HTTP - QUICK GUIDE

The Hypertext Transfer Protocol *HTTP* is an application-level protocol for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web *ie. internet* since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extension of its request methods, error codes and headers.

Basically, HTTP is an TCP/IP based communication protocol, which is used to deliver data *HTMLfiles, imagefiles, queryresultsetc* on the World Wide Web. The default port is TCP 80, but other ports can be used. It provides a standardized way for computers to communicate with each other. HTTP specification specifies how clients request data will be constructed and sent to the serve, and how servers respond to these requests.

Basic Features

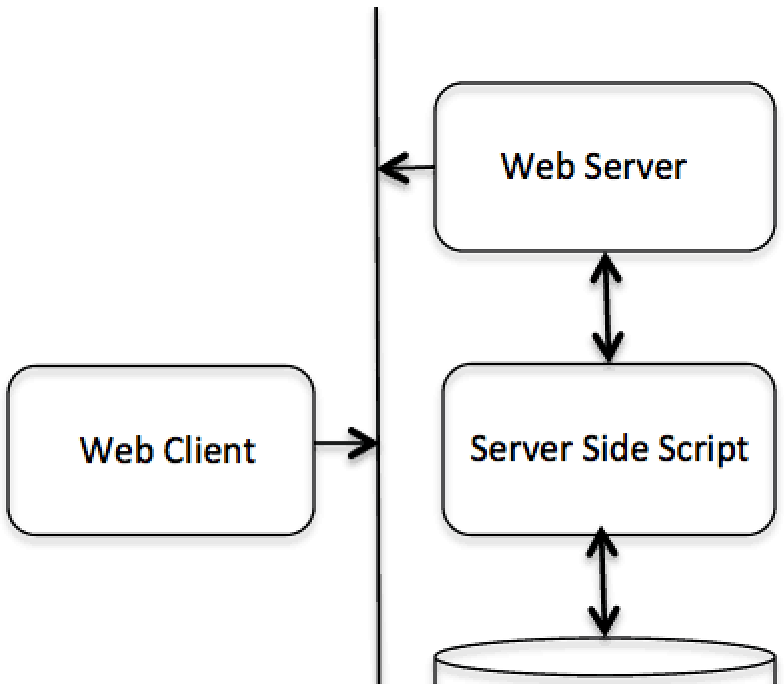
There are following three basic features which makes HTTP a simple but powerful protocol:

- **HTTP is connectionless:** The HTTP client ie. browser initiates an HTTP request and after a request is made, the client disconnects from the server and waits for a response. The server process the request and re-establish the connection with the client to send response back.
- **HTTP is media independent:** This means, any type of data can be sent by HTTP as long as both the client and server know how to handle the data content. This is required for client as well as server to specify the content type using appropriate MIME-type.
- **HTTP is stateless:** As mentioned above, HTTP is a connectionless and this is a direct result that HTTP is a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each other. Due to this nature of the protocol, neither the client nor the browser can retain information between different request across the web pages.

> *HTTP/1.0 uses a new connection for each request/response exchange where as HTTP/1.1 connection may be used for one or more request/response exchanges.*

Basic Architecture

Following diagram shows a very basic architecture of a web application and depicts where HTTP sits:



HTTP Protocol

The HTTP protocol is a request/response protocol based on client/server based architecture where web browser, robots and search engines, etc. act like HTTP clients and Web server acts as server.

Client

The HTTP client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a TCP/IP connection.

Server

The HTTP server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.

HTTP - PARAMETERS

This chapter is going to list down few of the important HTTP Protocol Parameters and their syntax in a way they are used in the communication. For example, format for date, format of URL etc. This will help you in constructing your request and response messages while writing HTTP client or server programs. You will see complete usage of these parameters in subsequent chapters while explaining message structure for HTTP requests and responses.

HTTP Version

HTTP uses a **<major>.<minor>** numbering scheme to indicate versions of the protocol. The version of an HTTP message is indicated by an HTTP-Version field in the first line. Here is the general syntax of specifying HTTP version number:

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

Example

```
HTTP/1.0

or

HTTP/1.1
```

Uniform Resource Identifiers *URI*

Uniform Resource Identifiers *URI* is simply formatted, case-insensitive string containing name, location etc to identify a resource, for example a website, a web service etc. A general syntax of URI used for HTTP is as follows:

```
URI = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

Here if the **port** is empty or not given, port 80 is assumed for HTTP and an empty **abs_path** is equivalent to an **abs_path** of "/". The characters other than those in the **reserved** and **unsafe** sets are equivalent to their ""%" HEX HEX" encoding.

Example

Following two URIs are equivalent:

```
http://abc.com:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7esmith/home.html
```

Date/Time Formats

All HTTP date/time stamps MUST be represented in Greenwich Mean Time *GMT*, without exception. HTTP applications are allowed to use any of the following three representations of date/time stamps:

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime() format
```

Character Sets

You use character set to specify the character sets that the client prefers. Multiple character sets can be listed separated by commas. If a value is not specified, the default is US-ASCII.

Example

Following are valid character sets:

```
US-ASCII

or

ISO-8859-1

or

ISO-8859-7
```

Content Encodings

A content ecoding values indicate an encoding algorithm has been used to encode the content before passing it over the network. Content codings are primarily used to allow a document to be compressed or otherwise usefully transformed without losing the identity.

All content-coding values are case-insensitive. HTTP/1.1 uses content-coding values in the Accept-Encoding and Content-Encoding header fields which we will see in subsequent chapters.

Example

Following are valid encoding schemes:

```
Accept-encoding: gzip

or

Accept-encoding: compress

or

Accept-encoding: deflate
```

Media Types

HTTP uses Internet Media Types in the **Content-Type** and **Accept** header fields in order to provide open and extensible data typing and type negotiation. All the Media-type values are registered with the Internet Assigned Number Authority (*IANA*. Following is a general syntax to specify media type:

```
media-type     = type "/" subtype *( ";" parameter )
```

The type, subtype, and parameter attribute names are case- insensitive.

Example

```
Accept: image/gif
```

Language Tags

HTTP uses language tags within the **Accept-Language** and **Content-Language** fields. A language tag is composed of 1 or more parts: A primary language tag and a possibly empty series of subtags:

```
language-tag  = primary-tag *( "-" subtag )
```

White space is not allowed within the tag and all tags are case- insensitive.

Example

Example tags include:

```
 en, en-US, en-cockney, i-cherokee, x-pig-latin
```

Where any two-letter primary-tag is an ISO-639 language abbreviation and any two-letter initial subtag is an ISO-3166 country code.

HTTP - MESSAGES

HTTP is based on client-server architecture model and a stateless request/response protocol that operates by exchanging messages across a reliable TCP/IP connection.

An HTTP "client" is a program *Webbrowseroranyotherclient* that establishes a connection to a server for the purpose of sending one or more HTTP request messages. An HTTP "server" is a program *generallyawebserverlikeApacheWebServerorInternetInformationServicesIISetc.* that accepts connections in order to serve HTTP requests by sending HTTP response messages.

HTTP makes use of the Uniform Resource Identifier *URI* to identify a given resource and to establish a connection. Once connection is established, **HTTP messages** are passed in a format similar to that used by Internet mail [RFC5322] and the Multipurpose Internet Mail Extensions *MIME* [RFC2045]. These messages are consisted of **requests** from client to server and **responses** from server to client which will have following format:

```
 HTTP-message   = <Request> | <Response> ; HTTP/1.1 messages
```

HTTP request and HTTP response use a generic message format of RFC 822 for transferring the required data. This generic message format consists of following four items.

```
• A Start-line

• Zero or more header fields followed by CRLF

• An empty line (i.e., a line with nothing preceding the CRLF) indicating the end of
  the header fields

• Optionally a message-body
```

Following section will explain each of the entities used in HTTP message.

Message Start-Line

A start-line will have following generic syntax:

```
start-line = Request-Line | Status-Line
```

We will discuss Request-Line and Status-Line while discussing HTTP Request and HTTP Response messages respectively. For now let's see the examples of start line in case of request and response:

```
GET /hello.htm HTTP/1.1     (This is Request-Line sent by the client)

HTTP/1.1 200 OK              (This is Status-Line sent by the server)
```

Header Fields

HTTP deader fields provide required information about the request or response, or about the object sent in the message body. There are following four types of HTTP message headers:

- **General-header:** These header fields have general applicability for both request and response messages.
- **Request-header:** These header fields are applicability only for request messages.
- **Response-header:** These header fields are applicability only for response messages.
- **Entity-header:** These header fields define metainformation about the entity-body or, if no body is present

All the above mentioned headers follow the same generic format and each of the header field consists of a name followed by a colon (**:**) and the field value as follows:

```
message-header = field-name ":" [ field-value ]
```

Following are the examples of various header fields:

```
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Message Body

The message body part is optional for an HTTP message but if it is available then it is used to carry the entity-body associated with the request or response. If entity body is associated then usually **Content-Type** and **Content-Length** headers lines specify the nature of the body associated.

A message body is the one which carries actual HTTP request data *includingformdataanduploadedetc*. and HTTP response data from the server *includingfiles, imagesetc*. Following is a simple content of a message body:

```
<html>
```

```
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

HTTP - REQUESTS

An HTTP client sends an HTTP request to a server in the form of a request message which includes following format:

```
• A Request-line

• Zero or more header (General|Request|Entity) fields followed by CRLF

• An empty line (i.e., a line with nothing preceding the CRLF) indicating the end of
  the header fields

• Optionally a message-body
```

Following section will explain each of the entities used in HTTP message.

Message Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by space SP characters.

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Let's discuss each of the part mentioned in Request-Line.

Request Method

The request **Method** indicates the method to be performed on the resource identified by the given **Request-URI**. The method is case-sensitive ans should always be mentioned uppercase. Following are supported methods in HTTP/1.1

S.N.	Method and Description
1	**GET** The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
2	**HEAD** Same as GET, but only transfer the status line and header section.
3	**POST** A POST request is used to send data to the server, for example customer information, file upload etc using HTML forms.
4	**PUT** Replace all current representations of the target resource with the uploaded content.
5	**DELETE** Remove all current representations of the target resource given by URI.

6	**CONNECT** Establish a tunnel to the server identified by a given URI.
7	**OPTIONS** Describe the communication options for the target resource.
8	**TRACE** Perform a message loop-back test along the path to the target resource.

Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request. Following are the most commonly used forms to specify an URI:

```
Request-URI = "*" | absoluteURI | abs_path | authority
```

S.N.	Method and Description
1	The asterisk ***** is used when HTTP request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. For example: **OPTIONS * HTTP/1.1**
2	The **absoluteURI** is used when HTTP request is being made to a proxy. The proxy is requested to forward the request or service it from a valid cache, and return the response. For example: **GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1**
3	The most common form of Request-URI is that used to identify a resource on an origin server or gateway. For example, a client wishing to retrieve the resource above directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the lines: **GET /pub/WWW/TheProject.html HTTP/1.1** **Host: www.w3.org** Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" *theserverroot*

Request Header Fields

We will study General-header and Entity-header in a separate chapter when we will learn HTTP header fields. For now let's check what are Request header fields.

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers and there are following important Request-header fields available which can be used based on requirement.

- Accept-Charset
- Accept-Encoding
- Accept-Language
- Authorization
- Expect

- From
- Host
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Proxy-Authorization
- Range
- Referer
- TE
- User-Agent

You can introduce your custom fields in case you are going to write your own custom Client and Web Server.

Request Message Examples

Now let's put it all together to form an HTTP request to fetch **hello.htm** page from the web server running on tutorialspoint.com

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Here we are not sending any request data to the server because we are fetching a plan HTML page from the server. Connection is a general-header used here and rest of the headers are request headers. Following is one more example where we send form data to the server using request message body:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string
```

Here given URL */cgi-bin/process.cgi* will be used to process the passed data and accordingly a response will be retuned. Here **content-type** tells the server that passed data is simple web form data and **length** will be actual length of the data put in the message body. Following example shows how you can pass plan XML to your web server:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
```

```
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://clearforest.com/">string</string>
```

HTTP - RESPONSES

After receiving and interpreting a request message, a server responds with an HTTP response message:

```
• A Status-line

• Zero or more header (General|Response|Entity) fields followed by CRLF

• An empty line (i.e., a line with nothing preceding the CRLF) indicating the end of
  the header fields

• Optionally a message-body
```

Following section will explain each of the entities used in HTTP message.

Message Status-Line

The Status-Line consisting of the protocol version followed by a numeric status code and its associated textual phrase. The elements are separated by space SP characters.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Let's discuss each of the part mentioned in Status-Line.

HTTP Version

A server supporting HTTP version 1.1 will return following version information:

```
HTTP-Version = HTTP/1.1
```

Status Code

The Status-Code element is a 3-digit integer where first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

S.N.	Code and Description
1	**1xx: Informational** This means request received and continuing process.
2	**2xx: Success** This means the action was successfully received, understood, and accepted.
3	**3xx: Redirection** This means further action must be taken in order to complete the request.

4	**4xx: Client Error** This means the request contains bad syntax or cannot be fulfilled
5	**5xx: Server Error** The server failed to fulfill an apparently valid request

HTTP status codes are extensible and HTTP applications are not required to understand the meaning of all registered status codes. A list of all the status code has been given in a separate chapter for you reference.

Response Header Fields

We will study General-header and Entity-header in a separate chapter when we will learn HTTP header fields. For now let's check what are Response header fields.

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

- Accept-Ranges
- Age
- ETag
- Location
- Proxy-Authenticate
- Retry-After
- Server
- Vary
- WWW-Authenticate

You can introduce your custom fields in case you are going to write your own custom Web Client and Server.

Response Message Examples

Now let's put it all together to form an HTTP response for a request to fetch **hello.htm** page from the web server running on tutorialspoint.com

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Following is an example of HTTP response message showing error condition when web server could not find requested page:

```
HTTP/1.1 404 Not Found
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
```

```
Connection: Closed
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL /t.html was not found on this server.</p>
</body>
</html>
```

Following is an example of HTTP response message showing error condition when web server encountered a wrong HTTP version in given HTTP request:

```
HTTP/1.1 400 Bad Request
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Content-Type: text/html; charset=iso-8859-1
Connection: Closed

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>400 Bad Request</title>
</head>
<body>
   <h1>Bad Request</h1>
   <p>Your browser sent a request that this server could not understand.<p>
   <p>The request line contained invalid characters following the protocol string.<p>
</body>
</html>
```

HTTP - METHODS

The set of common methods for HTTP/1.1 is defined below and this set can be expanded based on requirement. These method names are case sensitive and they must be used in uppercase.

S.N.	Method and Description
1	**GET** The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
2	**HEAD** Same as GET, but only transfer the status line and header section.
3	**POST** A POST request is used to send data to the server, for example customer information, file upload etc using HTML forms.
4	**PUT** Replace all current representations of the target resource with the uploaded content.
5	**DELETE** Remove all current representations of the target resource given by URI.
6	**CONNECT** Establish a tunnel to the server identified by a given URI.
7	**OPTIONS**

Describe the communication options for the target resource.

8 **TRACE**
Perform a message loop-back test along the path to the target resource.

GET Method

A GET request retrieves data from a web server by specifying parameters in the URL portion of the request. This is the main method used for document retrieval. Following is a simple example which makes use of GET method to fetch hello.htm:

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Following will be a server response against the above GET request:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Vary: Authorization,Accept
Accept-Ranges: bytes
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

HEAD Method

The HEAD method is functionally like GET, except that the server replies with a response line and headers, but no entity-body. Following is a simple example which makes use of HEAD method to fetch header information about hello.htm:

```
HEAD /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Following will be a server response against the above GET request:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Vary: Authorization,Accept
Accept-Ranges: bytes
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

You can notice that here server does not send any data after header.

POST Method

The POST method is used when you want to send some data to the server, for example file update, form data etc. Following is a simple example which makes use of POST method to send a form data to the server which will be processed by a process.cgi and finally a response will be returned:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: 88
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://clearforest.com/">string</string>
```

Server side script process.cgi process the passed data and send following response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Vary: Authorization,Accept
Accept-Ranges: bytes
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Request Processed Successfully</h1>
</body>
</html>
```

PUT Method

The PUT method is used to request the server to store the included entity-body at a location specified by the given URL. The following example request server to save the given entity-boy in **hello.htm** at the root of the server:

```
PUT /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Connection: Keep-Alive
Content-type: text/html
Content-Length: 182

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

The server will store given entity-body in **hello.htm** file and will send following response back to the client:

```
HTTP/1.1 201 Created
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Content-type: text/html
Content-length: 30
Connection: Closed
```

```
<html>
<body>
<h1>The file was created.</h1>
</body>
</html>
```

DELETE Method

The DELETE method is used to request the server to delete file at a location specified by the given URL. The following example request server to delete the given file **hello.htm** at the root of the server:

```
DELETE /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Connection: Keep-Alive
```

The server will delete mentioned file **hello.htm** and will send following response back to the client:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Content-type: text/html
Content-length: 30
Connection: Closed

<html>
<body>
<h1>URL deleted.</h1>
</body>
</html>
```

CONNECT Method

The CONNECT method is used by the client to establish a network connection to a web server over HTTP. The following example request a connection with a web server running on host tutorialspoint.com:

```
CONNECT www.tutorialspoint.com HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
```

The connection is established with the server and following response is sent back to the client:

```
HTTP/1.1 200 Connection established
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
```

OPTIONS Method

The OPTIONS method is used by the client to find out what are the HTTP methods and other options supported by a web server. The client can specify a URL for the OPTIONS method, or an asterisk * to refer to the entire server. The following example request a list of methods supported by a web server running on tutorialspoint.com:

```
OPTIONS * HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
```

The server will send information based on the current configuration of the server, for example:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
```

```
Server: Apache/2.2.14 (Win32)
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Type: httpd/unix-directory
```

TRACE Method

The TRACE method is used to eacho the contents of an HTTP Request back to the requester which can be used for debugging purpose at the time of development. The following example shows the usage of TRACE method:

```
TRACE / HTTP/1.1
Host: www.tutorialspoint.com
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
```

The server will send following message in response of the above request:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Content-Type: message/http
Content-Length: 39
Connection: Closed

TRACE / HTTP/1.1
Host: www.tutorialspoint.com
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
```

HTTP - STATUS CODES

The Status-Code element in a server response, is a 3-digit integer where first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

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

HTTP status codes are extensible and HTTP applications are not required to understand the meaning of all registered status codes. Following is a list of all the status code.

1xx: Information

Message:	Description:
100 Continue	Only a part of the request has been received by the server, but as long as it has not been rejected, the client should continue with the request

101 Switching Protocols	The server switches protocol

2xx: Successful

Message:	**Description:**
200 OK	The request is OK
201 Created	The request is complete, and a new resource is created
202 Accepted	The request is accepted for processing, but the processing is not complete
203 Non-authoritative Information	The information in the entity header is from a local or third-party copy, not from the original server.
204 No Content	A status code and header are given in the response, but there is no entity-body in the reply.
205 Reset Content	The browser should clear the form used for this transaction for additional input.
206 Partial Content	The server is returning partial data of the size requested. Used in response to a request specifying a *Range* header. The server must specify the range included in the response with the *Content-Range* header.

3xx: Redirection

Message:	**Description:**
300 Multiple Choices	A link list. The user can select a link and go to that location. Maximum five addresses
301 Moved Permanently	The requested page has moved to a new url
302 Found	The requested page has moved temporarily to a new url
303 See Other	The requested page can be found under a different url
304 Not Modified	This is the response code to an *If-Modified-Since* or *If-None-Match* header, where the URL has not been modified since the specified date.
305 Use Proxy	The requested URL must be accessed through the proxy mentioned in the *Location* header.
306 *Unused*	This code was used in a previous version. It is no longer used, but the code is reserved
307 Temporary Redirect	The requested page has moved temporarily to a new url

4xx: Client Error

Message:	**Description:**

400 Bad Request	The server did not understand the request
401 Unauthorized	The requested page needs a username and a password
402 Payment Required	*You can not use this code yet*
403 Forbidden	Access is forbidden to the requested page
404 Not Found	The server can not find the requested page
405 Method Not Allowed	The method specified in the request is not allowed
406 Not Acceptable	The server can only generate a response that is not accepted by the client
407 Proxy Authentication Required	You must authenticate with a proxy server before this request can be served
408 Request Timeout	The request took longer than the server was prepared to wait
409 Conflict	The request could not be completed because of a conflict
410 Gone	The requested page is no longer available
411 Length Required	The "Content-Length" is not defined. The server will not accept the request without it
412 Precondition Failed	The precondition given in the request evaluated to false by the server
413 Request Entity Too Large	The server will not accept the request, because the request entity is too large
414 Request-url Too Long	The server will not accept the request, because the url is too long. Occurs when you convert a "post" request to a "get" request with a long query information
415 Unsupported Media Type	The server will not accept the request, because the media type is not supported
416 Requested Range Not Satisfiable	The requested byte range is not available and is out of bounds.
417 Expectation Failed	The expectation given in an Expect request-header field could not be met by this server.

5xx: Server Error

Message:	**Description:**
500 Internal Server Error	The request was not completed. The server met an unexpected condition
501 Not Implemented	The request was not completed. The server did not support the functionality required
502 Bad Gateway	The request was not completed. The server received an invalid response from the upstream server
503 Service Unavailable	The request was not completed. The server is temporarily overloading or down

504 Gateway Timeout	The gateway has timed out
505 HTTP Version Not Supported	The server does not support the "http protocol" version

HTTP - HEADER FIELDS

HTTP deader fields provide required information about the request or response, or about the object sent in the message body. There are following four types of HTTP message headers:

- **General-header:** These header fields have general applicability for both request and response messages.
- **Client Request-header:** These header fields are applicability only for request messages.
- **Server Response-header:** These header fields are applicability only for response messages.
- **Entity-header:** These header fields define metainformation about the entity-body or, if no body is present

General Headers

Cache-control

The Cache-Control general-header field is used to specify directives that MUST be obeyed by all caching system. Following is the syntax:

```
Cache-Control : cache-request-directive|cache-response-directive
```

An HTTP clients or servers can use the **Cache-control** general header to specify parameters for the cache or to request certain kinds of documents from the cache. The caching directives are specified in a comma-separated list. For example:

```
Cache-control: no-cache
```

There are following important cache request directives which can be used by the client in its HTTP request:

S.N.	Cache Request Directive and Description
1	**no-cache** A cache must not use the response to satisfy a subsequent request without successful revalidation with the origin server.
2	**no-store** The cache should not store anything about the client request or server response.
3	**max-age = seconds** Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds.
4	**max-stale [= seconds]** Indicates that the client is willing to accept a response that has exceeded its expiration time. If seconds are given, it must not be expired by more than that time.
5	**min-fresh = seconds** Indicates that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the specified time in seconds.
6	**no-transform** Do not convert the entity-body.

7	**only-if-cached** Do not retrieve new data. The cache can send a document only if it is in the cache, and should not contact the origin-server to see if a newer copy exists.

There are following important cache response directives which can be used by the server in its HTTP response:

S.N.	Cache Request Directive and Description
1	**public** Indicates that the response may be cached by any cache.
2	**private** Indicates that all or part of the response message is intended for a single user and must not be cached by a shared cache.
3	**no-cache** A cache must not use the response to satisfy a subsequent request without successful revalidation with the origin server.
4	**no-store** The cache should not store anything about the client request or server response.
5	**no-transform** Do not convert the entity-body.
6	**must-revalidate** The cache must verify the status of stale documents before using it and expired one should not be used.
7	**proxy-revalidate** The proxy-revalidate directive has the same meaning as the must- revalidate directive, except that it does not apply to non-shared user agent caches.
8	**max-age = seconds** Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds.
9	**s-maxage = seconds** The maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header. The s-maxage directive is always ignored by a private cache.

Connection

The Connection general-header field allows the sender to specify options that are desired for that particular connection and must not be communicated by proxies over further connections. Following is the simple syntax of using connection header:

```
Connection : "Connection"
```

HTTP/1.1 defines the "closed" connection option for the sender to signal that the connection will be closed after completion of the response. For example:

```
Connection: Closed
```

By default, HTTP 1.1 uses persistent connections, where the connection does not automatically close after a transaction. HTTP 1.0, on the other hand, does not have persistent connections by default. If a 1.0 client wishes to use persistent connections, it uses the **keep-alive** parameter as follows:

```
Connection: keep-alive
```

Date

All HTTP date/time stamps MUST be represented in Greenwich Mean Time *GMT*, without exception. HTTP applications are allowed to use any of the following three representations of date/time stamps:

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime() format
```

Here first format is the most preferred one.

Pragma

The Pragma general-header field is used to include implementation- specific directives that might apply to any recipient along the request/response chain. For example:

```
Pragma: no-cache
```

The only directive defined in HTTP/1.0 is the no-cache directive and is maintained in HTTP 1.1 for backward compatibility. No new Pragma directives will be defined in the future.

Trailer

The Trailer general field value indicates that the given set of header fields is present in the trailer of a message encoded with chunked transfer-coding. Following is the syntax of Trailer header field:

```
Trailer : field-name
```

Message header fields listed in the Trailer header field must not include the following header fields:

- Transfer-Encoding
- Content-Length
- Trailer

Transfer-Encoding

The *Transfer-Encoding* general-header field indicates what type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient. This is not the same as content-encoding because transfer-encodings are a property of the message, not of the entity-body. Following is the syntax of Transfer-Encoding header field:

```
Transfer-Encoding: chunked
```

All transfer-coding values are case-insensitive.

Upgrade

The *Upgrade* general-header allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. For example:

```
Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11
```

The Upgrade header field is intended to provide a simple mechanism for transition from HTTP/1.1 to some other, incompatible protocol

Via

The *Via* general-header must be used by gateways and proxies to indicate the intermediate protocols and recipients. For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at nowhere.com, which completes the request by forwarding it to the origin server at www.ics.uci.edu. The request received by www.ics.uci.edu would then have the following Via header field:

```
Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
```

The Upgrade header field is intended to provide a simple mechanism for transition from HTTP/1.1 to some other, incompatible protocol

Warning

The *Warning* general-header is used to carry additional information about the status or transformation of a message which might not be reflected in the message. A response may carry more than one Warning header.

```
Warning : warn-code SP warn-agent SP warn-text SP warn-date
```

Client Request Headers

Accept

The *Accept* request-header field can be used to specify certain media types which are acceptable for the response. Following is the general syntax:

```
Accept: type/subtype [q=qvalue]
```

Multiple media types can be listed separated by commas and the optional qvalue represents an acceptable quality level for accept types on a scale of 0 to 1. Following is an example:

```
Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.8, text/x-c
```

This would be interpreted as **text/html** and **text/x-c** are the preferred media types, but if they do not exist, then send the **text/x-dvi** entity, and if that does not exist, send the **text/plain** entity.

Accept-Charset

The *Accept-Charset* request-header field can be used to indicate what character sets are acceptable for the response. Following is the general syntax:

```
Accept-Charset: character_set [q=qvalue]
```

Multiple character sets can be listed separated by commas and the optional qvalue represents an acceptable quality level for nonpreferred character sets on a scale of 0 to 1. Following is an example:

```
Accept-Charset: iso-8859-5, unicode-1-1; q=0.8
```

The special value "*", if present in the **Accept-Charset** field, matches every character set and if no **Accept-Charset** header is present, the default is that any character set is acceptable.

Accept-Encoding

The *Accept-Encoding* request-header field is similar to Accept, but restricts the content-codings that are acceptable in the response. Following is the general syntax:

```
Accept-Encoding: encoding types
```

Following are examples:

```
Accept-Encoding: compress, gzip
Accept-Encoding:
Accept-Encoding: *
Accept-Encoding: compress;q=0.5, gzip;q=1.0
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

Accept-Language

The *Accept-Language* request-header field is similar to Accept, but restricts the set of natural languages that are preferred as a response to the request. Following is the general syntax:

```
Accept-Language: language [q=qvalue]
```

Multiple languages can be listed separated by commas and the optional qvalue represents an acceptable quality level for nonpreferred languages on a scale of 0 to 1. Following is an example:

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

Authorization

The *Authorization* request-header field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested. Following is the general syntax:

```
Authorization : credentials
```

The HTTP/1.0 specification defines the BASIC authorization scheme, where the authorization parameter is the string of **username:password** encoded in base 64. Following is an example:

```
Authorization: BASIC Z3Vlc3Q6Z3Vlc3QxMjM=
```

The value decodes into is **guest:guest123** where **guest** is user ID and **guest123** is the password.

Cookie

The *Cookie* request-header field value contains a name/value pair of information stored for that URL. Following is the general syntax:

```
Cookie: name=value
```

Multiple cookies can be specified separated by semicolons as follows:

```
Cookie: name1=value1;name2=value2;name3=value3
```

Expect

The *Expect* request-header field is used to indicate that particular server behaviors are required by the client. Following is the general syntax:

```
Expect : 100-continue | expectation-extension
```

If a server receives a request containing an Expect field that includes an expectation-extension that it does not support, it must respond with a 417 $ExpectationFailed$ status.

From

The *From* request-header field contains an Internet e-mail address for the human user who controls the requesting user agent. Following is a simple example:

```
From: webmaster@w3.org
```

This header field may be used for logging purposes and as a means for identifying the source of invalid or unwanted requests.

Host

The *Host* request-header field is used to specify the Internet host and port number of the resource being requested. Following is the general syntax:

```
Host : "Host" ":" host [ ":" port ] ;
```

A **host** without any trailing port information implies the default port, which is 80. For example, a request on the origin server for *http://www.w3.org/pub/WWW/* would be:

```
GET /pub/WWW/ HTTP/1.1
Host: www.w3.org
```

If-Match

The *If-Match* request-header field is used with a method to make it conditional. This header request the server to perform the requested method only if given value in this tag matches the given entity tags represented by **ETag**. Following is the general syntax:

```
If-Match : entity-tag
```

An asterisk $*$ matches any entity, and the transaction continues only if the entity exists. Following are possible examples:

```
If-Match: "xyzzy"
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *
```

If none of the entity tags match, or if "*" is given and no current entity exists, the server must not perform the requested method, and must return a 412 $PreconditionFailed$ response.

If-Modified-Since

The *If-Modified-Since* request-header field is used with a method to make it conditional. If the requested URL has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 $notmodified$ response will be returned without any message-body. Following is the general syntax:

```
If-Modified-Since : HTTP-date
```

An example of the field is:

```
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

If none of the entity tags match, or if "*" is given and no current entity exists, the server must not perform the requested method, and must return a 412 $PreconditionFailed$ response.

If-None-Match

The *If-None-Match* request-header field is used with a method to make it conditional. This header request the server to perform the requested method only if one of the given value in this tag matches the given entity tags represented by **ETag**. Following is the general syntax:

```
If-None-Match : entity-tag
```

An asterisk * matches any entity, and the transaction continues only if the entity does not exist. Following are possible examples:

```
If-None-Match: "xyzzy"
If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-None-Match: *
```

If-Range

The *If-Range* request-header field can be used with a conditional GET to request only the portion of the entity that is missing, if it has not been changed, and the entire entity if it has changed. Following is the general syntax:

```
If-Range : entity-tag | HTTP-date
```

Either an entity tag or a date can be used to identify the partial entity already received. For example:

```
If-Range: Sat, 29 Oct 1994 19:43:31 GMT
```

Here if the document has not been modified since the given date, the server returns the byte range given by the Range header otherwise, it returns all of the new document.

If-Unmodified-Since

The *If-Unmodified-Since* request-header field is used with a method to make it conditional. Following is the general syntax:

```
If-Unmodified-Since : HTTP-date
```

If the requested resource has not been modified since the time specified in this field, the server should perform the requested operation as if the If-Unmodified-Since header were not present. For example:

```
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

If the request normally would result in anything other than a 2xx or 412 status, the *If-Unmodified-Since* header should be ignored.

Max-Forwards

The *Max-Forwards* request-header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server. Following is the general syntax:

```
Max-Forwards : n
```

The Max-Forwards value is a decimal integer indicating the remaining number of times this request message may be forwarded. This is useful for debugging with the TRACE method, avoiding infinite loops. For example:

```
Max-Forwards : 5
```

The Max-Forwards header field may be ignored for all other methods defined in HTTP specification.

Proxy-Authorization

The *Proxy-Authorization* request-header field allows the client to identify itself *oritsuser* to a proxy which requires authentication. Following is the general syntax:

```
Proxy-Authorization : credentials
```

The Proxy-Authorization field value consists of credentials containing the authentication information of the user agent for the proxy and/or realm of the resource being requested.

Range

The *Range* request-header field specifies the partial ranges of the content requested from the document. Following is the general syntax:

```
Range: bytes-unit=first-byte-pos "-" [last-byte-pos]
```

The first-byte-pos value in a byte-range-spec gives the byte-offset of the first byte in a range. The last-byte-pos value gives the byte-offset of the last byte in the range; that is, the byte positions specified are inclusive. You can specify a byte-unit as bytes Byte offsets start at zero. Following are a simple examples:

```
- The first 500 bytes
Range: bytes=0-499

- The second 500 bytes
Range: bytes=500-999

- The final 500 bytes
Range: bytes=-500

- The first and last bytes only
Range: bytes=0-0,-1
```

Multiple ranges can be listed, separated by commas. If the first digit in the comma-separated byte ranges is missing, the range is assumed to count from the end of the document. If the second digit is missing, the range is byte n to the end of the document.

Referer

The *Referer* request-header field allows the client to specify the address *URI* of the resource from which the URL has been requested. Following is the general syntax:

```
Referer : absoluteURI | relativeURI
```

Following is a simple example:

```
Referer: http://www.tutorialspoint.org/http/index.htm
```

If the field value is a relative URI, it should be interpreted relative to the *Request-URI*.

TE

The *TE* request-header field indicates what extension *transfer-coding* it is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked *transfer-coding*. Following is the general syntax:

```
TE   : t-codings
```

The presence of the keyword "trailers" indicates that the client is willing to accept trailer fields in a chunked transfer-coding and it is specified either of the ways:

```
TE: deflate
TE:
TE: trailers, deflate;q=0.5
```

If the TE field-value is empty or if no TE field is present, the only transfer-coding is *chunked*. A message with no transfer-coding is always acceptable.

User-Agent

The *User-Agent* request-header field contains information about the user agent originating the request. Following is the general syntax:

```
User-Agent : product | comment
```

Example:

```
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
```

Server Response Headers

Accept-Ranges

The *Accept-Ranges* response-header field allows the server to indicate its acceptance of range requests for a resource. Following is the general syntax:

```
Accept-Ranges  : range-unit | none
```

For example a server that accept byte-range requests may send

```
Accept-Ranges: bytes
```

Servers that do not accept any kind of range request for a resource may send:

```
Accept-Ranges: none
```

This will advise the client not to attempt a range request.

Age

The *Age* response-header field conveys the sender's estimate of the amount of time since the response *oritsrevalidation* was generated at the origin server. Following is the general syntax:

```
Age : delta-seconds
```

Age values are non-negative decimal integers, representing time in seconds. Following is a simple example:

```
Age: 1030
```

An HTTP/1.1 server that includes a cache must include an Age header field in every response generated from its own cache.

ETag

The *ETag* response-header field provides the current value of the entity tag for the requested variant. Following is the general syntax:

```
ETag :  entity-tag
```

Following are simple examples:

```
ETag: "xyzzy"
ETag: W/"xyzzy"
ETag: ""
```

Location

The *Location* response-header field is used to redirect the recipient to a location other than the Request-URI for completion. Following is the general syntax:

```
Location : absoluteURI
```

Following is a simple example:

```
Location: http://www.tutorialspoint.org/http/index.htm
```

The Content-Location header field differs from Location in that the Content-Location identifies the original location of the entity enclosed in the request.

Proxy-Authenticate

The *Proxy-Authenticate* response-header field must be included as part of a 407 *ProxyAuthenticationRequired* response. Following is the general syntax:

```
Proxy-Authenticate  : challenge
```

Retry-After

The *Retry-After* response-header field can be used with a 503 *ServiceUnavailable* response to indicate how long the service is expected to be unavailable to the requesting client. Following is the general syntax:

```
Retry-After : HTTP-date | delta-seconds
```

Following are two simple examples:

```
Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
Retry-After: 120
```

In the latter example, the delay is 2 minutes.

Server

The *Server* response-header field contains information about the software used by the origin server to handle the request. Following is the general syntax:

```
Server : product | comment
```

Following is a simple example:

```
Server: Apache/2.2.14 (Win32)
```

If the response is being forwarded through a proxy, the proxy application must not modify the Server response-header.

Set-Cookie

The *Set-Cookie* response-header field contains a name/value pair of information to retain for this URL. Following is the general syntax:

```
Set-Cookie: NAME=VALUE; OPTIONS
```

Set-Cookie response header comprises the token Set-Cookie:, followed by a comma-separated list of one or more cookies. Here are possible values you can specify as options:

S.N.	Options and Description
1	**Comment=comment** This option can be used to specify any comment associated with the cookie.
2	**Domain=domain** The Domain attribute specifies the domain for which the cookie is valid.
3	**Expires=Date-time** The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser
4	**Path=path** The Path attribute specifies the subset of URLs to which this cookie applies.
5	**Secure** This instructs the user agent to return the cookie only under a secure connection.

Following is an example of a simple cookie header generated by the server:

```
Set-Cookie: name1=value1,name2=value2; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

Vary

The *Vary* response-header field specifies that the entity has multiple sources and may therefore vary according to specified list of request headers. Following is the general syntax:

```
Vary : field-name
```

You can specify multiple headers separated by commas and a value of asterisk "*" signals that unspecified parameters not limited to the request-headers. Following is a simple example:

```
Vary: Accept-Language, Accept-Encoding
```

Here field names are case-insensitive.

WWW-Authenticate

The *WWW-Authenticate* response-header field must be included in 401 *Unauthorized* response messages. The field value consists of at least one challenge that indicates the authentication schemes and parameters applicable to the Request-URI. Following is the general syntax:

```
WWW-Authenticate : challenge
```

WWW- Authenticate field value as it might contain more than one challenge, or if more than one WWW-Authenticate header field is provided, the contents of a challenge itself can contain a comma-separated list of authentication parameters. Following is a simple example:

```
WWW-Authenticate: BASIC realm="Admin"
```

Entity Headers

Allow

The *Allow* entity-header field lists the set of methods supported by the resource identified by the Request-URI. Following is the general syntax:

```
Allow : Method
```

You can specify multiple method separated by commas. Following is a simple example:

```
Allow: GET, HEAD, PUT
```

This field cannot prevent a client from trying other methods.

Content-Encoding

The *Content-Encoding* entity-header field is used as a modifier to the media-type. Following is the general syntax:

```
Content-Encoding : content-coding
```

The content-coding is a characteristic of the entity identified by the Request-URI. Following is a simple example:

```
Content-Encoding: gzip
```

If the content-coding of an entity in a request message is not acceptable to the origin server, the server should respond with a status code of 415 *UnsupportedMediaType*.

Content-Language

The *Content-Language* entity-header field describes the natural languages of the intended audience for the enclosed entity. Following is the general syntax:

```
Content-Language : language-tag
```

Multiple languages may be listed for content that is intended for multiple audiences. Following is a simple example:

```
Content-Language: mi, en
```

The primary purpose of Content-Language is to allow a user to identify and differentiate entities according to the user's own preferred language.

Content-Length

The *Content-Length* entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET. Following is the general syntax:

```
Content-Length : DIGITS
```

Following is a simple example:

```
Content-Length: 3495
```

Any Content-Length greater than or equal to zero is a valid value.

Content-Location

The *Content-Location* entity-header field may be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI. Following is the general syntax:

```
Content-Location:  absoluteURI | relativeURI
```

Following is a simple example:

```
Content-Location: http://www.tutorialspoint.org/http/index.htm
```

The value of Content-Location also defines the base URI for the entity.

Content-MD5

The *Content-MD5* entity-header field may be used to supply an MD5 digest of the entity, for checking the integrity of the message upon receipt. Following is the general syntax:

```
Content-MD5  : md5-digest using base64 of 128 bit MD5 digest as per RFC 1864
```

Following is a simple example:

```
Content-MD5  : 8c2d46911f3f5a326455f0ed7a8ed3b3
```

The MD5 digest is computed based on the content of the entity-body, including any content-coding that has been applied, but not including any transfer-encoding applied to the message-body.

Content-Range

The *Content-Range* entity-header field is sent with a partial entity-body to specify where in the full entity-body the partial body should be applied. Following is the general syntax:

```
Content-Range : bytes-unit SP first-byte-pos "-" last-byte-pos
```

Examples of byte-content-range-spec values, assuming that the entity contains a total of 1234 bytes:

```
- The first 500 bytes:
Content-Range : bytes 0-499/1234

- The second 500 bytes:
Content-Range : bytes 500-999/1234

- All except for the first 500 bytes:
Content-Range : bytes 500-1233/1234

- The last 500 bytes:
Content-Range : bytes 734-1233/1234
```

When an HTTP message includes the content of a single range, this content is transmitted with a Content-Range header, and a Content-Length header showing the number of bytes actually transferred. For example,

```
HTTP/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif
```

Content-Type

The *Content-Type* entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET. Following is the general syntax:

```
Content-Type : media-type
```

Following is an example:

```
Content-Type: text/html; charset=ISO-8859-4
```

Expires

The *Expires* entity-header field gives the date/time after which the response is considered stale. Following is the general syntax:

```
Expires : HTTP-date
```

Following is an example:

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

Last-Modified

The *Last-Modified* entity-header field indicates the date and time at which the origin server believes the variant was last modified. Following is the general syntax:

```
Last-Modified: HTTP-date
```

Following is an example:

```
Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
```

HTTP - CACHING

HTTP is typically used for distributed information systems, where performance can be improved by the use of response caches. The HTTP/1.1 protocol includes a number of elements intended to make caching work.

The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases.

The basic cache mechanisms in HTTP/1.1 are implicit directives to caches where server-specifies expiration times and validators. We use the **Cache-Control** header for this purpose.

The **Cache-Control** header allows a client or server to transmit a variety of directives in either requests or responses. These directives typically override the default caching algorithms. The caching directives are specified in a comma-separated list. For example:

```
Cache-control: no-cache
```

There are following important cache request directives which can be used by the client in its HTTP request:

S.N.	Cache Request Directive and Description
1	**no-cache** A cache must not use the response to satisfy a subsequent request without successful revalidation with the origin server.
2	**no-store** The cache should not store anything about the client request or server response.
3	**max-age = seconds** Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds.
4	**max-stale [= seconds]** Indicates that the client is willing to accept a response that has exceeded its expiration time. If seconds are given, it must not be expired by more than that time.
5	**min-fresh = seconds**

	Indicates that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the specified time in seconds.
6	**no-transform** Do not convert the entity-body.
7	**only-if-cached** Do not retrieve new data. The cache can send a document only if it is in the cache, and should not contact the origin-server to see if a newer copy exists.

There are following important cache response directives which can be used by the server in its HTTP response:

S.N.	Cache Request Directive and Description
1	**public** Indicates that the response may be cached by any cache.
2	**private** Indicates that all or part of the response message is intended for a single user and must not be cached by a shared cache.
3	**no-cache** A cache must not use the response to satisfy a subsequent request without successful revalidation with the origin server.
4	**no-store** The cache should not store anything about the client request or server response.
5	**no-transform** Do not convert the entity-body.
6	**must-revalidate** The cache must verify the status of stale documents before using it and expired one should not be used.
7	**proxy-revalidate** The proxy-revalidate directive has the same meaning as the must- revalidate directive, except that it does not apply to non-shared user agent caches.
8	**max-age = seconds** Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds.
9	**s-maxage = seconds** The maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header. The s-maxage directive is always ignored by a private cache.

HTTP - URL ENCODING

HTTP URLs can only be sent over the Internet using the ASCII *character-set*, which often contain characters outside the ASCII set. So these unsafe characters must be replaced with a **%** followed by two hexadecimal digits.

Following table shows ASCII symbol of the character and its equal Symbol and finally its replacement which can be used in URL before passing it to the server:

ASCII	Symbol	Replacement
< 32		Encode with %xx where xx is the hexadecimal representation of the

		character.
32	space	+ or %20
33	!	%21
34	"	%22
35	#	%23
36	$	%24
37	%	%25
38	&	%26
39	'	%27
40	(	%28
41	)	%29
42	*	*
43	+	%2B
44	,	%2C
45	-	-
46	.	.
47	/	%2F
48	0	0
49	1	1
50	2	2
51	3	3
52	4	4
53	5	5
54	6	6
55	7	7
56	8	8
57	9	9
58	:	%3A
59	;	%3B
60	<	%3C
61	=	%3D
62	>	%3E
63	?	%3F
64	@	%40
65	A	A

66	B	B
67	C	C
68	D	D
69	E	E
70	F	F
71	G	G
72	H	H
73	I	I
74	J	J
75	K	K
76	L	L
77	M	M
78	N	N
79	O	O
80	P	P
81	Q	Q
82	R	R
83	S	S
84	T	T
85	U	U
86	V	V
87	W	W
88	X	X
89	Y	Y
90	Z	Z
91	[	%5B
92	\	%5C
93	]	%5D
94	^	%5E
95	_	_
96	`	%60
97	a	a
98	b	b
99	c	c
100	d	d

101	e	e
102	f	f
103	g	g
104	h	h
105	i	i
106	j	j
107	k	k
108	l	l
109	m	m
110	n	n
111	o	o
112	p	p
113	q	q
114	r	r
115	s	s
116	t	t
117	u	u
118	v	v
119	w	w
120	x	x
121	y	y
122	z	z
123	{	%7B
124	|	%7C
125	}	%7D
126	~	%7E
127		%7F
> 127		Encode with %xx where xx is the hexadecimal representation of the character

HTTP - SECURITY

HTTP is used for a communication over the internet, so application developers, information providers, and users should be aware of the security limitations in HTTP/1.1. This discussion does not include definitive solutions to the problems mentioned here but it does make some suggestions for reducing security risks.

Personal Information leakage

HTTP clients are often privy to large amounts of personal information such as the user's name, location, mail address, passwords, encryption keys, etc. So you should be very careful to prevent unintentional leakage of this information via the HTTP protocol to other sources.

- All the confidential information should be stored at server side in encrypted form.
- Revealing the specific software version of the server might allow the server machine to become more vulnerable to attacks against software that is known to contain security holes.
- Proxies which serve as a portal through a network firewall should take special precautions regarding the transfer of header information that identifies the hosts behind the firewall.
- The information sent in the From field might conflict with the user's privacy interests or their site's security policy, and hence it should not be transmitted without the user being able to disable, enable, and modify the contents of the field.
- Clients should not include a Referer header field in a $non-secure$ HTTP request if the referring page was transferred with a secure protocol.
- Authors of services which use the HTTP protocol should not use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the Request-URI

File and path names based attack

The document should be restricted to the documents returned by HTTP requests to be only those that were intended by the server administrators.

For example, UNIX, Microsoft Windows, and other operating systems use **..** as a path component to indicate a directory level above the current one. On such a system, an HTTP server MUST disallow any such construct in the Request-URI if it would otherwise allow access to a resource outside those intended to be accessible via the HTTP server.

DNS Spoofing

Clients using HTTP rely heavily on the Domain Name Service, and are thus generally prone to security attacks based on the deliberate mis-association of IP addresses and DNS names. So clients need to be cautious in assuming the continuing validity of an IP number/DNS name association.

If HTTP clients cache the results of host name lookups in order to achieve a performance improvement, they must observe the TTL information reported by DNS. If HTTP clients do not observe this rule, they could be spoofed when a previously-accessed server's IP address changes.

Location Headers and Spoofing

If a single server supports multiple organizations that do not trust one another, then it MUST check the values of Location and Content- Location headers in responses that are generated under control of said organizations to make sure that they do not attempt to invalidate resources over which they have no authority.

Authentication Credentials

Existing HTTP clients and user agents typically retain authentication information indefinitely. HTTP/1.1. does not provide a method for a server to direct clients to discard these cached credentials which is a big security risk.

There are a number of work- arounds to parts of this problem, and so its is recommended to make the use of password protection in screen savers, idle time-outs, and other methods which mitigate the security problems inherent in this problem.

Proxies and Caching

HTTP proxies are men-in-the-middle, and represent an opportunity for man-in-the-middle attacks. Proxies have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers.

Proxy operators should protect the systems on which proxies run as they would protect any system that contains or transports sensitive information.

Caching proxies provide additional potential vulnerabilities, since the contents of the cache represent an attractive target for malicious exploitation. Therefore, cache contents should be protected as sensitive information.

HTTP - MESSAGE EXAMPLES

Example 1

HTTP request to fetch **hello.htm** page from the web server running on *tutorialspoint.com*

Client request

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Server response

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Example 2

HTTP request to fetch **t.html** page which does not exist on the web server running on *tutorialspoint.com*

Client request

```
GET /t.html HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Server response

```
HTTP/1.1 404 Not Found
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Content-Type: text/html; charset=iso-8859-1
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
```

```
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL /t.html was not found on this server.</p>
</body>
</html>
```

Example 3

HTTP request to fetch **hello.htm** page from the web server running on *tutorialspoint.com*, but request goes with wrong HTTP version:

Client request

```
GET /hello.htm HTTP1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Server response

```
HTTP/1.1 400 Bad Request
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Content-Type: text/html; charset=iso-8859-1
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>400 Bad Request</title>
</head>
<body>
   <h1>Bad Request</h1>
   <p>Your browser sent a request that this server could not understand.<p>
   <p>The request line contained invalid characters following the protocol string.<p>
</body>
</html>
```

Example 4

HTTP request to post form data to **process.cgi** CGI page on a web server running on *tutorialspoint.com*. Server returns passed name after setting them as cookies:

Client request

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: 60
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

first=Zara&last=Ali
```

Server response

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 88
Set-Cookie: first=Zara,last=Ali;domain=tutorialspoint.com;Expires=Mon, 19-
Nov-2010 04:38:14 GMT;Path=/
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello Zara Ali</h1>
</body>
</html>
```

Processing math: 100%