PART 2.
DATA STORAGE AND FLOW MANAGEMENT

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

FAQs

- Term project submission TP2
  - Your source code and final report (04/27/2016)
  - TEAM Demonstration (Next week)
- Term project TP3
  - TEAM Demonstration (Next week)
  - You are required to attend at least 2 of above sessions
    - Participation score (4)
    - Your attendance and questions/feedbacks during the presentation
- Final exam: 05/11/2016
  - 20%
  - Same format as the midterm
  - Not comprehensive

Today’s topics

- FAQs
- Data Exchange models

Wearable devices and sensors

- Guest lecture
  - 04/28/2016
  - Participation score
Google Fit

- Store, read, analyze user’s activity data
- Data collected from user’s devices are stored in anywhere available
- Immediate and historical analysis

For more information: https://developers.google.com/fit/

Google Fit APIs

- Sensors API
- Recording API
- History API

Sensors API

- Connecting your wearable devices
- Create a Google account
- Create a developer account
- Now, connect to the Google client

Google Fit REST API

- Enables you to store and access user data in the fitness store from apps on any platform
- Create, obtain, list and modify data sources
- Aggregate and delete datasets
- List data points and add them to the dataset
- Managing sessions
  - A session represents a time interval and associated metadata
Representational State Transfer (REST)

This material is built based on,

Deriving REST
:Architectural Constraints

Representational State Transfer (REST)
- An architectural style for networked hypermedia applications
- Used to build Web services that are lightweight, maintainable and scalable
- RESTful service
  - A service based on REST
- REST is not dependent on any protocol
  - But, almost every RESTful service uses HTTP as its underlying protocol

Starting with the Null Style
- The Null style is simply an empty set of constraints
  - No distinguished boundaries between components
  - Starting point of REST

Client-Server constraint
- Separation of concerns
  - Improve portability of the user interface
    - Across multiple platforms
  - Improve scalability
    - By simplifying the server components
Stateless constraint 1/2

- Communication must be stateless
- Each request from client to server
  - Must contain all of the information necessary to understand the request
  - Cannot take advantage of any stored context on the server
- Session state is kept at the client

Stateless constraint 2/2

- This constraint induces the properties of
  - Visibility
    - The monitoring system does not have to look beyond a single request
  - Reliability
    - Easy to recover from partial failures
  - Scalability
    - Server does not have to manage resources across requests

Disadvantage of the stateless constraint

- It may decrease network performance
  - Increasing the amount of repetitive data
- The servers have less control over consistent application behavior

Cache constraints 1/2

- Improves network efficiency
  - Requires that the data within the response to a request be implicitly or explicitly labeled as cacheable or non-cacheable

Cache constraints 2/2

- Reduces the average latency
  - They can eliminate partial or complete interactions
- May decrease reliability
  - Stale data

Uniform Interface Constraints

- Implementations are decoupled from the services
  - Independent evolvability
- Disadvantage
  - A uniform interface degrades efficiency for non-dominant cases
Layered System Constraints (1/2)
- Layered system style allows an architecture to be composed of hierarchical layers
- Each component cannot “see” beyond the immediate layer with which they are interacting

Layered System Constraints (2/2)
- Advantage
  - Encapsulating legacy services
  - Protecting new services from legacy clients
  - Simplifying components by moving infrequently used functionality to a shared intermediary
- Disadvantage
  - Adding overhead and latency

REST: Developing a REST Client

RESTful services
- REST is NOT a standard
- It uses components that are based on standards
  - HTTP
  - URL
  - XML/HTML/GIF/JPEC/etc (Resource Representation)
  - Text/xml, text/html, image/gif, etc (MIME Types)

Who are providing REST interfaces?
- Google Cloud Storage Service
- Google Search REST
- Netflix
- Twitter
- Flickr
- Amazon eCommerce
- Amazon S3
- ...

To be a REST client
- Endpoint

http://www.thomas-bayer.com/restnames/countries.groovy
Results (Using Safari)

4 major HTTP methods for REST CRUD

- Create, Read, Update, and Delete
  - POST – Update
  - GET – Read
  - PUT – Create
  - DELETE – Delete

When to use GET

- Caches depend on the ability to serve cached representations
  - Without contacting the origin server
- Safe and idempotent information retrieval

Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of \( N > 0 \) identical requests is the same as that for a single request.

GET example

```
#Bookmark a page
GET /bookmarks/add_bookmark?href=http://www.example.org/2009/10/10/notes.html HTTP/1.1
Host: www.example.org

# Add an item to a shopping cart
GET /add_cart?pid=1234 HTTP/1.1
Host: www.example.org

# Send a message
GET /message/send?message=I%20am%20reading HTTP/1.1
Host: www.example.org

# Delete a note
GET /notes/delete?id=1234 HTTP/1.1
Host: www.example.org
```
Designing a Web Service with GET

- If it is not safe to cache
  - Make the response noncacheable
    - Add a Cache-Control: no-cache header
- Consider any possible side effects
- Implement servers which can handle frequently repeatable operations (e.g., concurrent access)

When to use POST

- To create a new resource (sub-resource)
- To modify an existing resource
- To run a query with large inputs
- To perform any unsafe or non-idempotent operation (when no other HTTP method is available)

Creating Resources Using POST

- Submit a POST request with a representation of the resource to be created by the factory resource
- Optional Slug header
  - Name of the new resource suggested by clients

```
# Request
POST /user/smith HTTP/1.1
Host: www.example.org
Content-Type: application/xml; charset=UTF-8
Slug: Home Address

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```
POST Response

```xml
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address
Content-Location: http://www.example.org/user/smith/address/home_address
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/home_address"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

Creating Resources Using PUT

- PUT is for creating or replacing something with a known URL
- PUT is idempotent
- Use PUT to create new resources only when clients can decide URIs of resources
- Otherwise, use POST

In RFC of HTTP,

The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource. If the server desires that the request be applied to a different URI, it MUST send a 301 (Moved Permanently) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

Is PUT idempotent?

- Is DELETE idempotent?

Is PUT idempotent? -- Yes

- Is DELETE idempotent? -- Yes
What if there are two conflicting PUTs?
- HTTP/REST does not require "lock" for these concurrent access.
- REST is STATELESS.

## PUT request

```
# Request
PUT /user/smith/address/home_address HTTP/1.1
Host: www.example.org
Slug: Home Address

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

## PUT Response

```
# Response
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <adtom:link rel="self" href="http://www.example.org/user/smith/address/home_address"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

## POST example

```
# A SOAP message tunneled over HTTP POST
POST /Messages HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=UTF-8

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope" xmlns:ns="http://www.example.org/messages">
  <soap:Body>
    <ns:DeleteMessage>
      <ns:MessageId>1234</ns:MessageId>
    </ns:DeleteMessage>
  </soap:Body>
</soap:Envelope>
```

## RESTful Service Interface

```
DELETE
```

Is this safe?
DELETE

# Using DELETE
DELETE /message/1234 HTTP/1.1
Host: www.example.org

DELETE response
• The server creates a new resource and representation indicating the status of the job
• The client can query http://www.example.org/task/1 to learn the status of the request

HTTP/1.1 202 Accepted
Content-Type: application/xml;charset=UTF-8

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <status> pending </status>
  <atom:link href="http://www.example.org/task/1" rel="self"/>
  <message xml:lang="en"> Your request has been accepted for processing. </message>
  <created> 2009-07-05T03:10:00Z</created>
  <ping-after> 2009-07-05T03:15:00Z </ping-after>
</status>

RESTful Service Interface
Managing Errors

Description of Error
• Formatted and localized document (HTML or plain text) included in a body
• Except for the HEAD method
• Other details can be linked via a Link header or in the body
• Keep the body descriptive

Error Message
# Avoid returning success code with an error in the body.
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<error>
  <message> Account limit exceeded. </message>
</error>
Include your error code in the Header

- 400 Bad request
- 401 unauthorized
- 403 forbidden
- 404 not found
- 409 conflict
- 410 gone
- 412 precondition failed
- 413 request entity too large
- 415 unsupported media type

Provide description

- A brief message describing the error condition
- A longer description with information on how to fix, if applicable
- An identifier for the error
- A link to learn more about the error condition, with tips on how to resolve it

Example of a Good Error message

```xml
<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Account limit exceeded. We cannot complete the transfer due to insufficient funds in your accounts</message>
  <error-id>321-553-495</error-id>
  <account-from>urn:example:account:1234</account-from>
  <account-to>urn:example:account:5678</account-to>

  <atom:link href="http://example.org/rels/transfer/from" rel="http://example.org/rels/transfer/to"/>

</error>
```

Example of a Good Error message

```xml
<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Account limit exceeded. We cannot complete the transfer due to insufficient funds in your accounts</message>
  <error-id>321-553-495</error-id>
  <account-from>urn:example:account:1234</account-from>
  <account-to>urn:example:account:5678</account-to>

  <atom:link href="http://example.org/rels/transfer/from" rel="http://example.org/rels/transfer/to"/>

</error>
```