PART 3.
DATA STORAGE AND FLOW MANAGEMENT

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

Today’s topics
• FAQs
• Data Exchange Model
• RESTful service interface

FAQs
• Term project presentation
  - May 1 4-6PM CSB130
  - May 3 4-5:50PM CSB130
  - 10 minutes per team
    - Presentation
    - Q&A
    - Transition
• Send your slides 2hrs before the class starts
• Final Exam: May 11 7:30-9:30AM CSB130
  - Guidelines are available on the course web page
  - Quizzes (with answers) are posted

Wearable devices and sensors

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

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 associated with an app

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

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

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 resource 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

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/JPEG/etc (Resource Representation)
  - Text/xml, text/html, image/gif, etc (MIME Types)

To be a REST client

- Endpoint
  - https://yoda.p.mashape.com/yoda

Results (Using Safari)

4 major HTTP methods for REST CRUD

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

When to use GET

- Caches depend on the ability to serve cached representations
  - Without contacting the original server
- Safe and idempotent information retrieval
**GET example**

- Bookmark a page
  ```
  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)

**RESTful Service Interface**

**POST**

**When to use POST**

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

**Continued**

- Originally, POST was designed for
  - Annotation of existing resources
  - Posting on group articles
  - Creates a child resource
  - Providing append operations for database
  - E.g. Create a resource that lives under /items resource

- Unsafe and non-idempotent processing for the server

**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
POST request

# 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

# Response
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>
  <adtom: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 requests that the enclosed entity be stored under the supplied URI
- PUT is idempotent
- Use PUT to create/add 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
Content-Type: application/xml; charset=UTF-8
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-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 /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:soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  <soap:Body>
    <ns:DeleteMessage>
      <ns:MessageId>1234</ns:MessageId>
    </ns:DeleteMessage>
  </soap:Body>
</soap:Envelope>
## POST example

A SOAP message tunneled over HTTP POST

```plaintext
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
  soap:encodingStyle = http://www.w3c.org/2001/12/soap-encoding>
  <soap:Body xmlns:ns=http://www.example.org/messages>
    <ns:DeleteMessage>
      <ns:MessageId>1234</ns:MessageId>
    </ns:DeleteMessage>
  </soap:Body>
</soap:Envelope>
```

Is this safe?

## DELETE

Using DELETE

```plaintext
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](http://www.example.org/task/1) to learn the status of the request

```plaintext
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

How to Return Errors

- Error needs to be represented as well
- Errors in the clients' input
  - 4xx status code
- Error due to server implementation or current state
  - 5xx status code
- Include a Date header
  - The date-time at which the error occurred

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>

Is this a good Error response?
Error must be handled by software

# 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

Include your error code in the Header
- 500 Internal Server Error
- 503 Service Unavailable

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

# Response
HTTP/1.1 409 Conflict
Content-Type: application/xml;charset=UTF-8
Content-Language: en
Date: Wed, 14 Oct 2009 10:16:54 GMT
Link: <http://www.example.org/errors/limits.html>;rel="help"

<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>121-553-495</error-id>
  <account-from>urn:example:account:1234</account-from>
  <account-to>urn:example:account:5678</account-to>
  <atom:link href="http://example.org/account/1234" rel="http://example.org/rels/transfer/from/"/>
  <atom:link href="http://example.org/account/5678" rel="http://example.org/rels/transfer/to/"/>
</error>
Example of a Good Error

HTTP/1.1 409 Conflict
Content-Type: application/xml;charset=UTF-8
Content-Language: en
Date: Wed, 14 Oct 2009 10:16:54 GMT

<error xmlms: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/account/1234" rel="http://example.org/rels/transfer/from/">
  </atom:link>
  <atom:link href="http://example.org/account/5678" rel="http://example.org/rels/transfer/to/">
  </atom:link>
</error>

The Web server thinks that the request submitted by the client cannot be completed because it conflicts with some rule already established.

Description

Traceable ID

URI for Queries

- To select data from the available resources
- To specify a sort criteria
- To list the fields of resources to be included in the response

www.example.org/book/978-0374292881/reviews

www.example.org/book/978-0374292881/reviews?sortByDesc=created&limit=5

Query with Large Inputs

- HTTP does not impose any limits on the length of URIs
  - Explorer limits the length of URI to 2,083 characters.
  - The Apache web server limits the length of the request line to 8,190 bytes
- These limits may prevent you from encoding a large number of filter conditions into URIs
- Use POST

Caching your response with POST

- Reduce end user perceived latency
- Increase reliability
- Reduce bandwidth usage and cost
- Reduce server load

Example

# Request
POST /jobs HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded
Keywords=web,ajax,php&industry=software&experience=5&...

"Responses to POST method are not cacheable, UNLESS the response includes appropriate Cache-Control or Expires header fields." RFC 2616, section 9.5

Disadvantage:
- Using the POST method is a misuse of HTTP uniform interface.
- Loss of cacheability
Caching your response

- Caches can be anywhere
  - Server network
  - Content delivery networks (CDNs)
  - Client network (forward proxies)
- Object cache (e.g. memcached)
- HTTP cache (e.g. Squid or Traffic Server)

Specifying Expiration caching

- Expiration caching
  - To reduce the number of requests received by the origin server
  - To reduce the bandwidth used by application
- Cache-control and Expires headers
  - Instructs clients and caches to keep a copy of the representation returned by the server for a specific length of time.

Freshness lifetime

- Frequency of updates
  - Determines how long the caches can serve a representation
- Cache-Control header
  - max-age (in seconds)
- Expires header
  - The expiration date-time

Cache-Control

- Public
  - Default option
  - Shared caches
- Private
  - Response is private to the client or the user
  - The client-side cache (e.g. the browser cache or a forward proxy) can cache
  - Shared cache (e.g. server side or network) does not cache
- No-cache and no-store
  - Prevents any cache from storing or serving a cached response

Must-revalidate

- Requires caches to check the origin server before serving stale representation
- Proxy-revalidate
  - Similar to the must-revalidate directive except that it applies only to shared caches

e.g. POST body digest
Dealing with Expiration Headers in Clients

- In general, client should stay independent of expiration caching
- Building a client application that supports HTTP's caching protocol
- Common browsers store representations in memory or file system
- Developing this involves expiration directives
  - no-store, no-cache, must-revalidate, etc.
- Location for cache
  - e.g., sharing current runtime Vs. using a forward proxy cache

RESTful Service Interface
Using a RESTful service from your Web Server

Can we run a servlet under Tomcat?

- Download and install Tomcat
- Now, the host is localhost:8080
- When Tomcat sees a URL, it consults its server.xml
  ```xml
  <Context path="/examples" docBase="/webapps/examples" crossContext="false" debug="0" reloadable="true">
  </Context>
  ```
- The <Context> tag tells Tomcat to map
  ```http://localhost:8080/examples/```
  ```%TOMCAT_HOME%/webapps/examples/```
- Therefore, when a user asks for
  ```http://localhost:8080/examples/servlets/index.html```
  Tomcat will access the file
  ```%TOMCAT_HOME%/webapps/examples/servlets/index.html```

Register your servlet (web.xml)

```xml
<web-app>
  <servlet>
    <servlet-name>testServlet</servlet-name>
    <servlet-class>testServletPackage.testServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>testServlet</servlet-name>
    <url-pattern>/testServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Creating HTML tags as response to a GET request

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException{
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<BIG>Hello World</BIG>");
    out.println("</BODY></HTML>");
  }
}
```
Servlet interacting with a backend SQL server

```java
public class ResultFromSQLServlet extends HttpServlet {
    public void doGet (HttpServletRequest req,
            HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");
        ServletOutputStream out = res.getOutputStream();
        out.println("<html>);
        out.println("<head><title>Result from SQL server</title></head>"seud); 
        out.println("<body>"); 
        try {
            Class.forName("org.gjt.mm.mysql.Driver").newInstance();
            Connection Conn = 
            DriverManager.getConnection("jdbc:mysql://
                 sql.useractive.com/USERNAME?user=USERNAME&password=PASSWORD");
            Statement Stmt = Conn.createStatement();
            ResultSet RS = Stmt.executeQuery("SELECT * from SOMETABLE");
            while (RS.next()) {
                out.println(RS.getString(1));
            }
            RS.close();
            Stmt.close();
            Conn.close();
        }
        catch (Exception E) {
            out.println("Unable to load driver.");
            E.printStackTrace();
        } 
        out.println("<h1>Result for SQL server</h1>"
        out.println("</body"></html>");
    }
    public String getServletInfo() {
        return "Create a page that says <i>Result from SQL server</i> and send it back";
    }
}
```