Pre-reading

Week 6 – Fetching data

URI

A **URI** *(Uniform Resource Identifier)* is a string that refers to a resource. The most common are URLs, which identify the resource by giving its location on the Web. URNs, by contrast, refer to a resource by a name, in a given namespace, such as the ISBN of a book.

URL

A **URL** (*Uniform Resource Locator*) is a text string that specifies where a resource (such as a web page, image, or video) can be found on the Internet.

URN

A **URN** (*Uniform Resource Name*) is a URI in a standard format, referring to a resource without specifying its location or whether it exists.

Example: urn:oasis:names:specification:docbook:dtd:xml:4.1.2

Idempotent

An HTTP method is idempotent if an identical request can be made once or several times in a row with the same effect while leaving the server in the same state. In other words, an idempotent method should not have any side-effects (except for keeping statistics).

Real-world example: “Sweep the floor”

- If the floor is **dirty** and you correctly sweep the floor, the floor will end up **clean**.
- If the floor is already **clean**, sweeping the floor will do **nothing**.
- Sweeping the floor is an **idempotent** action.

Safe

An HTTP method is safe if it doesn't alter the state of the server. In other words, a method is safe if it leads to a read-only operation. All safe methods are idempotent, but not all idempotent methods are safe. For example, PUT and DELETE are both idempotent but unsafe.

Even if safe methods have a read-only semantic, servers can alter their state: e.g. they can log or keep statistics. What is important here is that by calling a safe method, the client doesn't request any server change itself, and therefore won't create an unnecessary load or burden for the server. Browsers can call safe methods without fearing to cause any harm to the server; this allows them to perform activities like pre-fetching without risk. Web crawlers also rely on calling safe methods.

Safe methods don't need to serve static files only; a server can generate an answer to a safe method on-the-fly, as long as the generating script guarantees safety: it should not trigger external effects, like triggering an order in an e-commerce Web site.

It is the responsibility of the application on the server to implement the safe semantic correctly, the webserver itself, being Apache, Nginx or IIS, can't enforce it by itself. In particular, an application should not allow GET requests to alter its state.

Cacheable

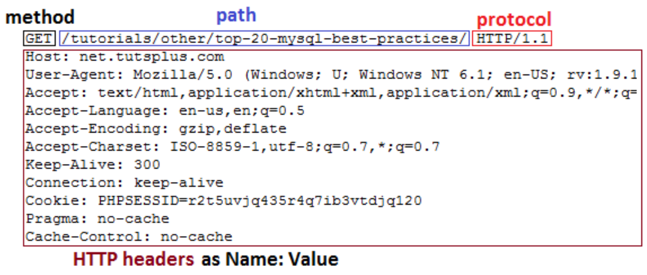
A cacheable response is an HTTP response that can be cached, that is stored to be retrieved and used later, saving a new request to the server.

There are specific headers in the response, like Cache-Control, that prevents caching.

Note that some non-cacheable requests/responses to a specific URI may invalidate previously cached responses on the same URI. For example, a PUT to pageX.html will invalidate all cached GET or HEAD requests to the same URI.

HTTP Headers

HTTP headers let the client and the server pass additional information with an HTTP request or response. An HTTP header consists of its case-insensitive name followed by a colon (:), then by its value.



```
GET /tutorials/other/top-20-mysql-best-practices/ HTTP/1.1
Host: net.tutsplus.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

HTTP Status Codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirects (300–399)
- Client errors (400–499)
- Server errors (500–599)

Read more: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

HTTP Request Methods

HTTP defines a set of **request methods** to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as *HTTP verbs*. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be safe, idempotent, or cacheable.

GET

The HTTP GET method requests a representation of the specified resource. Requests using GET should only be used to request data (they shouldn't include data).

Note: Sending body/payload in a GET request may cause some existing implementations to reject the request — while not prohibited by the specification, the semantics are undefined. It is better to just avoid sending payloads in GET requests.

Syntax: `GET /index.html`

HEAD

The HTTP HEAD method requests the headers that would be returned if the HEAD request's URL was instead requested with the HTTP GET method. For example, if a URL might produce a large download, a HEAD request could read its Content-Length header to check the filesize without actually downloading the file.

A response to a HEAD method should not have a body. If it has one anyway, that body must be ignored: any entity headers that might describe the erroneous body are instead assumed to describe the response which a similar GET request would have received.

If the response to a HEAD request shows that a cached URL response is now outdated, the cached copy is invalidated even if no GET request was made.

Syntax: `HEAD /index.html`

POST

The HTTP POST method sends data to the server. The type of the body of the request is indicated by the Content-Type header.

The difference between PUT and POST is that PUT is idempotent: calling it once or several times successively has the same effect (that is no side effect), where successive identical POST may have additional effects, like passing an order several times.

A POST request is typically sent via an HTML form and results in a change on the server.

When the POST request is sent via a method other than an HTML form — like via an XMLHttpRequest — the body can take any type. As described in the HTTP 1.1 specification, POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Adding a new user through a signup modal;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.

POST Example Request

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

field1=value1&field2=value2
```

PUT

The HTTP PUT request method creates a new resource or replaces a representation of the target resource with the request payload.

The difference between PUT and POST is that PUT is idempotent: calling it once or several times successively has the same effect (that is no side effect), whereas successive identical POST requests may have additional effects, akin to placing an order several times.

PUT Example Request

```
PUT /new.html HTTP/1.1
Host: example.com
Content-type: text/html
Content-length: 16

<p>New File</p>
```

PUT Example Responses

If the target resource does not have a current representation and the PUT request successfully creates one, then the origin server must inform the user agent by sending a 201 (Created) response.

```
HTTP/1.1 201 Created
Content-Location: /new.html
```

If the target resource does have a current representation and that representation is successfully modified in accordance with the state of the enclosed representation, then the origin server must send either a 200 (OK) or a 204 (No Content) response to indicate successful completion of the request.

```
HTTP/1.1 204 No Content
Content-Location: /existing.html
```

DELETE

The HTTP DELETE request method deletes the specified resource.

Example request:

```
DELETE /file.html HTTP/1.1
```

If a DELETE method is successfully applied, there are several response status codes possible:

- A 202 (Accepted) status code if the action will likely succeed but has not yet been enacted.
- A 204 (No Content) status code if the action has been enacted and no further information is to be supplied.
- A 200 (OK) status code if the action has been enacted and the response message includes a representation describing the status.

CONNECT

The HTTP CONNECT method starts two-way communications with the requested resource. It can be used to open a tunnel.

For example, the CONNECT method can be used to access websites that use SSL (HTTPS). The client asks an HTTP Proxy server to tunnel the TCP connection to the desired destination. The server then proceeds to make the connection on behalf of the client. Once the connection has been established by the server, the Proxy server continues to proxy the TCP stream to and from the client.

OPTIONS

The HTTP OPTIONS method requests permitted communication options for a given URL or server. A client can specify a URL with this method, or an asterisk (*) to refer to the entire server.

Example request:

```
OPTIONS /index.html HTTP/1.1
OPTIONS * HTTP/1.1
```

Example response:

```
HTTP/1.1 204 No Content
Allow: OPTIONS, GET, HEAD, POST
Cache-Control: max-age=604800
Date: Thu, 13 Oct 2016 11:45:00 GMT
```

PATCH

The HTTP PATCH request method applies partial modifications to a resource.

PATCH is somewhat analogous to the "update" concept found in CRUD (in general, HTTP is different than CRUD, and the two should not be confused).

A PATCH request is considered a set of instructions on how to modify a resource. Contrast this with PUT; which is a complete representation of a resource.

A PATCH is not necessarily idempotent.

PATCH (like POST) may have side-effects on other resources.

To find out whether a server supports PATCH, a server can advertise its support by adding it to the list in the Allow or Access-Control-Allow-Methods (for CORS) response headers.

Another (implicit) indication that PATCH is allowed, is the presence of the Accept-Patch header, which specifies the patch document formats accepted by the server.

PATCH Example Request

```
PATCH /file.txt HTTP/1.1
Host: www.example.com
Content-Type: application/example
If-Match: "e0023aa4e"
Content-Length: 100
```

```
[
  { "op": "replace", "path": "/name", "value": "Iman Tumorang" },
  { "op": "add", "path": "/likes", "value": ["go", "blogging"] }
]
```

Description of changes

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources. CORS also relies on a mechanism by which browsers make a “preflight” request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

An example of a cross-origin request: the front-end JavaScript code served from https://oranges.com uses `XMLHttpRequest` to make a request for https://bananas.com/data.json.

For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, `XMLHttpRequest` and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.

The CORS mechanism supports secure cross-origin requests and data transfers between browsers and servers. Modern browsers use CORS in APIs such as XMLHttpRequest or Fetch to mitigate the risks of cross-origin HTTP requests.