Unit 4: HTTP

Contents

HTTP:

Application level protocol that defines how a web client talks to a server and how data is transferred from the server back to the client.

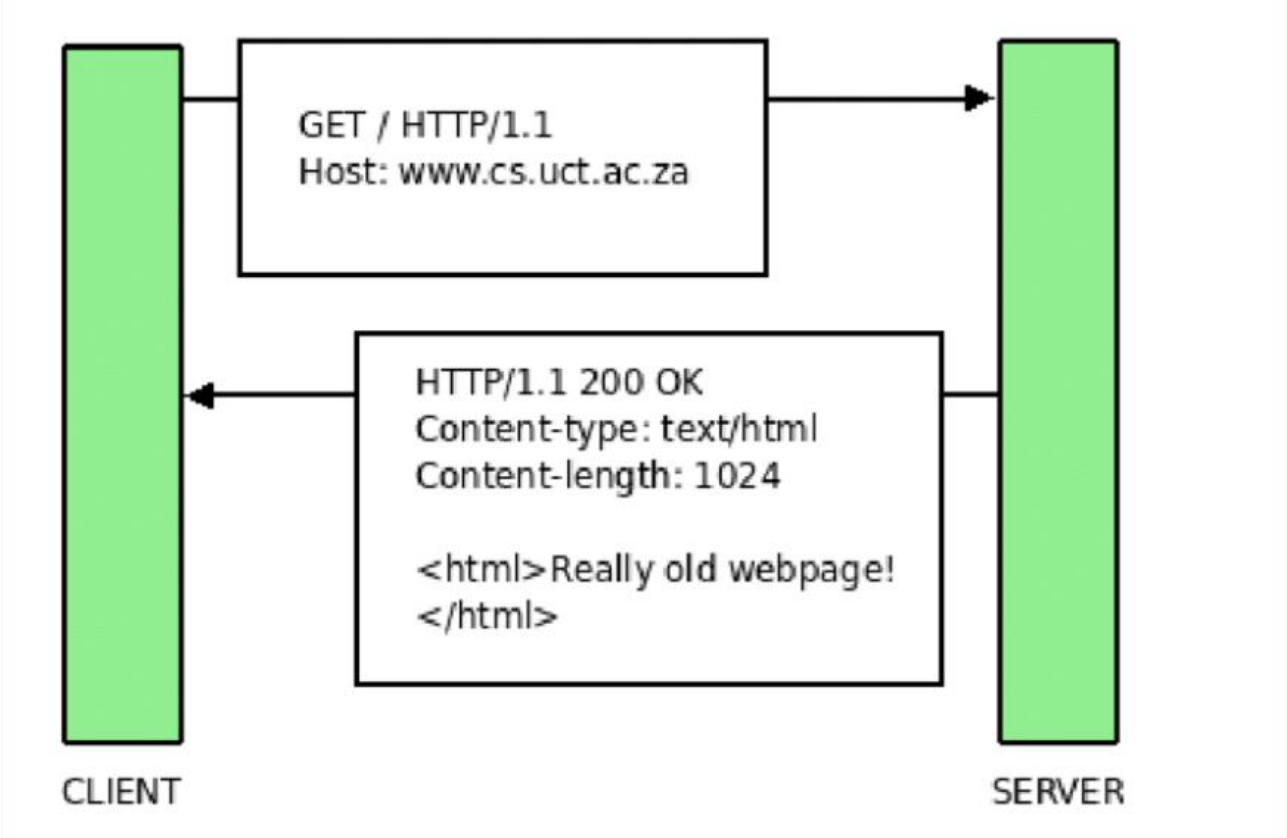


Fig: Client-Server Communication Over HTTP

HTTP is usually thought of as a means of transferring HTML files and the pictures embedded in them.

It can be used to transfer pictures, Microsoft Word documents, Windows.exe files, or anything else that can be represented in bytes.

The Protocol:

HTTP is the standard protocol for communication between web browsers and web servers

- HTTP specifies how a client and server establish a connection,
- how the client requests data from the server,
- how the server responds to that request, and
- finally, how the connection is closed

For each request from client to server, there is a sequence of four steps:

1. Client opens a TCP connection to the server on port 80, by default; other ports may be specified in the URL
2. Client sends a message to the server requesting the resource at a specified path.
3. Server sends a response to the client.
4. The server closes the connection.

A typical client request :

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:20.0)
Gecko/20100101 Firefox/20.0
Host: en.wikipedia.org
Connection: keep-alive
Accept-Language: en-US,en;
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml
```

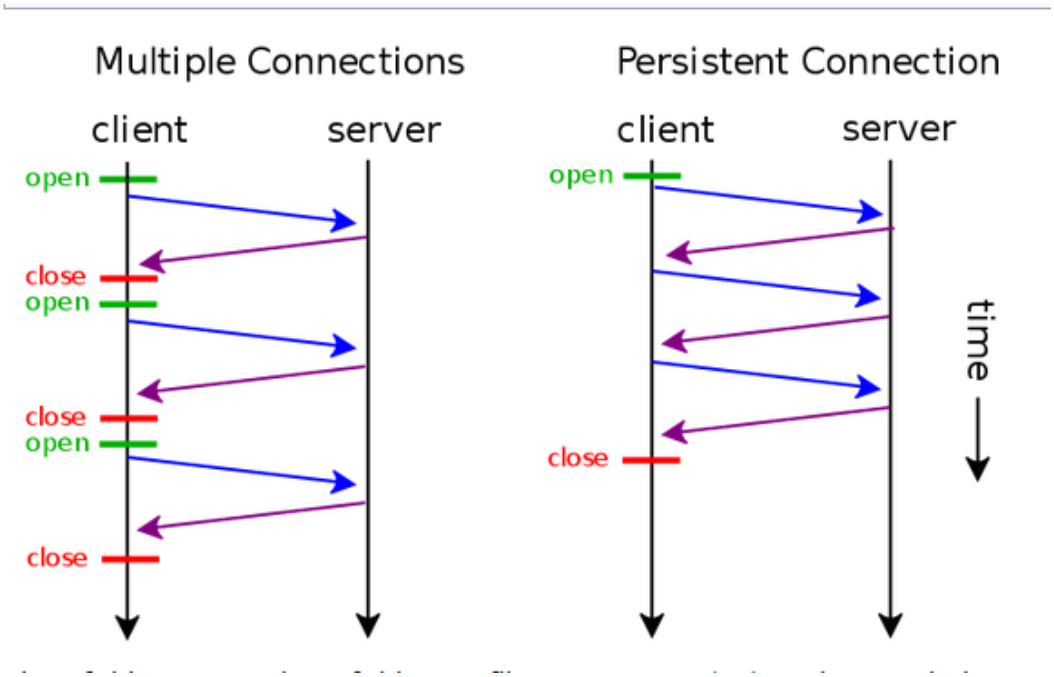
A typical successful response :

HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-length: 115

HTTP: Keep-Alive/Persistent Connection:

HTTP persistent connection, also called HTTP keep-alive, or HTTP connection reuse, is the idea of using a single TCP connection to send and receive multiple HTTP requests/responses, as opposed to opening a new connection for every single request/response pair.

Fig:



Multiple vs Persistent Connection

HTTP 1.0 opens a new connection for each request. The time taken to open and close all the connections in a typical web sessioncan outweigh the time taken to transmit the data, especially for sessions with many small documents

This is a problematic for encrypted HTTPS connections using Secure Sockets Layer & Transport Layer Security

In HTTP 1.1 and later, the server doesn't have to close the socket after it sends its response

A client indicates that it's willing to reuse a socket by including a Connection field in the HTTP request header with the value Keep-Alive: Connection: Keep-Alive

Java http package Properties for HTTP Keep-Alive:

We can control Java's use of HTTP Keep-Alive with several system properties:

• http.keepAlive to "true or false" to enable/disable. Default enable.

• http.maxConnections to the number of sockets. The default is 5.

• http.keepAlive.remainingData to true, It is false by default.

• sun.net.http.errorstream.enableBuffering to true, It is false by default.

• sun.net.http.errorstream.bufferSize, The default is 4,096 bytes.

• sun.net.http.errorstream.timeout , It is 300 milliseconds by default.

HTTP Methods:

There are different methods that are commonly used to send request message to a HTTP server.

Each HTTP request has two or three parts:

• Start line containing the HTTP method and a path to the resource.

• Header of name-value fields that provide meta-information.

• Request body containing a representation of a resource (optional) → present only if a new resource is to be created or updated. (POST, PUT/PATCH)

1. GET: The GET method is used to retrieve a resource from the server. The resource can be a web page, an image, or any other type of file. The GET method should only be used to retrieve data and should not modify any data on the server.
2. POST: The POST method is used to submit data to the server, typically to create or update a resource. For example, a POST request can be used to submit a form on a website or to upload a file to the server.
3. PUT: The PUT method is used to update a resource on the server. The entire resource is replaced with the new data sent in the request. If the resource does not exist, the server can create it.

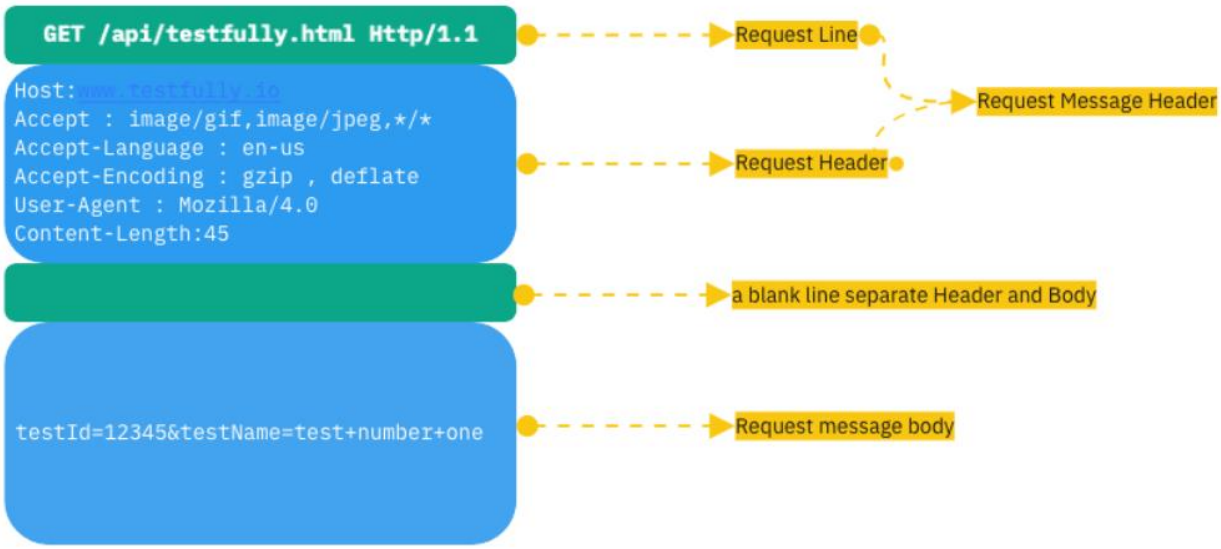
4. DELETE: The DELETE method is used to delete a resource from the server. The resource is identified by its URL, and once deleted, it cannot be restored.
5. HEAD: The HEAD method is similar to the GET method, but it only retrieves the headers of a resource, without the body. This can be useful to retrieve metadata about a resource without downloading the entire resource.
6. OPTIONS: The OPTIONS method is used to retrieve information about the communication options available for a resource. The OPTIONS method can be useful in situations where a client needs to determine which HTTP methods are supported by a server before attempting to make a request using one of those methods.
7. PATCH: The PATCH method is used to update a resource on the server with a partial update, rather than a full replacement. The request body contains a set of instructions for modifying the resource.
8. CONNECT: The CONNECT method is used to establish a network connection to a server using the HTTP protocol, typically for use with SSL (Secure Sockets Layer) or TLS (Transport Layer Security) encryption.
9. TRACE: The TRACE method is used to retrieve a diagnostic trace of the HTTP request and response

messages, which can be useful for debugging and troubleshooting.

There are other methods as well such as MOVE, COPY but Java does not support these.

- Safe Methods: These are a list of HTTP methods which does not send update,create or remove request to server. e.g. GET, OPTION, HEAD, TRACE
- Idempotent Methods: Methods that can be repeated (with same data), and causes no error. e.g. GET, PUT, PATCH, DELETE, HEAD, TRACE, OPTION.

HTTP General Syntax:



Note: Here, the request is for GET request, replace GET keyword with relevant method.

HTTP status codes

S.N.	Code and Description
1	**1xx: Informational** It means the request was received and the process is continuing.
2	**2xx: Success** It means the action was successfully received, understood, and accepted.
3	**3xx: Redirection** It means further action must be taken in order to complete the request.
4	**4xx: Client Error** It means the request contains incorrect syntax or cannot be fulfilled.
5	**5xx: Server Error** It means the server failed to fulfill an apparently valid request.

Here are some of the most commonly encountered HTTP status codes:

- 200 OK: The request was successful, and the server is returning the requested data.
- 201 Created: The request was successful, and the server has created a new resource as a result.
- 204 No Content: The request was successful, but there is no data to return.
- 304 Not Modified: The requested resource has not been modified since the last time it was requested.
- 400 Bad Request: The request contains invalid syntax or cannot be fulfilled by the server.

- 401 Unauthorized: The request requires authentication, and the client has not provided valid authentication credentials.
- 404 Not Found: The requested resource could not be found on the server.
- 500 Internal Server Error: The server encountered an error while processing the request.
- 503 Service Unavailable: The server is currently unable to handle the request due to a temporary overload or maintenance.

HTTP Request Body

- The GET method retrieves a representation of a resource identified by a URL.

- The exact location of the resource we want to GET from a server is specified by the various parts of the path and query string.

- POST and PUT are more complex. In these cases, the client supplies the representation of the resource, in addition to the path and the query string.

- The representation of the resource is sent in the body of the request, after the header. That is, it sends these four

items in order:

1. A starter line including the method, path and query string, and HTTP version A
2. An HTTP header
3. A blank line
4. The body

For example, this POST request sends form data to a server:

```
POST /cgi-bin/register.pl HTTP 1.0
Date: Sun, 27 Apr 2013 12:32:36
Host: www.cafeaulait.org
Content-type: application/x-www-form-urlencoded
Content-length: 54

username=Elliotte+Harold&email=elharo%40ibiblio.org
```

However, the HTTP header should include two fields that specify the nature of the body

• A Content-length field that specifies how many bytes are in the body.
• A Content-type field that specifies the MIME (Multipurpose Internet Mail Extensions) media type of the byte

For example, here’s a PUT request that uploads an Atom document:

PUT /blog/software-development/the-power-of-pomodoros/ HTTP/1.1
Host: elharo.com
User-Agent: AtomMaker/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA== (This is a Base64-encoded username and password used for authentication.)
Content-Type: application/atom+xml;type=entry
Content-Length: 322

Cookies

- small strings of text known as cookies to store persistent client-side state between connections.
- Cookies are passed from server to client and back again in the HTTP headers of requests and responses.
- Cookies are limited to non-whitespace ASCII text, and may not contain commas or semicolons.
- To set a cookie in a browser, the server includes a Set-Cookie header line in the HTTP header.
- For example, this HTTP header sets the cookie “cart” to the value “ATVPDKIKX0DER”:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: cart=ATVPDKIKX0DER
```

- If a browser makes a second request to the same server, it will send the cookie back in a Cookie line in the HTTP request header like so:

```
GET /index.html HTTP/1.1
Host: www.example.org
```

Cookie: cart=ATVPDKIKX0DER

Accept: text/html

- As long as the server doesn’t reuse cookies, this enables it to track individual users and sessions across multiple, otherwise stateless, HTTP connections. Servers can set more than one cookie.
- For example, a request made to Amazon fed my browser five cookies:

  Set-Cookie:skin=noskin

  Set-Cookie:ubid-main=176-5578236-9590213

  Set-Cookie:session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6Ktr

  Set-Cookie:session-id-time=2082787201l

  Set-Cookie:session-id=187-4969589-3049309
- In addition to a simple name=value pair, cookies can have several attributes that control their scope including:
  - expiration date,
  - path,
  - domain,
  - port,

- version, and
- security options

- For example, this request sets a user cookie for the entire foo.example.com
  domain:

  Set-Cookie: user=elharo;Domain=.foo.example.com

  Set-Cookie: user=elharo; Path=/restricted

  Set-Cookie:
  user=elharo;Path=/restricted;Domain=.example.com

  Cookie: user=elharo;
  Path=/restricted;Domain=.foo.example.com

  Set-Cookie: user=elharo; expires=Wed, 21-Dec-2015
  15:23:00 GMT

  Set-Cookie: user="elharo"; Max-Age=3600

  Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure

  Set-Cookie: key=etrogl7*;Domain=.foo.example.com;
  secure; httponly

Cookie Manager:

- Java 5 includes an abstract java.net.CookieHandler class that defines an API for storing and retrieving cookies.

- Java 6 has a concrete java.net.CookieManager subclass of CookieHandler can be use.
- Before Java will store and return cookies, you need to enable it:

  CookieManager manager = new CookieManager();
  CookieHandler.setDefault(manager);
- Three policies are predefined:
  - CookiePolicy.ACCEPT_ALL: All cookies allowed
  - CookiePolicy.ACCEPT_NONE: No cookies allowed
  - CookiePolicy.ACCEPT_ORIGINAL_SERVER: Only first party cookies allowed (i.e., the server that the client is directly communicating with).

- For example, this code fragment tells Java to block third-party cookies but accept first-party cookies:

```
CookieManager manager = new CookieManager();
manager.setCookiePolicy(CookiePolicy.ACCEPT_ORIGINAL_SERVER);
CookieHandler.setDefault(manager);
```

- you can implement the CookiePolicy interface yourself and override the shouldAccept() method:

```
public boolean shouldAccept(URI uri, HttpCookie cookie)
```

Example. A cookie policy that blocks all .gov cookies but allows others

```
import java.net.*;
public class NoGovernmentCookies implements
CookiePolicy {
@Override
public boolean shouldAccept(URI uri, HttpCookie cookie)
```

```
{
if (uri.getAuthority().toLowerCase().endsWith(".gov")
|| cookie.getDomain().toLowerCase().endsWith(".gov")) {
return false;
}
return true;
}
}
```

Cookie Store:

It is sometimes necessary to put and get cookies locally. For instance, when an application quits, it can save the cookie store to disk and load those cookies again when it next starts up

You can retrieve the store in which the CookieManager saves its cookies with the getCookieStore() method:

```
CookieStore store = manager.getCookieStore();
```

The CookieStore class allows you to add, remove, and list cookies so you can control the cookies that are sent outside the normal flow of HTTP requests and responses:

1. public void add(URI uri, HttpCookie cookie) : Adds a cookie for the given URI.
2. public List<HttpCookie> get(URI uri) : Returns a list of cookies for the given URI.
3. public List<HttpCookie> getCookies() : Returns a list of all cookies in the store.
4. public List<URI> getURIs(): Returns a list of URIs to which cookies have been added.
5. public boolean remove(URI uri, HttpCookie cookie) : Removes the specified cookie for the given URI.
6. public boolean removeAll(): Removes all cookies from the store.