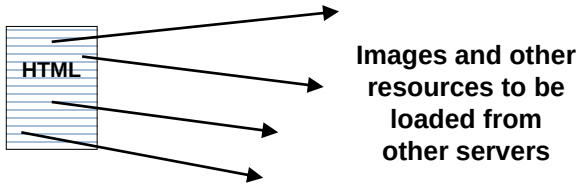
HTTP contents transfer.
Web Browsers and Web Servers.
Uniform Resource Locator.

SCOMRED, December 2021

Hypertext contents transfer

Hypertext refers to documents with live links to other documents, they may be directly clickable references (hyperlinks) or references to other resources (e.g. images) to be included in the current document's presentation.
In either case, references are links to other documents and resources. Each reference is represented by an URL with a filename location to be accessed through the network by using a specific file transfer application protocol.

To fully load an HTML (Hypertext Mark-up Language) document, beyond the file itself, there may be several references to additional resources to be loaded. For each, an additional file transfer will be required.



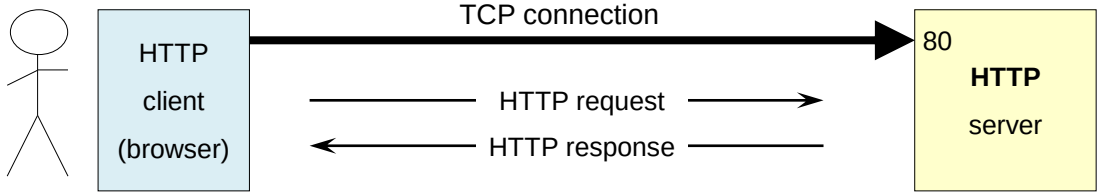
Although FTP (File Transfer Protocol) can be used, it proved to be inappropriate for this kind of use. FTP requires one control connection with user authentication (even if it's anonymous), and then another connection for each file transfer from that server. It's not suitable for transferring a big number or relatively small files from different servers at different locations.
To workaround FTP issues on hypertext, a content-oriented file transfer protocol was designed, the Hypertext Transfer Protocol (HTTP).

Hypertext Transfer Protocol (HTTP)

Despite earlier versions, the first fully functional version appeared in 1996, named as HTTP 1.0, it's still supported nowadays . The key idea for HTTP is providing an expedite data transfer, thought it may not be a file, so we just call it a content.
HTTP is also content-aware, this means it will exchange information about content related characteristics with applications using it.

The service model for HTTP is the typical TCP client-server. The client starts by establishing a TCP connection with the server (the standard TCP service port number is 80). Once the connection is established, the client sends an **HTTP request message**, the server must then send back an **HTTP response message**.

HTTP defines several request types, they are known as **methods**.



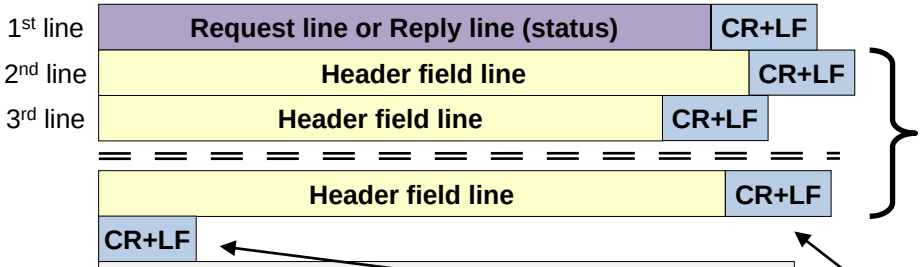
HTTP 1.0 defines GET, POST, and HEAD methods, earlier versions only had GET.

HTTP 1.1 adds to them, OPTIONS, PUT, DELETE, TRACE, and CONNECT methods.

HTTP message format

All HTTP messages (**both requests and responses**) share the same well-defined general format. They always start with a sequence of variable length text lines, terminated by one empty line. Every text line itself is terminated by CR (Carriage Return) byte followed by the LF (Line Feed) byte.

The first line is either the request line (for an HTTP request message), or the reply / status line (for an HTTP response message). Unlike the first one, additional text lines are optional, they are called **header fields**. Header fields transport additional information about the protocol operation and the message content.



A variable number of header field lines, there may be none.

Two consecutive CR+LF sequences (an empty line) points out the header's end, the message body (if exists) starts next.

CR	13	0x0D	\r
LF	10	0x0A	\n

HTTP – Request and response messages

The first line in an **HTTP request message** is called the **request line** and has the following format:

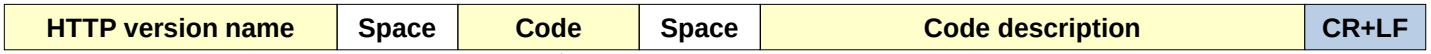
OPTIONS
GET
HEAD
POST
PUT
DELETE
TRACE
CONNECT

Identifies the resource over which the method will be enforced, no spaces neither CR or LF are allowed. It can may have one of three forms:

- An asterisk – not to be applied to any specific resource
- An absolute path (slash started) – a counterpart local resource (URI)
- An URI (may be an URL)

HTTP/1.0
HTTP/1.1
(…)

The first line in an **HTTP response message** is called the **status line** and has the following format:



HTTP/1.0
HTTP/1.1
(…)

The status code is a three digits number.

For instance, 200 means total success on the operation and usually will have the **OK** code description.

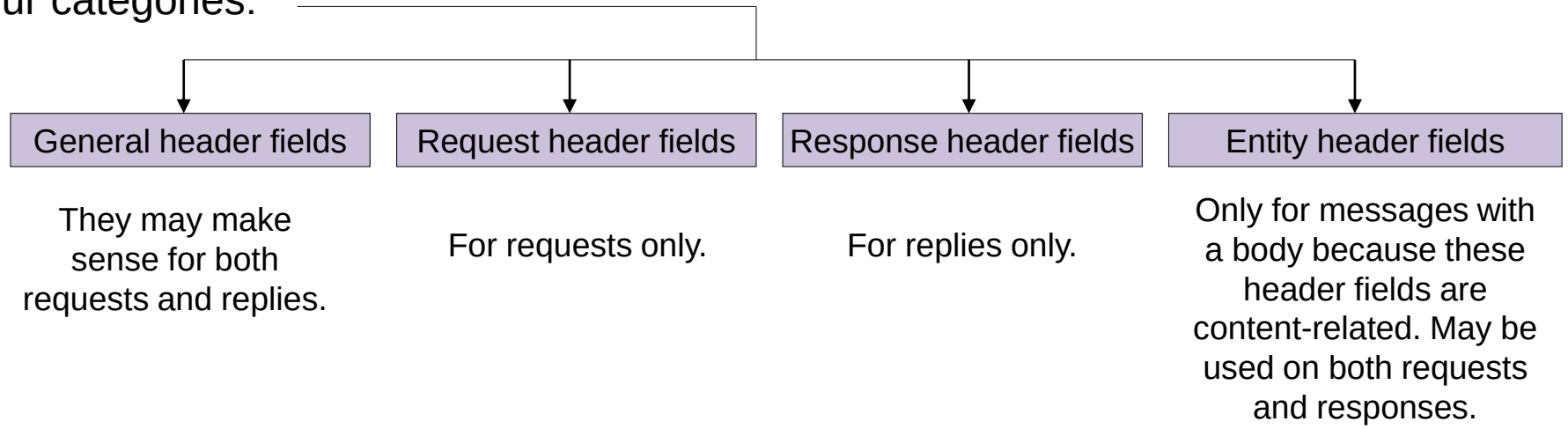
Header fields

Header fields are text lines used to transmit control information, either related to HTTP operations or related to the message's content (body).

The general form is:

Field name	:	Field value	CR+LF

Where **Field name** is a standard case-insensitive identifier with special meaning for HTTP, it's immediately followed by a colon, no whitespaces between.
The **Field value**, on the other hand, may be preceded by white characters, they should be ignored. Depending on the field name, the field value may or not be case-sensitive.
Both requests and replies usually have header fields, but many header fields only make sense for some messages. Traditionally, header lines have been divided into four categories:



HTTP/1.1 general-header fields

They may be used both in requests and responses and do not refer to the content. Some most often used general-header fields are:

Cache-Control	Settles how data may be cached by clients, servers and proxies, some possible values are **no-cache**, **no-store**, **max-age**, **public** and **private**. Not available for HTTP/1.0, Pragma must be used instead.
Connection	Unlike in HTTP/1.0, in HTTP/1.1 the default behavior is keeping connections open to allowing multiple requests and replies. The header field **Connection: close** informs the counterpart the connection is going to be closed after the current transaction.
Date	Hold the date/time of the message creation. For instance: **Date: Tue, 15 Nov 1994 08:12:31 GMT**
Pragma	Settles operational directives, most used value is **no-cache** to indicate no data caching is allowed.
Upgrade	Clients may include this on requests to inform the server about new protocol versions they support, the server may then inform the client it wants to switch to on of them by sending a **101 Switching Protocols** response with the Upgrade header field indicating to which is switching to. For instance: **Upgrade: HTTP/2**
Transfer-Encoding	Informs the counterpart about a transformation applied to the message body. This is similar to the entity header field **Content-Encoding**.

HTTP/1.1 entity-header fields

They are content-related. Even though, they make most sense for messages with a body, they are also used in some other cases. Some common entity-header fields are:

Allow	Informs about supported methods to access a resource. It will be included in a **405 Method Not Allowed** response.
Content-Encoding	Informs the message receiver about some coding was applied to the content, for instance: **Content-Encoding: gzip**
Content-Language	Describes the natural language of the intended audience for the content.
Content-Length	Defines the size in octets of the content. This is supposed to be used by message receivers to know how many bytes they should read from the body starting point.
Content-MD5	Holds the result of applying the Message Digest 5 algorithm to the content, used for integrity checking.
Content-Range	This is used for a partial content message body. It must specify the body position within the original content and the total original content length. Example: **Content-Range: bytes 21010-47021/47022**
Content-Type	Informs the receiver about the content media, thus, how the content should be interpreted and ultimately displayed to the end-user.
Expires	Contains a date/time after which cached copies of the content are no longer valid.
Last-Modified	Contains the date/time the content was last modified. If the content comes from a file may be the file last modification time.

HTTP/1.1 request-header fields

HTTP request messages specific. Some most often used request-header fields are:

From	Contains the personal e-mail address of the human user on the client application side.
Host	Contains the hostname and port number being accessed, either typed by the user at the browser or from the clicked URL in a document. Default port number is 80. Example: **Host: www.dei.isep.ipp.pt.pt:8080**
Referer (misspelled)	Contains the URL of the document from where the present request was followed (referred by). This field should not exist for directly user typed requests. Example: **Referer: http://www.dei.isep.ipp.pt/index.html**
User-Agent	A string identification for the client application, usually a browser. Example: **User-Agent: Mozilla/5.0 (Linux; Android 4.0.4; Galaxy Nexus Build/IMM76B)**
Authorization	User authentication data, usually username/password. Must be included following an **401 Unauthorized** response.
Cookie	Contains a pair name and value provided by the counterpart in a previous response. Example: **Cookie: sessionToken=ts12325**

HTTP/1.1 request-header fields – conditional requests

These HTTP request-header fields introduce client demands regarding contents to be returned in responses.

Accept	Requires the response content to be in one of the specified media types (content types). Example: **Accept: text/*, text/html, text/html;level=1, */***
Accept-Charset	Requires the response content to be in one of the specified charsets. Example: **Accept-Charset: iso-8859-5, unicode-1-1;q=0.8**
Accept-Encoding	Restricts possible content-coding values for the response content. Example: **Accept-Encoding: compress, gzip**
Accept-Language	Restricts possible content languages to a given set. Example: **Accept-Language: da, en-gb;q=0.8, en;q=0.7**
If-Modified-Since **If-Unmodified-Since**	Causes the response to depend on the requested resource last modification time/data.
If-Match **If-None-Match**	Causes the response to depend on the value for the ETag entity-header field. Messages with a body may define a **ETag** entity-header for the content (body) they carry.

HTTP/1.1 response-header fields

HTTP response messages specific. Some most often used response-header fields are:

Age	For cached replies, this is estimated elapsed time in seconds since the original response was obtained.
Location	This is used to redirect the requester to a different document from the one requested. Contains a document URL. Is used for **3xx** responses, for instance **307 Temporary Redirect**.
Public	Inform the client about server supported methods in general, not specifically on the requested URI. Example: **Public: OPTIONS, MGET, MHEAD, GET, HEAD**
Retry-After	Used in **503 Service Unavailable** response to inform about when the service is expected to be available, may be a date/time or a time period is seconds.
Server	A string identification for the server application. Example: **Server: CERN/3.0 libwww/2.17**
WWW-Authenticate	Used with **401 Unauthorized** response informing access to the resource requires user authentication. This field informs about the expected authentication mechanism to be used.
Set-Cookie	Contains a pair name and value for the client to use in the **Cookie** header field on subsequent requests. The purpose is the server being able to identify this particular client in next requests, and thus, maintain with it a stateful session. Example: **Set-Cookie: sessionToken=ts12325**

HTTP/1.1 – OPTIONS and GET methods

OPTIONS	Space	Argument (URI)	Space	HTTP/1.1	CR+LF

The response to this request provides the client with a list of available methods to access the URI, if the URI is an asterisk, then a list of methods supported by the server is provided.
The response will be usually **200 OK** and the response-header field **Allow** will l have a list of supported methods and eventually other header fields defining the server capabilities.

GET	Space	Argument (URI)	Space	HTTP/1.1	CR+LF

GET is used to obtain (download) the content pointed by URI. Typically, URI refers to a static content stored in a file, however, that may not be the case, it may also be dynamically generated by the server. Those are called dynamic contents. Dynamic contents are generated by applications running on the server side.
Often applications executed on the server side require input data to be provided by the client (e.g., collected in an HTML form). However, GET method requests can´t have a body, this is overcome by embedding form data in the URI, appending to it the query string.
The query-string starts by a question mark, and it's made of an ampersand separated list of pairs form field name and form field value. For instance:

http://www.server1.net/login?username=teste&password=pppttee&dep=5

A URI is obviously not the best-suited local to place data, only plain text data is supported and there are length issues. Beyond that, data will be visible in the URL. The POST method request is more suitable because it can have a body to carry data.
In lab classes, the technique known as CGI (Common Gateway Interface) will be used, it allows the HTTP server to execute external programs and return their output as response message content to clients.

HTTP/1.1 – HEAD, POST, PUT and DELETE methods

HEAD	Space	Argument (URI)	Space	HTTP/1.1	CR+LF

The response to the HEAD request is exactly the same that would be achieved with a GET request for the URI, except that it will have no body, nevertheless, all header fields will be the same.

POST	Space	Argument (URI)	Space	HTTP/1.1	CR+LF

The POST request purpose is sending data to an URI, this will normally be some kind of executable application. Unlike with the GET method, data is placed on the message body, therefore, there are no restrictions whatsoever on data content and length.

PUT	Space	Argument (URI)	Space	HTTP/1.1	CR+LF

The PUT request can be interpreted as the reverse of GET method. It allows the upload of a content to a URI. It is mostly intended to upload a content to a file named by the URI; however, it may also be used with the same purpose of POST if URI refers to an application.

DELETE	Space	Argument (URI)	Space	HTTP/1.1	CR+LF

Used to request the removal of a resource on the counterpart. The URI is supposed to represent the name of the file to be removed.

HTTP/1.1 – Response status-codes (1xx, 2xx, and 3xx)

HTTP responses status-codes can be grouped in five categories depending on the leftmost digit:

HTTP/1.1	Space	1XX	Space	Textual code description	CR+LF

Codes 1XX didn't exist in HTTP/1.0, they indicate some additional messages are expected over the same connection. For instance, **100 Continue** indicates the server has accepted the request first part and is expecting something else. The **101 Switching Protocols** is used when the server wants to upgrade to a higher HTTP version (**Upgrade** general-header field).

HTTP/1.1	Space	2XX	Space	Textual code description	CR+LF

Notify about a success on the requested operation. Examples:

200 OK – indicates total success on a GET, HEAD or POST.
201 Created – as result of the request a new resource has been created.
202 Accepted – the request was accepted, but may not have been yet executed, there may be a delay.
206 Partial Content – the content on the response body is only partial.

HTTP/1.1	Space	3XX	Space	Textual code description	CR+LF

Alert about a failure and the need for the client to reformulated the request. Examples:

300 Multiple Choices – there are several option to execute the request. A list is provided.
301 Moved Permanently – the resource was dislocated; the new location is provided by the **Location** field.
303 Moved Temporarily – temporary dislocation, the new location is provided by the **Location** field.
304 Not Modified – response to a conditional GET request when conditions are not meet.

HTTP/1.1 – Response status-codes (4xx and 5xx)

HTTP/1.1	Space	4XX	Space	Textual code description	CR+LF

Codes 4XX alert about what the server thinks it's a client-side request error. Examples:

400 Bad Request – the server simply did not understand the request made.
401 Unauthorized – the server demands user authentication for the request made.
403 Forbidden – the resource exists but is not accessible due to the lack of permission.
404 Not Found – the requested resource was not found in the server.
405 Method Not Allowed – the used method is not possible for the requested resource.
406 Not Acceptable – an Accept field restriction on the request could not be satisfied by the server.
411 Length Required – the server refuses to accept the request with no Content-Length specified.
412 Precondition Failed – an If field precondition on the request could not be satisfied by the server.

HTTP/1.1	Space	5XX	Space	Textual code description	CR+LF

These codes are about server-side issues, they mean the server is aware there is a problem and was unable to fulfill the request. Examples:

500 Internal Server Error – the server has a severe problem and was unable to process the request.
501 Not Implemented – the request method is not supported by the server.
503 Service Unavailable – the server was unable to process the request due to a temporary overload.
505 HTTP Version Not Supported – the request HTTP version is not supported by the server.

Persistent TCP connections

Under HTTP 1.0, TCP connections between the client and the server are presumed to be non-persistent, this means for each request/response one TCP connection is required and it's closed once the response is received.

HTTP 1.0 may optionally support persistent connections, to force that behavior, clients must include the **Connection: keep-alive** header line. If the server supports it, then it will also include the same header line on the response.

Under HTTP 1.1, TCP connections between the client and the server are presumed to be persistent, this means one TCP connection can be used for several request/response dialogues.
Even so, clients should include the **Connection: keep-alive** header line on their requests if they want to reuse the connection for further requests.

Persistent connections HTTP 1.1 behavior can be reverted to HTTP 1.0 behavior by adding the **Connection: close** header line. Clients using HTTP 1.1, and not supporting persistent connections must include this header line on every request. The server response will also include it, and the connection is then closed.

In principle, persistent connections are maintained until the client sends a request with the **Connection: close** header line. Then, the server response will also include the same header line and once the response is received by the client the connection is closed.

Persistent TCP connections keep alive timeout

In HTTP a persistent TCP connection can be used for several request/response dialogue sequences. If the client wants to close the connection after a request/response sequence it must include the **Connection: close** header line in the request.

Nevertheless, persistent connections don't persist indefinitely. For the sake of resources saving, both client and server applications define a **keep alive timeout**, if no request/response is sent during that time the connection is closed.

Default persistent connections a keep alive timeout for each application differs and may be an application configurable parameter.

Nevertheless, the **Keep-Alive:** header line can be included in requests and responses that contain the **Connection: keep-alive** header line.

This informs the counterpart about its current settings; two parameters are currently supported for the **Keep-Alive:** header line: **max** and **timeout**.

max specifies the maximum number of request/response sequences the connection supports (since it started), once that number is exhausted the connection is closed.

timeout specifies the number of seconds the connection is kept open with no traffic, if no request is sent within this time period, the connection is closed.

Example:

```
Connection: Keep-Alive
Keep-Alive: timeout=10, max=5
```

HTTPS (*Hyper Text Transfer Protocol Secure*) - HTTP over TLS (SSL)

HTTPS is not different from HTTP, it's the same protocol, but instead of running over plain TCP it runs over TLS (Transport Layer Security). TLS is the successor of Secure Sockets Layer (SSL), it provides secure network services for applications.
An HTTP client creates a TCP connection to the server and may then send the request. An HTTPS client creates a TCP connection to the server, secures it with TLS, and only then, can send the request.

To secure the connection, the client sends the **TLS ClientHello** message to the server. At this stage some, TLS messages are exchanged, the server's authenticity is assured by a **valid public key certificate** and a secret cryptographic key is then generated to encrypt data. Once the TLS handshake is finished, HTTP protocol can then be used, now requests and replies have guaranteed privacy.

Public key certificates have a critical role in HTTPS security, they give clients the guarantee they are talking with the authentic server and not a fake.

While the standard HTTP service port number is 80, for HTTPS it's port number 443. The browser will assume as default those port numbers by looking at the initial section the URL, correspondingly **http://** or **https://**. Default port numbers may be overridden if explicitly specified, for instance: **http://server.pt:8080**.

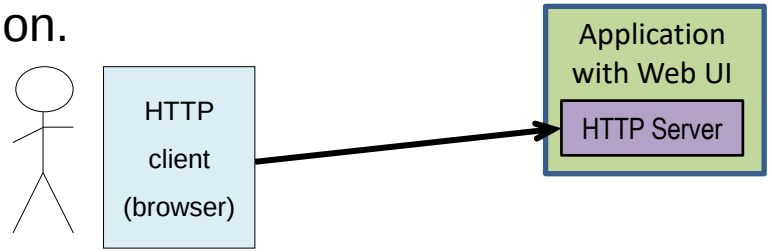
Web based user interface for applications (Web UI)

Developing a Graphical User Interface (GUI) for an application is an awfully time-consuming task, also, these applications consume significant hardware resources when running.

With HTTP available to transfer contents and browser applications ready to present HTML contents (and even interpret JavaScript), it's tempting to use these techniques to provide a friendly application's user interface, we can call that a Web UI.

To provide a Web UI, the application should include an HTTP server, it doesn't have to be a generic HTTP server supporting all HTTP functionalities, it only needs to support what's required by that specific application.

Once the application is running, the interacts with it by using his personal Web Browser. On most operating systems, the application can even request the opening of a local browser window to access the application.



Currently there are several frameworks available that include a web server and allow the development of Web UI applications without the need to implement the HTTP server from scratch.

URI (Uniform Resource Identifier)

To be addressable, resources spread over a network must be named, such name is generically called a URI. For a static content stored in a file, often the URI is the filename, optionally with path components to establish the location within the filesystem. For instance: **`/pages/info/document-list.html`**

Resources are stored in network nodes (acting as servers) so to uniquely identify the resource there's a special type of URI called URN (Uniform Resource Name) that includes the server's DNS name or IP address, the general form is:

`{SERVER}/{PATH-TO-RESOURCE}/{RESOURCE-NAME}`

Where {SERVER} is the node DNS name or IP address.

For instance: **`myserver.dei.isep.ipp.pt/pages/info/document-list.html`**

URL (Uniform Resource Locator)

An URL is a special URI that takes one step further by including not only the server's address and the resource name/path within the server, but also the way to access it. The general form of a URL is:

{ACCESS}://{SERVER}/{PATH-TO-RESOURCE}/{RESOURCE}

The {ACCESS} part of the URL specifies a mean of accessing the resource on the server, typically by using and appropriate content transfer protocol, like for instance HTTP, HTTPS or FTP.

In this case, the {SERVER} specification may also, optionally, include elements related to the access to the server:

[{USERNAME}@]{SERVER}[:PORT-NUMBER]

When no {USERNAME}@ is used, it's assumed the access doesn't require a login, if no :PORT-NUMBER is used, the default standard port number for the protocol is used, e.g. 80 for HTTP and 443 for HTTPS.

The Web Browser

One thing Web Browsers are required to do is interpret and use URLs, such URLs may be provided by the user (manually typed), or be part of loaded hypertext documents (e.g., references to images to be loaded, and live links for the user to click).

For a manually typed URL, most browsers accept a URN instead of a URL, they simply transform it into a URL by defaulting to HTTP. For instance, **www.server.com** becomes **http://www.server.com**.

Both manually typed URLs and those present in hypertext are processed by browsers by sending HTTP requests with the GET method. If no resource name is identified within the server, then it's assumed by browsers the resource name is / (the root path).

So, the URL **http://www.server.com** processing by the browser will result in sending the **GET /** request to node **www.server.com**.

There are very few cases where browsers send requests with methods other than GET. One case is when HTML forms are used, the submission method may be either GET or POST. The other case is when JavaScript is used to send HTTP requests to the server, this is called AJAX (Asynchronous JavaScript And XML) as we will see later.

The Web Browser, location and origin

When a page is loaded, the web browser establishes the **location**, it's the full URL from where the page was loaded, among others the location has one important property: **origin** is the access protocol and the server, e.g., http://www.dei.isep.ipp.pt:8080.

A reference included in an HTML page (e.g., images, scripts, links) is always a URI, but most often it's not a URL. When the reference is not a URL, the browser transforms it into an URL:

- If the URI starts by / (absolute path), then the browser uses the **origin** and adds the URI.
- If the URI doesn´t start by / (not an absolute path), then the browser uses the full location (original URL) to establish the new URL from the URI.

For instance, if a page is loaded from **http://serv.ex.com/testing/page1.html**, then the location will be **http://serv.ex.com/testing/page1.html**, and the origin propriety **http://serv.ex.com**. Then:

URI in the page	Corresponding URL to be used
/newpages/list.html	http://serv.ex.com/newpages/list.html
doc.html	http://serv.ex.com/testing/doc.html
http://s2.example.com/doc.html	http://s2.example.com/doc.html
../newpages/list2.html	http://serv.ex.com/newpages/list2.html

Using relative URIs in an HTML page makes all sense.

Why?

The Web Server

On the other side there's the Web Server, it's basically a HTTP/HTTPS server, but some Web Servers may understand other protocols like FTP.

The HTTP/HTTPS server role is responding to HTTP requests by clients (Web Browsers), nevertheless, some facets of a Web Server behavior are dependent on its configuration.

Most received HTTP requests have a URI as target, it identifies a resource on the server side. If the requested URI is absent on the Web Server configuration, then a **404 Not Found** or similar response is sent. This doesn't mean the resource is required to exist before the request, POST and PUT requests may be used to create new resources on the server, but that must be encompassed on the configuration.

Most Web Servers use the local filesystem folders' structure as a base to establish the resources naming path structure they will be providing (e.g., in Apache by setting the **Document Root**, pointing to a local folder, most often /var/www/html/).

When a client requests for the content of a folder (e.g., **GET /**), instead of a file (e.g., **GET /page1.html**), the content to be returned by the server depends on its configuration. For most HTTP servers a set of default filenames can be established, if one of such files exists in the folder, its content is returned. Usually, one of those filenames is **index.html**. One other option for the content to be return when a folder is requested, is a server generated page with the list of objects in the folder.