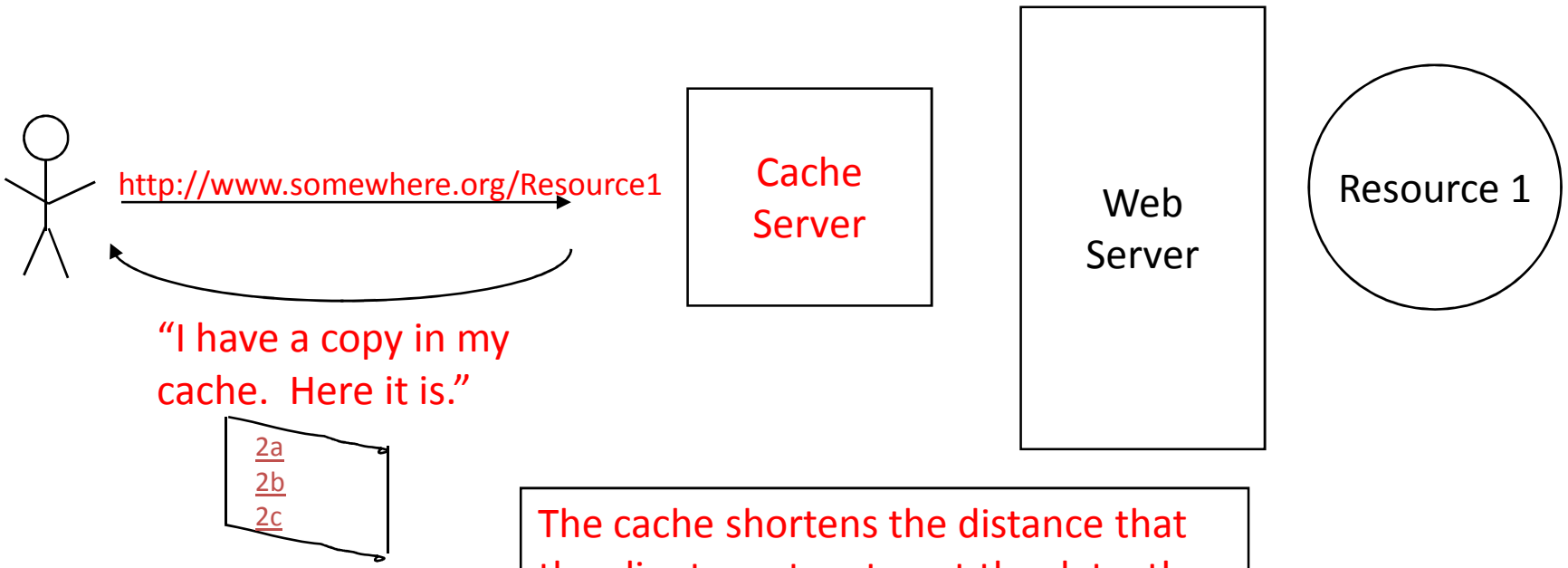
```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:getPartsListResponse xmlns:p="http://www.parts-depot.com">
      <Parts>
          <Part-ID>00345<Part-ID>
          <Part-ID>00346<Part-ID>
          <Part-ID>00347<Part-ID>
          <Part-ID>00348<Part-ID>
      </Parts>
    <p:getPartsListResponse>

  </soap:Body>
</soap:Envelope>
```

The SOAP document contains no hyperlinks. In the Web world this document is an island, cut off from the rest of the Web. Information about "what to do next" must be obtained elsewhere, i.e., out-of-band.

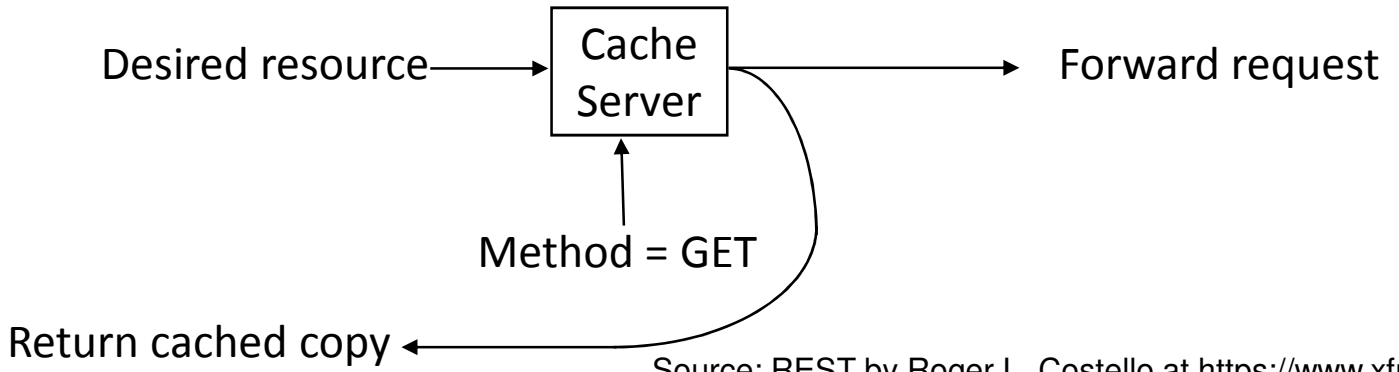
REST and Caching (Performance)



Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

REST and Caching (cont.)

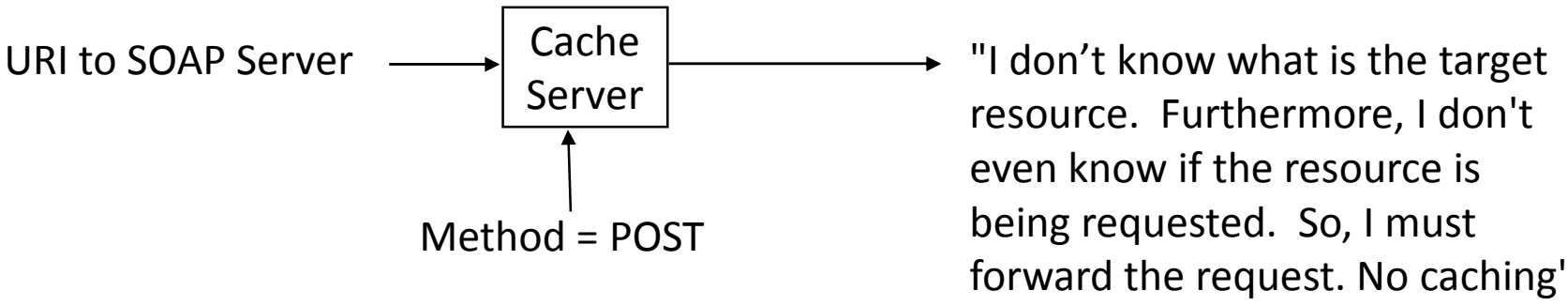
- The **results of a resource request** contains an indication in the HTTP header of whether these results are cacheable (and when the data will become stale).
- If the HTTP header says that it is cacheable, then cache servers can make a local copy.
- Later, if a client requests the same resource then the client can return a cached copy.



Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

SOAP and Caching

- When you **submit a SOAP message it is always with an HTTP POST, even though the intent of the message may be to "get" data**.
  - So a cache server would not know from the HTTP method whether the client is doing a request.
- A SOAP URI is always to the SOAP server, not to the actual target
  - Consequently, a cache server would not know from the URI what resource is being requested.
- Thus, with a SOAP message the cache server cannot determine (1) if data is being requested, nor (2) what resource is being requested.
  - Conclusion: No caching possible with SOAP!



Evolving the Web (Semantic Web)

- The vision of Tim Berners-Lee is to turn the Web into a semantic Web[1].
- One of the key components of his vision is that every resource on the Web have its own URI.
  - Axiom 0: Universality 1
    - Any resource anywhere can be given a URI
  - Axiom 0a: Universality 2
    - Any resource of significance should be given a URI.
- The REST style is consistent with this vision - every resource has a logical URI.
- SOAP URI's are funneled through a single URI to the SOAP server. This is not consistent with the semantic Web vision.

[1] http://www.w3.org/DesignIssues/Axioms.html

Generic Interface

- A key feature of REST (and the Web) is that every resource have a generic interface.
  - Access to every resource is accomplished using HTTP GET, POST, PUT, and DELETE.
- We have seen how the combination of a URI and a generic method set {URI, method} enables Web components to perform useful work:
  - Proxy Server(URI, method) -> accept/reject
  - cache(URI, method) -> forward request/return cached representation

Generic Interface (cont.)

- With SOAP there is no defined set of methods. Each SOAP application is free to define its own set of methods.
  - Consequently, tools must be customized on a per-application basis. This is not scalable. In the Web, where independent evolution and scalability are of supreme importance, this is not a very attractive idea.

Interoperability

- The key to interoperability is standardization. The reason why independent resources on the Web are able to interoperate today is because the Web has standardized on:
  - Addressing and naming resources -> URI
  - Generic resource interface -> HTTP GET, POST, PUT, DELETE
  - Resource representations -> HTML, XML, GIF, JPEG, etc
  - Media types -> MIME types (text/html, text/plain, etc)
- These are the standards that REST advocates.

Interoperability (cont.)

- SOAP depends much more on customization:
  - Addressing and naming resources -> each SOAP message provides its own unique method of naming a resource
  - Resource interface -> each SOAP application defines its own interface

Processing the Request/Response Payload

- With both REST and SOAP you need prior agreement on the semantics of the data.

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.parts-depot.com
                   http://www.parts-depot.com/parts.xsd">
   <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>
```

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>

    <p:getPartsListResponse xmlns:p="http://www.parts-depot.com">
      <Parts>
        <Part-ID>00345<Part-ID>
        <Part-ID>00346<Part-ID>
        <Part-ID>00347<Part-ID>
        <Part-ID>00348<Part-ID>
      </Parts>
    <p:getPartsListResponse>

  </soap:Body>
</soap:Envelope>
```

With both REST and SOAP client applications will need to understand these responses.

Source: REST by Roger L. Costello at https://www.xfront.com/REST.ppt

Processing the Request/Response Payload

- Note: if the payload is in the format of RDF or DAML then the client application may be able to dynamically learn the meaning of the response data.
- As we saw earlier, with REST there are links to the next state built within each response. We hinted earlier at how a client application may be able to reason dynamically about which link to traverse (using xlink:role). Thus, dynamic learning of response data combined with dynamic reasoning of link traversals would yield a self-reasoning automata. This is the next step of the Web!

REST vs SOAP

Feature	REST	SOAP
Specification	JAX-WS	JAX-RS
Language & Platform Independent	Yes	Yes
Transport Protocol	HTTP	HTTP, SMTP, JMS
Message Size	Lightweight (no extra XML markup)	Heavyweight (has SOAP specific Markup)
Message Communication	XML, JSON, other valid MIME Type	XML
Caching	Response of GET operations can be cached	No

REST Best Practices

1. Provide a URI for each resource that you want (or will want) exposed. This is consistent with Tim Berners-Lee's axioms for the Web, as well as the W3 TAG recommendations.

2. Prefer URIs that are logical over URIs that are physical. For example:
   Prefer:
   http://www.boeing.com/airplanes/747
   Over:
   http://www.boeing.com/airplanes/747.html
Logical URIs allow the resource implementation to change without impacting client applications.

3. As a corollary to (2) use nouns in the logical URI, not verbs. Resources are "things" not "actions".

4. Make all HTTP GETs side-effect free. Doing so makes the request "safe".

5. Use links in your responses to requests! Doing so connects your response with other data. It enables client applications to be "self-propelled". That is, the response itself contains info about "what's the next step to take". Contrast this to responses that do not contain links. Thus, the decision of "what's the next step to take" must be made out-of-band.

REST Best Practices (cont.)

6. Minimize the use of query strings. For example:

Prefer:
http://www.parts-depot.com/parts/00345
Over:
http://www.parts-depot.com/parts?part-id=00345

Rationale: the relationship between 'parts' and '00345' is clear, and you can instantiate sub-resources of '00345' easily; this is not possible if that information is tucked away in a query string.

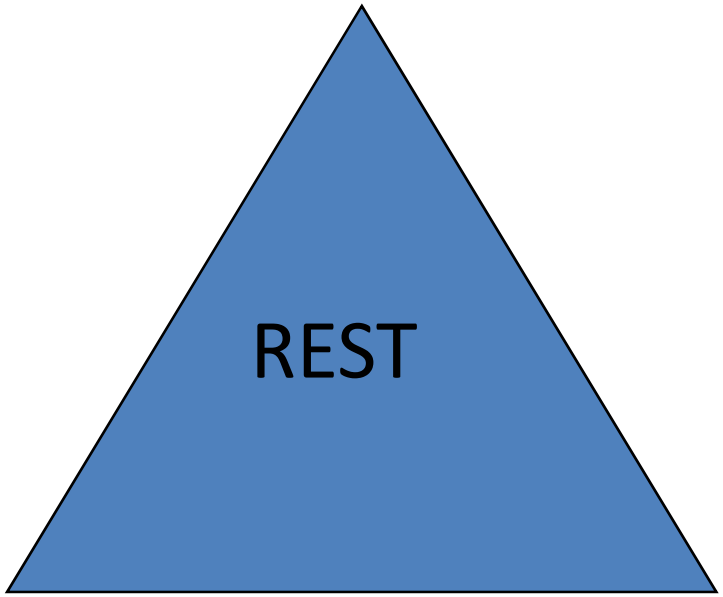
7. Use the slash "/" in a URI to represent a parent-child, whole-part relationship.

8. Use a "gradual unfolding methodology" for exposing data to clients. That is, a resource representation should provide links to obtain more details.

9. Always implement a service using HTTP GET when the purpose of the service is to allow a client to retrieve a resource representation i.e. don't use HTTP POST method

Summary

Nouns (Resources)
unconstrained
i.e., http://example.com/employees/12345



Verbs
constrained
i.e., GET

Representations
constrained
i.e., XML, JSON

Is this API “fully RESTful” ?

- Is there any way to conclude that this API is “fully RESTful” or Not?
- Answer can be in “yes” or “no”
  - However, there is a spectrum
  - We can answer the question using a model developed by Leonard Richardson: Richardson Maturity Model.
  - The model defines four levels: level 0 to level 3

Richardson Maturity Model

- Level 0
  - SOAP web services: One URI (where web service is exposed) which is used to receive all requests from clients.
  - The request body of message contains all the details: How service knows what to do or what operation to execute. The message contains the details like operation to be performed and data required for operation.
  - Example

```
<create-message>
    <message-content> Hello World </message-content>
    <message-author> ABCD </message-author>
<create-message>
```

```
<delete-comment>
    <message-id> 30 </message-id>
    <comment-id> 2 </comment-id>
</delete-comment>
```

Both these messages will be sent to the same URI. The action is part of message itself, the same URI could work. The same HTTP method (POST) could work. No HTTP concepts are used in communication.

Richardson Maturity Model

- Level 1
  - Refines the Level 0 and introduces the concept of Resource URI. The starting level of RESTful API.
  - Individual URIs for each resource. Message request goes to one URI and comment request goes to another URI. The information about operation to execute still resides inside message.
  - Example

```
<create-message>
    <message-content> Hello World </message-content>
    <message-author> ABCD </message-author>
<create-message>
```

```
<delete-comment>
    <message-id> 30 </message-id>
    <comment-id> 2 </comment-id>
</delete-comment>
```

Both these messages will be sent to the different URIs. The action is part of message itself.

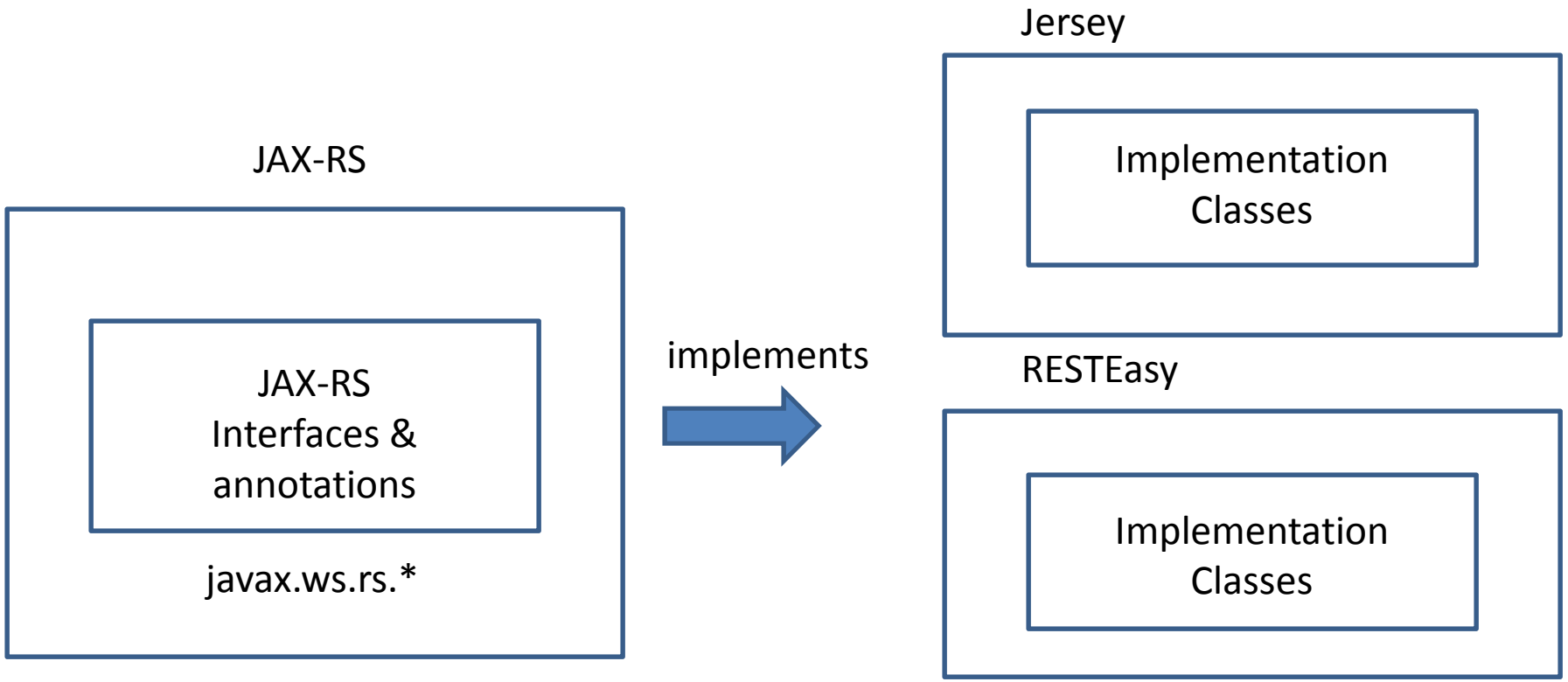
Richardson Maturity Model

- Level 2
  - If different HTTP methods are used for different operations then it is Level 2. API of Level 2 uses standard HTTP methods: GET, PUT, POST, DELETE to do different operations on Resource URI.
    - The URI specifies the resource being operated upon and HTTP method specifies what the operation is.
  - API at Level 2 also uses HTTP status codes and right use of idempotent and non-idempotent methods.
- Level 3
  - Level 3 API implements HATEOAS
  - The response has links that controls application state of client. All the URIs clients needs send to them in response.
  - Considered as fully RESTful API

JAX - RS

- JAX-RS is JAVA API for developing RESTful Web Service.
  - By using JAX-RS, you can turn a simple Java POJO into a RESTful resource.
- There are several libraries
  - RESTEasy (Jboss), Jersey (Sun/Oracle), Restlet
- Which Library to choose?
  - Learn any one. You have learned them all.
  - These libraries follow common interface. API is common across libraries
  - If you have used Jersey in your application and want to replace it by RESTEasy, you can do it because these libraries follow common interface (Plug and Play).
  - This common interface (API) is called JAX – RS.

JAX - RS

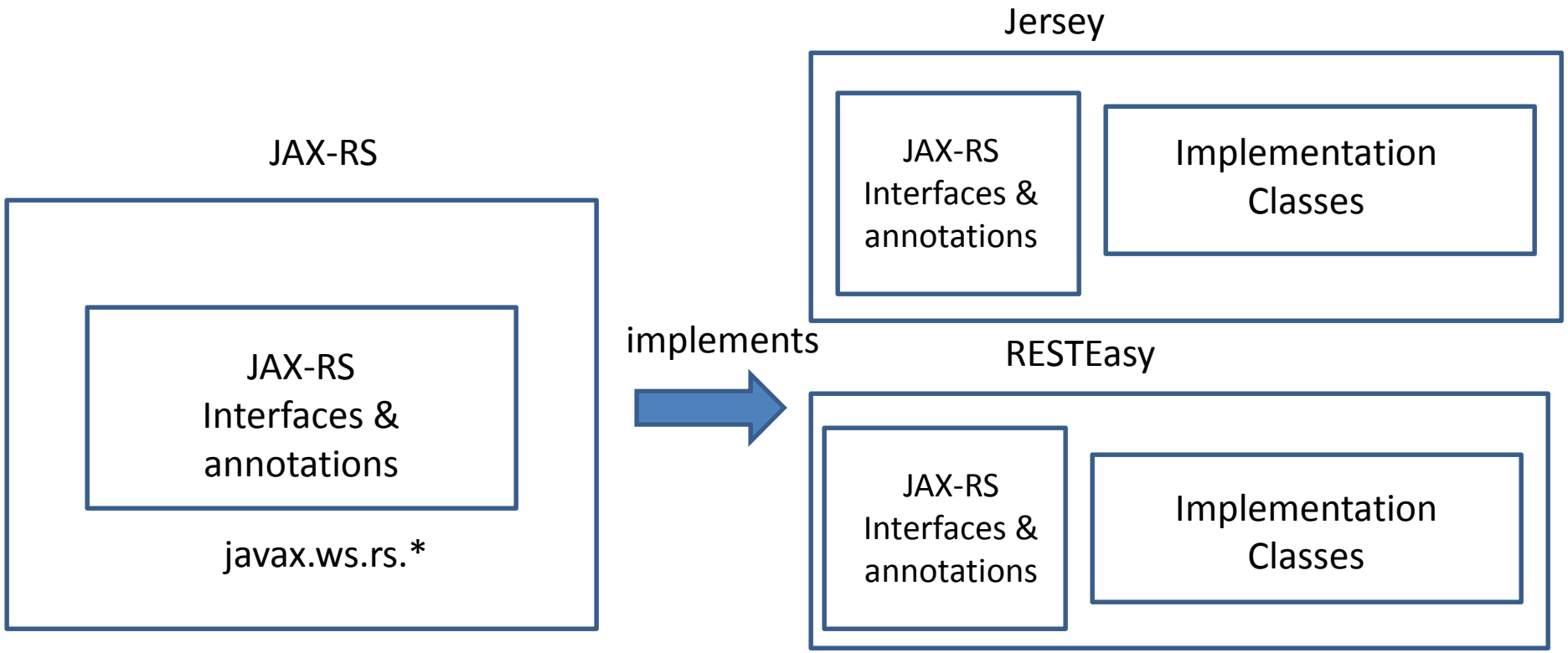


JAX-RS API is a bunch of interfaces and annotations. The JAX-RS does not have any functionality (classes that implement the interfaces and read the annotations).

The libraries (Jersey, RESTEasy) look at the code and find where you have called JAX-RS API and then do the actual work.

You have deployed your application on server (Glassfish). You have to include (a) JAX-RS API and (b) one of these libraries with your application in server.

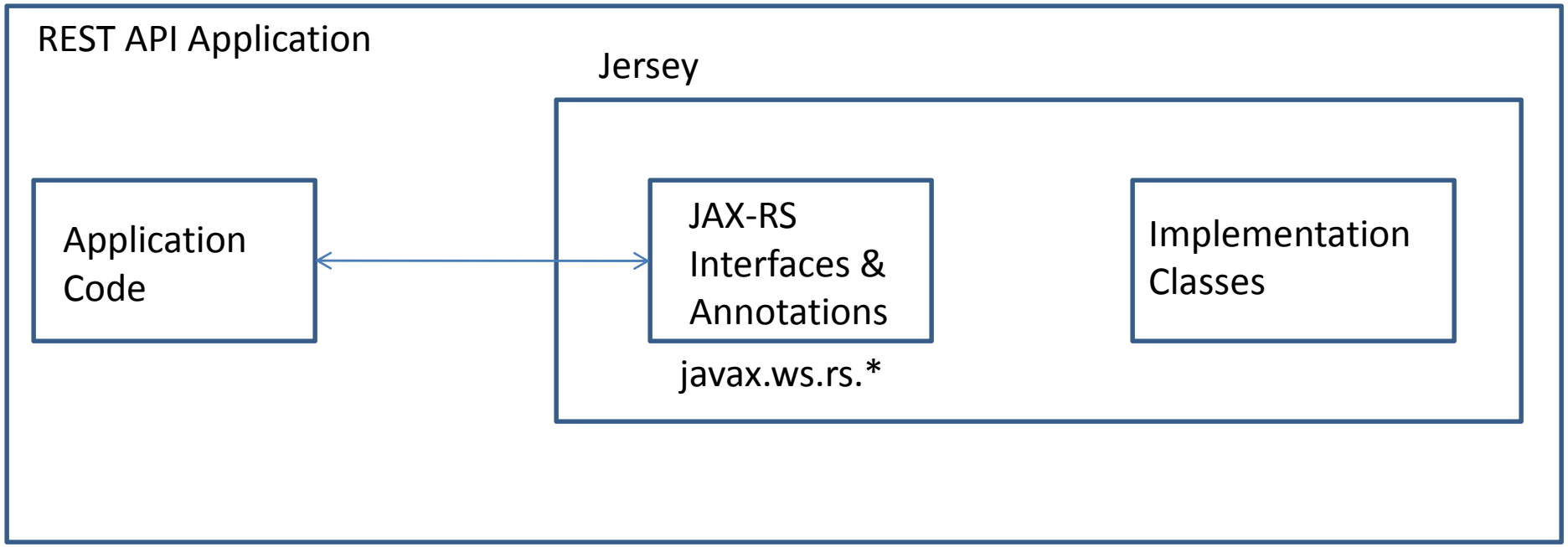
JAX - RS



You don't have to include two different things (JAX-RS API and any one of these libraries) in server. Every JAX-RS library comes with copy of API classes already.

You have to include only library of your choice. You can seamlessly replace one library with other.

JAX - RS



Your application depends on JAX-RS API (Interface & Annotations).

Jersey is “Reference Implementation” for JAX-RS specifications (by persons who designed JAX-RS specifications)

JAX-RS Annotations

- **@ApplicationPath("/api")**
  - Serves as base URI for all our resources URIs. All resources are to be found at the root */api*. The URL should look: *http://localhost:8080/webcontext/api/* where webcontext is the name of your application.
- **@Path("/books")**
  - URI path to the resource. URI to the book resource is */api/books* and the URL would be *http://localhost:8080/webcontext/api/books*
  - Once the path to our resource has been defined the individual resource method are configured for the HTTP method and context type.
- **@GET**
  - Methods annotation with the @GET annotation respond to HTTP get requests.
- **@POST**
  - Methods annotated @POST respond to POST method requests. The POST HTTP method is commonly used to create a resource.
- **@PUT**
  - The @PUT annotation is used for updating a record

JAX-RS Annotations

- **@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})**
  - Used to specify the media type or types that the method returned to the caller
- **@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})**
  - The media type that a method is capable of consuming can be specified with the annotation @Consumes.
- **@*Param (@PathParam, @HeaderParam, @QueryParam, @CookieParam)**
  - Used for retrieving request parameter
  - **@CookieParam:** Imagine you have sent a cookie called *cartId* to the clients so that you can track the customer's shopping cart. To pull the cookie from the HTTP request just annotate the method parameter to which you want the cookie data to be assigned.
  - **@HeaderParam:** used to inject HTTP request header values into resource method parameters

References

- Articles on REST by Roger L. Costello, Timothy D. Kehoe
  - https://www.xfront.com/files/rest.html
  - 5-Minute Introduction to REST (Learning REST by Example)
    - https://www.xfront.com/5-minute-intro-to-REST.ppt by Roger L. Costello, Timothy D. Kehoe
  - An Expanded Introduction to REST
    - https://www.xfront.com/REST-full.ppt by Roger L. Costello, Timothy D. Kehoe
  - REST
    - https://www.xfront.com/REST.ppt by Roger L. Costello
- Articles on REST by Paul Prescod
  - Second Generation Web Services
    - http://www.xml.com/pub/a/2002/02/06/rest.html
  - REST and the Real World
    - http://www.xml.com/pub/a/2002/02/20/rest.html
  - SOAP, REST and Interoperability
    - http://www.prescod.net/rest/standardization.html
  - Evaluating XML for Protocol Control Data
    - http://www.prescod.net/xml/envelopes/
- Youtube Videos on Developing RESTful APIs with JAX-RS by Java Brains
  - https://www.youtube.com/watch?v=xkKcdK1u95s&list=PLqq-6Pq4lTTZh5U8RbdXq0WaYvZBz2rbn

Thank You !