Architectural Style: Representational State Transfer (REST)

CSCE 742 - Lecture 10 - 10/04/2018

The Parts Depot Web Store

- Parts Depot, Inc wants to deploy a web service to enable its customers to:
  - Get a list of parts.
  - Get detailed information about a particular part.
  - Submit a Purchase Order (PO).
- How would you architect this?

RESTful Services

- REST is an architectural style for implementing web-based systems.
  - RESTful services provide resources (as web pages) designed to be consumed by programs rather than by people.
  - Design Principles:
    - Stateless
    - Resource-Based (URI)
    - Uniform Interface (GET, PUT, POST, DELETE)
    - Links describe relationships
    - Cacheable and monitorable using standard internet tools

Today's Class

- Introduce HTTP requests.
  - "The API of the Web"
- Introduce the REST architectural style.
  - Design principles
  - Fulfillment of quality properties
  - Implementation of RESTful services

The Interface of the Web

HTTP

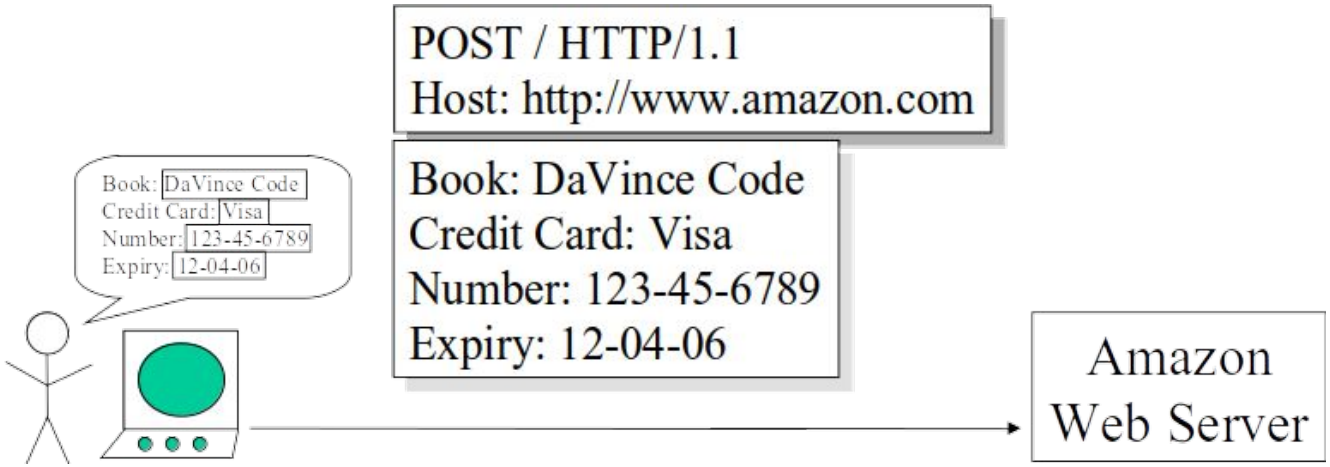
- The Hypertext Transfer Protocol is an communication protocol used for distributed networked systems.
  - Defines how to exchange or transfer hypertext between nodes in a network.
  - AKA, how your computer can access a webpage.
- Defines an API based on requests.
- Requests are performed using **verbs**.
  - I **get** a page, I **post** an update, I **delete** a photo, I **put** up my information.

Retrieving Information (GET)

- User types into the browser: http://www.amazon.com
- The browser creates an HTTP request (no body)
- The HTTP request identifies:
  - The desired action: GET ("get me resource")
  - The target machine (www.amazon.com)

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Updating Information (POST)



- The user fills in a form on a webpage.
- The browser creates an HTTP request with a body comprised of the form data
- The HTTP request identifies:
  - The action: POST ("here is some updated info")
  - The target machine (amazon.com)
- The body contains:
  - The data being POSTed (the form data)

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

The HTTP API

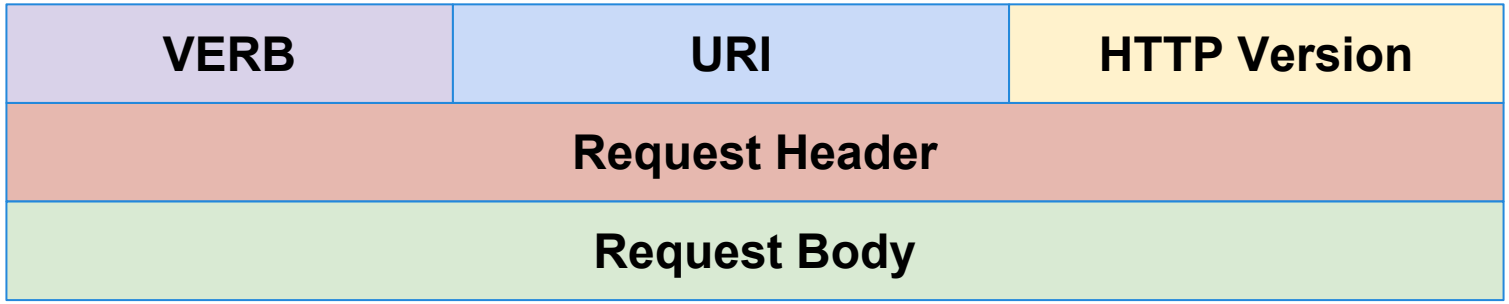
- HTTP provides a simple set of operations.
  - Amazingly, all Web exchanges are done using this simple HTTP API.
- Based on the idea of CRUD (Create, Retrieve, Update, and Delete)
  - PUT: “Here is some new info” (Create)
  - GET: “Give me some info” (Retrieve)
  - POST: “Here is some updated info” (Update)
  - DELETE: “Get rid of this info” (Delete)

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Additional Verbs

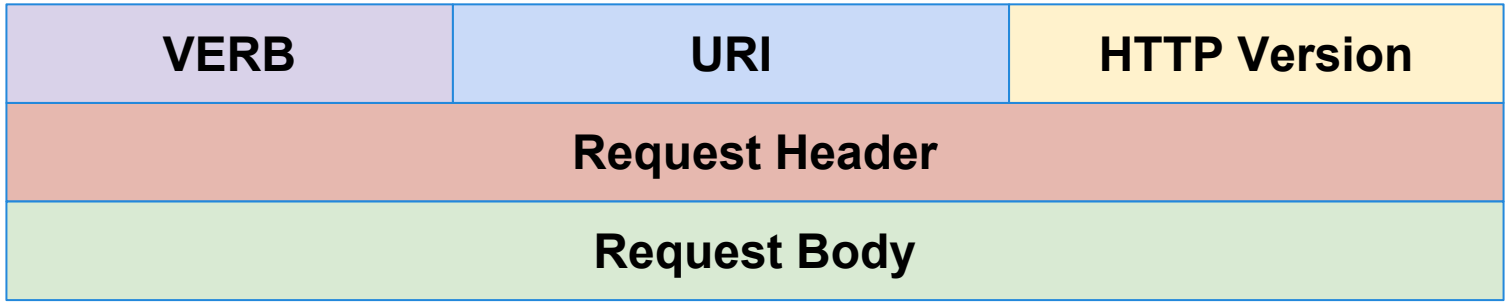
- HEAD
  - “Give me the metadata”
- TRACE
  - “Show me what changes have been made”
- OPTIONS
  - “Tell me what verbs you have implemented for this resource.”
- PATCH
  - “Apply partial resource modification”

Anatomy of an HTTP Request



- <VERB> is one of the HTTP verbs
- <URI> is the URI of the resource
- <HTTP Version> is the version of HTTP
- <Request Header> contains metadata
  - Collection of key-value pairs of headers and their values.
  - Information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and more.
- <Request Body> is the actual message content.

HTTP Request Examples



GET:

GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1

Host: www.w3.org, Accept: text/html,application/xhtml+xml,application/xml; …,
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 …,
Accept-Encoding: gzip,deflate,sdch, Accept-Language: en-US,en;q=0.8,hi;q=0.6

POST:

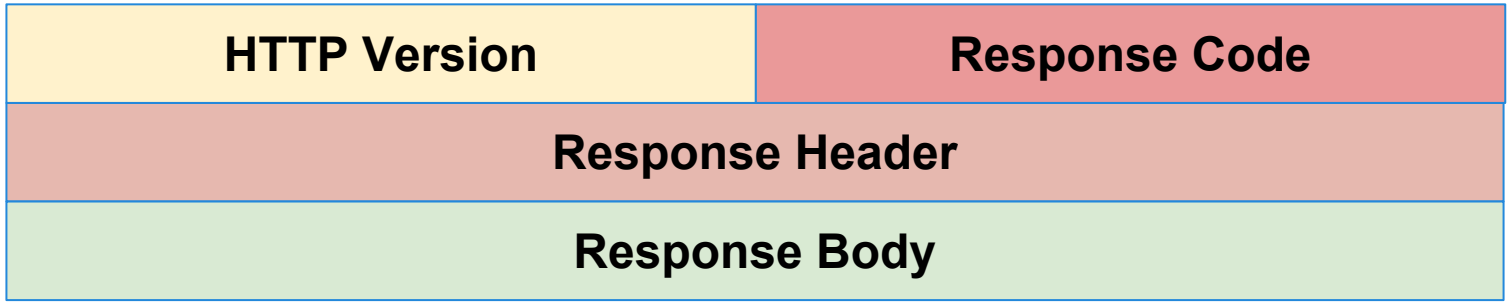
POST http://MyService/Person/ HTTP/1.1

Host: MyService, Content-Type: text/xml; charset=utf-8, Content-Length: 123

<?xml version="1.0" encoding="utf-8"?>

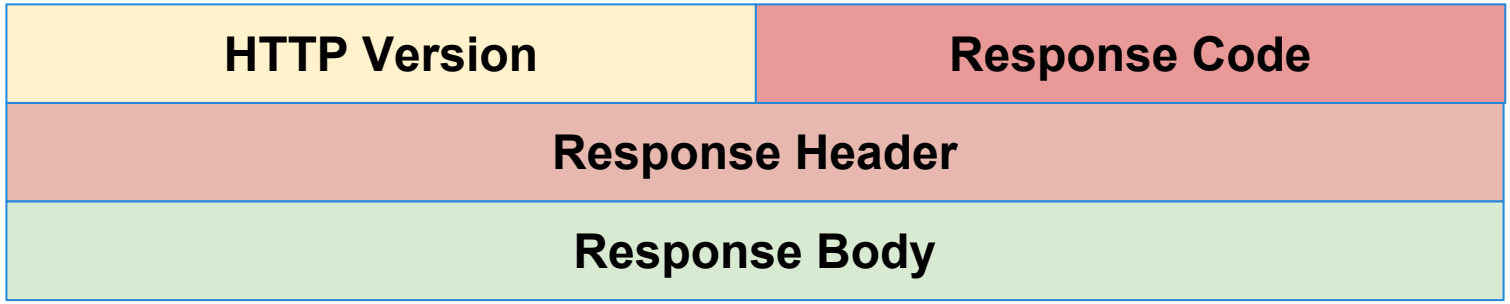
<Person><ID>1</ID><Name>M Vaqqas</Name>
<Email>m.vaqqas@gmail.com</Email><Country>India</Country></Person>

Anatomy of an HTTP Response



- <Response code> contains the status of the request. It is almost always a 3-digit HTTP status code from a pre-defined list.
- <Response Header> contains the metadata and settings about the response message.
- <Response Body> contains the representation if the request was successful.

HTTP Response Example



- GET Response:

HTTP/1.1 200 OK

Date: Sat, 23 Aug 2014 18:31:04 GMT, Server: Apache/2, Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT, Accept-Ranges: bytes, Content-Length: 32859, Cache-Control: max-age=21600, must-revalidate, Expires: Sun, 24 Aug 2014 00:31:04 GMT, Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns='http://www.w3.org/1999/xhtml'> <head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head> <body> …

HTTP Status Codes

- Common Responses:
  - 200 Ok (succeeded)
  - 201 Created (a new resource)
  - 202 Accepted (not completed)
  - 204 No Content (fulfilled request, nothing to return)
  - 205 Reset content (reload page)
  - 206 Partial content
  - 301 Redirection: moved permanently
  - 302 Redirection: found (temporary move)
  - 400 Bad request
  - 401 Unauthorized
  - 402 Payment Required
  - 404 Not found

Representational State Transfer (REST)

Representational State Transfer

- A Client references a resource using a URI.
- A **representation** of the resource is returned.
  - Receiving the representation 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 into yet another state.
  - Thus, the client application **transfers** state with each resource representation.

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

The Core Idea

- REST is modeled after the natural workflow of the Internet.
  - Motivation: Create a design pattern for how the web should work, such that it could serve as a guiding framework for designing web services.
  - A well-designed web application behaves as a network of web pages (a virtual state-machine).
  - The user progresses through an application by selecting links (state transitions).
  - Resulting in the next page (the next state of the application) being transferred to the user and rendered for their use.

REST - Not a Standard

- REST is not a standard.
  - W3C will not put out a REST specification.
  - Microsoft does not sell a REST developer's toolkit.
- REST is just an architectural pattern.
  - You can't bottle up a pattern.
  - You can only understand it and design your web services based on it.
- REST does prescribe the use of standards:
  - HTTP, URL
  - XML/HTML/JPEG/etc. (Resource Representations)
  - text/xml, text/html, image/gif, image/jpeg, etc. (Resource Types, MIME Types)

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Verbs in REST

- Verbs (loosely) describe actions that are applicable to nouns
- Using different verbs for every noun would make widespread communication impossible
- Some verbs only apply to a few nouns
  - In programming we call this “polymorphism”
- In REST, we use **universal verbs**
  - All RESTful services offer the same interface.
  - Based on HTTP requests and responses.

REST Fundamentals

- Services offer **resources** that can be interacted with.
- All resources have a unique **URI**.
  - Often web addresses (blah.com/form.html)
  - Not necessarily HTML - resources are just concepts.
  - URIs tell a client that there's a concept somewhere.
    - Clients can then request a specific representation of the concept from the representations the server makes available.
- HTTP **verbs** are used to retrieve or manipulate resources in a clear, universal manner.

Representation Formats

- Both client and server should be able to comprehend the format.
  - Structured content is often JSON or XML
  - JSON = Javascript Object Notation
- A representation should completely represent a resource.
  - If there is a need to partially represent a resource, break the resource into child resources.
  - Smaller representation = easier to transfer = less time required to create representation = faster services.
- The representation should be capable of linking resources to each other via URIs.

Verb Guarantees

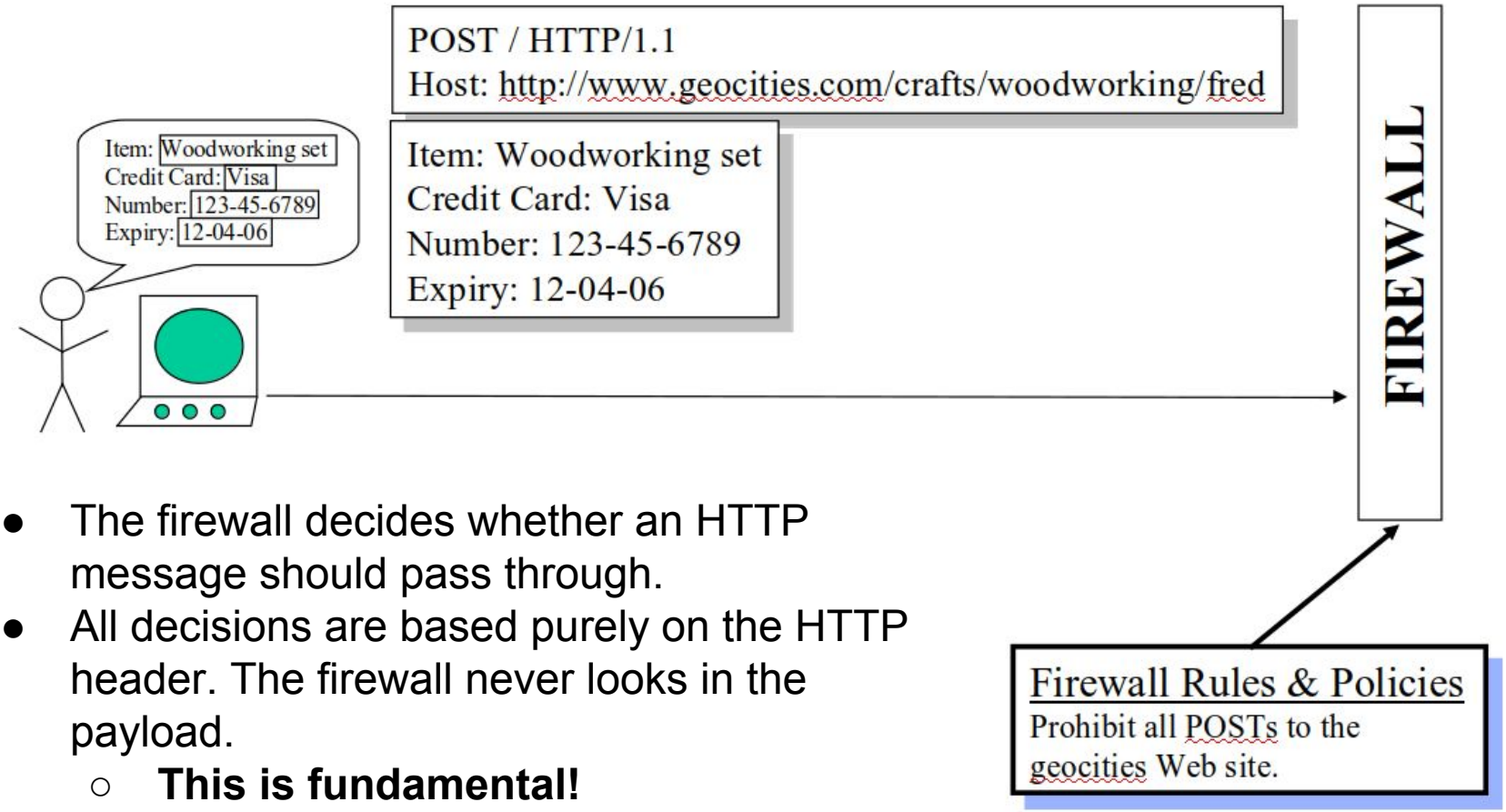
- GET, OPTIONS, TRACE, and HEAD are safe operations.
  - They should not change the resource in any way.
  - Be careful - no technical limitations ensuring safety.
    - A service may allow deletion of a resource by accessing a URL. A GET request to that URL would cause deletion.
- PUT and DELETE are idempotent.
  - Repeated requests should have the same effect as a single request.
  - Safe operations are also idempotent.
  - POST is not idempotent.

Elements of Web Architecture

- **Firewalls** decide which HTTP messages get out, and which get in.
  - These components enforce web *security*.
- **Routers** decide where to send HTTP messages.
  - These components manage web *scalability*.
- **Caches** decide if a saved copy of a resource can be used.
  - These components increase web *performance*.

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Firewalls



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

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

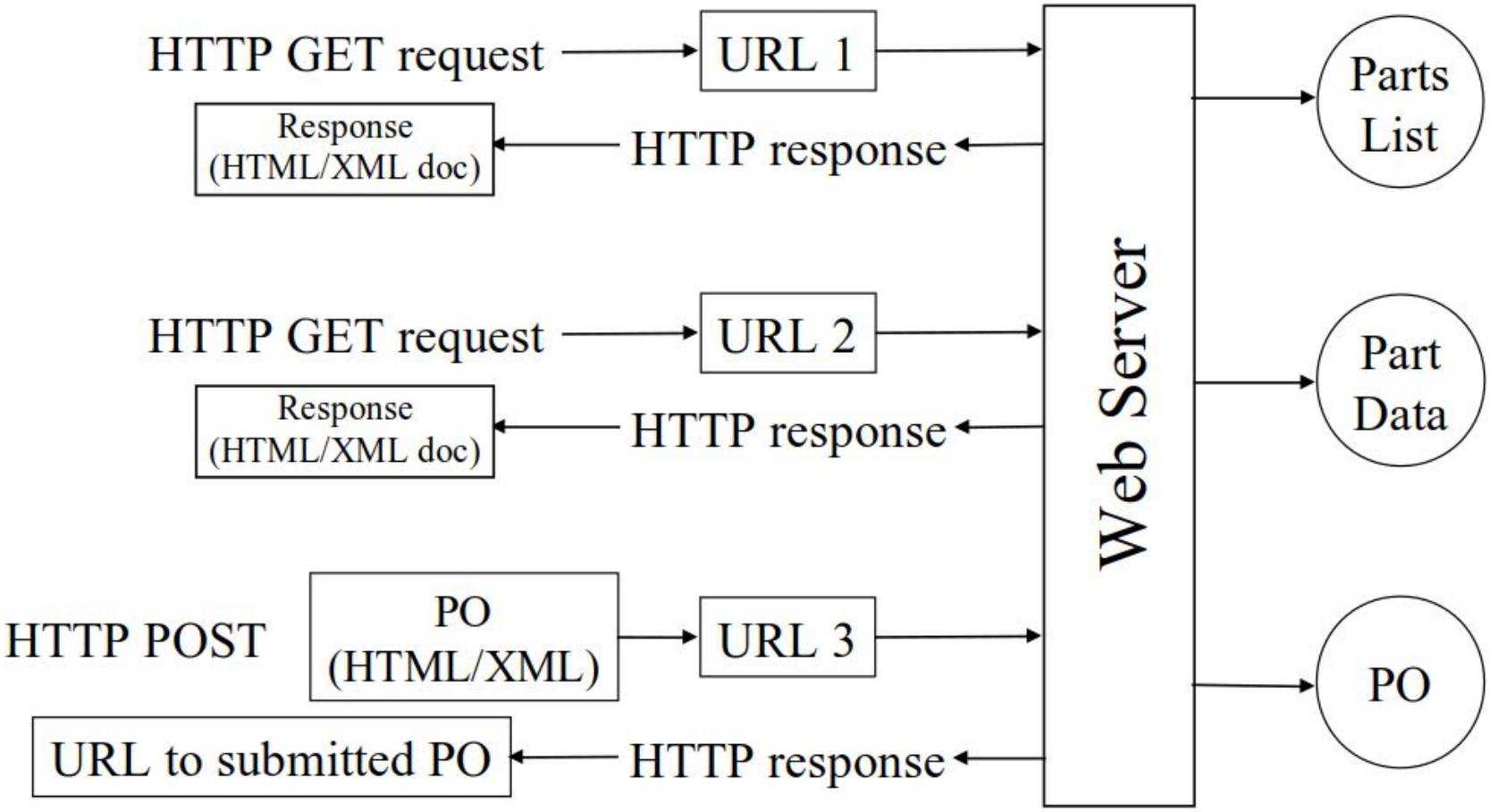
Privacy of Content

- All three components base decisions and actions purely upon information in the HTTP header.
  - **They should never examine the request body.**
- Letter analogy: The postal service doesn't look inside your letter (this is illegal), they just act based on addressing on the outside.
  - REST enforces a similar idea - the content should not matter, just the metadata.
  - Protects privacy of data.

The Parts Depot Web Store

- Parts Depot, Inc wants to deploy a web service to enable its customers to:
  - Get a list of parts.
  - Get detailed information about a particular part.
  - Submit a Purchase Order (PO).
- How would you architect this?
  - Let’s discuss the RESTful way to design this.

The RESTful Way of Designing the Web Services



Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Retrieving a List of Parts

Service: Get a list of parts

- The web service makes available a URL to a parts list **resource**.
- A client posts a GET request to this URL to get the parts list:
  - http://www.parts-depot.com/parts
- **How** the web service generates the parts list is completely transparent to the client.
  - **This enforces loose coupling.**

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

REST Fundamentals:

1. Create a resource for every service.
2. Identify each resource using a URI.

Data Returned: Parts List

```
<?xml version="1.0"?>
<Parts>
    <Part id="00345" href="http://www.parts-depot.com/parts/00345"/>
    <Part id="00346" href="http://www.parts-depot.com/parts/00346"/>
    <Part id="00347" href="http://www.parts-depot.com/parts/00347"/>
    <Part id="00348" href="http://www.parts-depot.com/parts/00348"/>
</Parts>
```

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

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

REST Fundamental:

The data that a service returns should link to other data.

- **Thus, design your data as a network of information.**
- **Contrast with OO design, which says to encapsulate information.**

Retrieving Details on a Part

Service: Get detailed information about a particular part

- The web service makes available a URL to each part resource.
- For example, a client can request information on a specific part by posting a GET request to http://www.parts-depot.com/parts/00345

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Data Returned: Part 00345

```
<?xml version="1.0"?>
<Part>
    <Part-ID>00345</Part-ID>
    <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</Part>
```

- Again, 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.

Designing Services with REST

Designing Services With REST

- Destination URL must be placed in the HTTP header for Web components to operate effectively.
  - Web components (firewalls, routers, caches) make their decisions based upon information in the HTTP Header.
  - Infrastructure will not work correctly if destination not in URL!
- It is anti-REST if the HTTP header just identifies an intermediate destination and the payload identifies the final destination.

Designing Services With REST

- Client requests should be **idempotent** – multiple calls should lead to the “same” response.
- Server responses are “idempotent”, but only in terms of the meaning of information and not necessarily the content.
  - Think of a URL that always returns the current time…

PUT and POST

- PUT is idempotent, POST is **not**.
  - No matter how many times PUT request is made (>=1), server state will be the same.
  - Multiple POSTs may create multiple resources.
- PUT requires a full resource ID path.
  - Client creates resource.
- POST does not require full resource ID path.
  - Server notifies client of resource location.
  - Post can still be used for resource updates.

PUT and POST

- PUT http://MyService/Persons/
  - Won't work. PUT requires a complete URI.
- PUT http://MyService/Persons/1
  - Insert a new person with PersonID=1 if it does not already exist, or else update the existing resource with the payload.
- POST http://MyService/Persons/
  - Insert a new person every time this request is made (using the payload) and generate a new PersonID.
- POST http://MyService/Persons/1
  - Update the existing person where PersonID=1

Handling POSTs

- Other methods are idempotent, but POST creates new resources.
- Multiple POSTs of the same data must be made harmless.
  - Put message ID in a header or in the message body.
  - This renders multiple posts harmless.
    - Prevents “multiple charge” issue with web stores.

Handling POSTs

- Many ways to do this:
  - Exact: client or server-side unique transaction ID.
  - Heuristic: check for and remove “likely duplicates” from POST stream.
- Wasted IDs are irrelevant.
  - Duplicated POSTs are not acted on by the server
- The server must send back the same response the original POST got, in case the application is retrying because it lost the response.

Idempotence

- What does this mean, strictly speaking?
  - Call to server must return the same thing each time?
  - No side effects?
- What about changing data?
  - Time-of-day service.
  - Each GET call returns a new time.
  - Is this still RESTful?
    - As long as the **resource is constant**.
    - The value does not need to be constant, just how we access it.

Statelessness

- Each request contains all information needed to service the request.
- No client state is held on the server.
- Benefits in terms of scale and availability.
- Negatives:
  - Development is harder.
  - Performance may be worse (multiple requests may be needed to get information).

Caching

- An important part of scaling services is caching generated results.
- Clients and servers can cache responses.
- Responses must be defined as cacheable or not to prevent clients from getting out-of-date information.
- Well-managed caching removes some client-server interactions, improving scalability and performance.

Caching

- Controlled through HTTP header metadata.
  - Date
    - Date and time when this representation was generated.
  - Last Modified
    - Date and time when the server last modified this representation.
  - Cache-Control
    - Used to control caching. Allows specification of max or min time, can prevent storage.
  - Expires
    - Expiration date and time for this representation.
  - Age
    - Duration passed in seconds since this was fetched from the server. Can be inserted by an intermediary component.

Cache-Control Response Directives

- **Public:** Indicates any component can cache this representation.
- **Private:** Intermediary components cannot cache this representation, only client or server can do so.
- **no-cache/no-store:** Caching turned off.
- **Max-age**: Duration in seconds after the date-time marked in the Date header for which this representation is valid.
- **S-maxage**: Similar to max-age but only meant for the intermediary caching.
- **Must-revalidate**: Indicates that the representation must be revalidated by the server if max-age has passed.
- **Proxy-validate:** Similar to max-validate but only meant for the intermediary caching.

Well-Structured URIs

- Avoid using spaces. Use an _ (underscore) or – (hyphen) instead.
- Remember that URIs are case insensitive.
- Stay consistent with naming conventions throughout the service..
- URIs are long lasting.
  - If you need to change the location of a resource, do not discard the old URI. If a request comes for the old URI, use status code 300 and redirect the client to the new location.

Well-Structured URIs

- Avoid verbs for your resource names unless your resource is actually an operation or a process.
  - Bad URIs:
    - http://MyService/FetchPerson/Mike
    - http://MyService/DeletePerson?id=Mike
  - Good URI:
    - http://MyService/Persons/Mike
    - You can apply verbs to this resource.

Food For Thought

What if Parts Depot has a million parts, will there be a million static pages?

http://www.parts-depot/parts/000000

http://www.parts-depot/parts/000001

...

http://www.parts-depot/parts/999999

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Food For Thought

- Distinguish between logical and physical entities.
- The URLs are **logical**.
  - They express what resource is desired.
  - They do not identify a physical object.
  - The advantage of using a logical identifier (URL) is that changes to the implementation of the resource will be transparent to clients (loose coupling!).
- Parts Depot will store all parts data in a database. Code at the Parts Depot website will receive each logical URL request, parse it to determine which part is being requested, query the database, and generate the part response document to return to the client.

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Food For Thought

Physical URLs

http://www.parts-depot/parts/000000.html

http://www.parts-depot/parts/000001.html

...

http://www.parts-depot/parts/999999.html

Logical URLs

http://www.parts-depot/parts/000000

http://www.parts-depot/parts/000001

...

http://www.parts-depot/parts/999999

- Physical URLs are pointing to HTML pages.
- If there are a million parts, it will not be very attractive to have a million static pages.
- Furthermore, changes to how these parts data is represented will effect all clients that were using the old representation.

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Food For Thought

What if I have a complex query?

Show me all parts whose unit cost is under $0.50 and for which the quantity is less than 10

How would you do that with a simple URL?

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Food For Thought

- For complex queries, you can provide a service (resource) for a client to retrieve a form that the client then fills in.
- When the client hits "Submit", the browser will gather up the client's responses (form data) and generate a URL based on the responses.
  - Often, the client doesn't generate the URL (think about using Amazon - you start by entering the URL to amazon.com; from then on you simply fill in forms, and the URLs are automatically created for you).

Slide from: REST by Roger Costello and Timothy Kehoe http://www.xfront.com/REST-full.ppt

Activity - Let's Make a Deal

- Game where contestants are presented with three doors.
  - One leads to a great prize, the other leads to nothing.
  - Users select one door.
  - Host opens one of the other doors.
  - Users can then choose to open their door or the remaining unopened door.

Activity - Let's Make a Deal

You have been asked to implement Let's Make a Deal as a web service. You must support:

- Creation of games.
- User selection of a door.
  - The game will open one of the other doors.
- User opening of a door.
- Querying of the current state of the game and outcome (if complete) by user.
- Deletion of a game.

Determine the appropriate resources, verbs, and response messages.

Sample API

Resource	Verb
/games	get – status of games server post – create new game
/games/{gid}	get – status of game (in_play, won, lost) delete – delete the game resource
/games/{gid}/doors	get – status of all doors
/games/{gid}/doors/{1..3}	get – door status {closed, selected, opened} put – update door status

- Use status codes to determine whether an operation is reasonable.
- Once game is finished (won/lost), only GET requests are allowed.

Key Points

- REST is an architectural style for implementing web-based systems.
  - RESTful services provide resources (as web pages) designed to be consumed by programs rather than by people.
  - Design Principles:
    - Stateless
    - Resource-Based (URI)
    - Uniform Interface (GET, PUT, POST, DELETE)
    - Links describe relationships
    - Cacheable and monitorable using standard internet tools

Next Time

- The Information Viewpoint
  - Sources: Rozanski and Woods, Ch: 18

- Midterm coming up soon
  - Practice midterm on site, with no answers.
  - In Lec 12 (10/11), we will go over answers
- Homework:
  - Project 2 - 10/11
  - Assignment 2 - 10/25