Recitation – Week 4

JEONGMIN LEE
A little bit of HTTP
Client and Server model

- **Client**, like your computer or mobile phone, makes requests to specific server.
Client and Server model

- **Client**, like your computer or mobile phone, makes requests to specific server.
- **Server** gets the request and processes it including retrieving data and sends it back to client.
Client and Server model

- **Client**, like your computer or mobile phone, makes requests to specific server.
- **Server** gets the request and processes it including retrieving data and sends it back to client.
- Client receives the data and present it to user through proper program. (i.e., Mail program, Web browser)
Client and Server model

- **Client**, like your computer or mobile phone, makes requests to specific server.
- **Server** gets the request and processes it including retrieving data and sends it back to client.
- Client receives the data and present it to user through proper program. (i.e., Mail program, Web browser)
- The communication is running over the **Internet** with protocol(=HTTP).
Client side

- For specific service user needs, there exists several programs that handles communication with server.
  - Surfing the web: Web browser (Chrome, Safari, IE)
  - E-mail: Mail Client program (Outlook Express, Apple Mail)
  - FTP: FTP Clients (Filezilla, Cyberduck)
  - SSH: Putty
  - Torrent: U-Torrent
Server side

- For specific service, different programs running on server to serve client’s requests
  - Web: Flask/Django (python based), Apache, PHP, Nginx, etc..
  - Database: MySQL
  - FTP: Vsftpd

- These programs are always running on server. When clients’ request received, proper program processes user’s request.
HTTP Protocols

- GET
- POST
- PUT
- DELETE
GET

- Read (or retrieve) something.
- Used only to read data and not change it.
- In the “happy” (or non-error) path,
  GET returns a representation in XML or JSON and an HTTP response code of 200 (OK).
- In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

GET

- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests should be used only to retrieve data

GET

- **Length Restriction**
  - When sending data, the GET method adds the data to the URL;
  - The length of a URL is limited (maximum URL length is 2048 characters)

- **Data type Restriction:**
  Only ASCII characters allowed

- **Idempotent**
  - Making multiple identical requests ends up having the same result as a single request.

GET

- Examples:
  - GET http://www.ex.com/a/?name1=value1&name2=value
  - GET http://www.example.com/customers/12345/orders
  - GET http://www.example.com/buckets/sample

- Query string (name/value pairs) is sent in the URL of a GET request

POST

- The POST verb is most-often utilized to **create** new resources.
- In particular, it's used to create subordinate resources.
- That is, subordinate to some other (e.g. parent) resource.
- In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

POST

- On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.
- POST is neither safe nor idempotent.
- It is therefore recommended for non-idempotent resource requests.
- Making two identical POST requests will most-likely result in two resources containing the same information.

POST

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

- Query string (name/value pairs) is sent in the HTTP message body of a POST request:

  ```
  POST /test/demo_form.php HTTP/1.1
  Host: w3schools.com
  name1=value1&name2=value2
  ```

POST

- *Idempotent*, which means that making multiple identical requests ends up having the same result as a single request.

- Examples:
  - POST http://www.example.com/customers
  - POST http://www.example.com/customers/12345/orders
PUT

- PUT is most-often utilized for **update** capabilities, PUT-ing to a **known resource URI** with the request body containing the **newly-updated** representation of the original resource.

However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server.

In other words, if the PUT is to a URI that contains the value of a non-existent resource ID.

Again, the request body contains a resource representation.

Many feel this is convoluted and confusing.

PUT

- On successful **update**, return **200** (or 204 if not returning any content in the body) from a PUT.
- If using PUT for **create**, return HTTP status **201** on successful creation.
- A body in the response is optional—providing one consumes more bandwidth.

PUT

- **PUT is not a safe operation**, in that it **modifies** (or creates) state on the server.

- It is **idempotent**. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

PATCH

- PATCH is used for **modify** capabilities.
- The PATCH request only needs to contain the changes to the resource, not the complete resource.
- PATCH is neither safe nor idempotent.

DELETE

- DELETE is pretty easy to understand. It is used to **delete** a resource identified by a URI.

- On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below).

- Either that or return HTTP status 204 (NO CONTENT) with no response body.

DELETE

- HTTP-spec-wise, DELETE operations are idempotent.

- If you DELETE a resource, it's removed.

- Repeatedly calling DELETE on that resource ends up the same: the resource is gone.

DELETE

- If calling DELETE, say, decrements a counter (within the resource), the DELETE call is no longer idempotent.

- Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable.