

# *RCOMP - Redes de Computadores (Computer Networks)*

*2025/2026*

*Lecture 09*

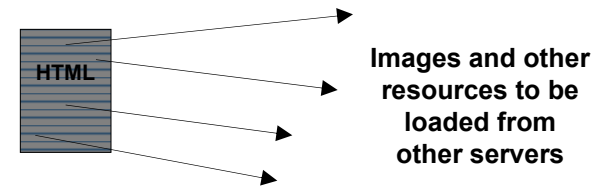
- The HTTP application protocol.

# Hypertext file transfer

Hypertext refers to documents with live links to other documents, this may mean directly clickable references (hyperlinks) or references to other resources to be included in the document's presentation, like for instance images.

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 the old FTP (File Transfer Protocol) can be used, it proved to be inappropriate for this kind of scenario. FTP requires one control connection for commands, 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 large number or relatively small files from different locations.

To overcome the FTP issues with hypertext, a content-oriented file transfer protocol has been developed, it's the Hypertext Transfer Protocol (HTTP).

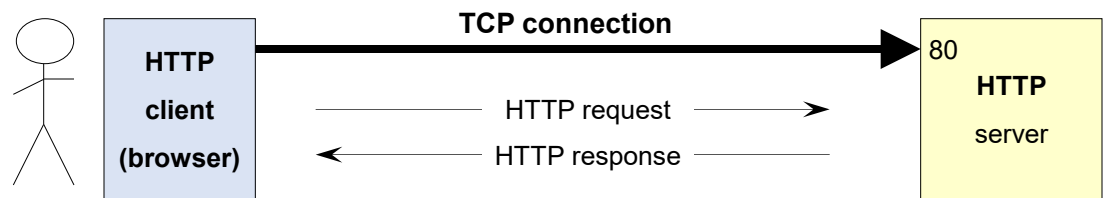
# Hypertext Transfer Protocol (HTTP)

Despite earlier versions, the first fully functional version, like those supported nowadays, appeared in 1996, named as HTTP 1.0. 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 and will provide that information to applications using HTTP.

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 for HTTP 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**.



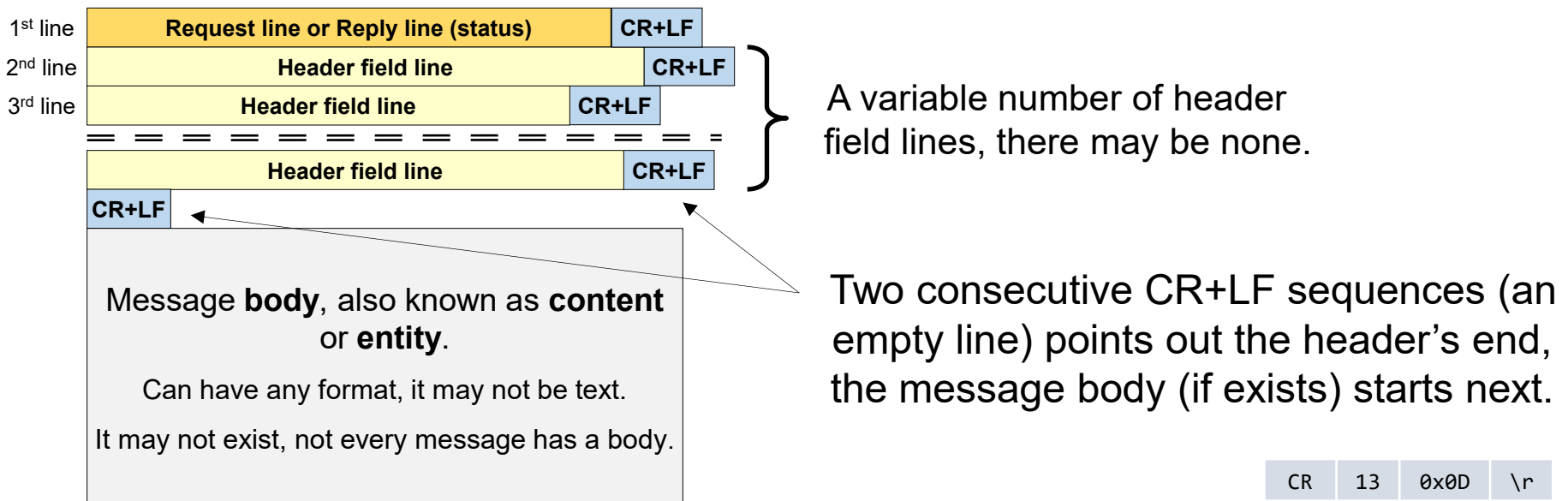
HTTP 1.0 defines GET, POST, and HEAD methods, earlier versions supported only GET.

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

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



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:



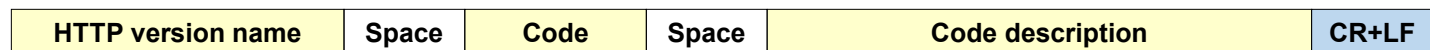
OPTIONS  
GET  
HEAD  
POST  
PUT  
DELETE  
TRACE  
CONNECT

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

- An asterisk – when 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 integer number.

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

# Header field lines

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

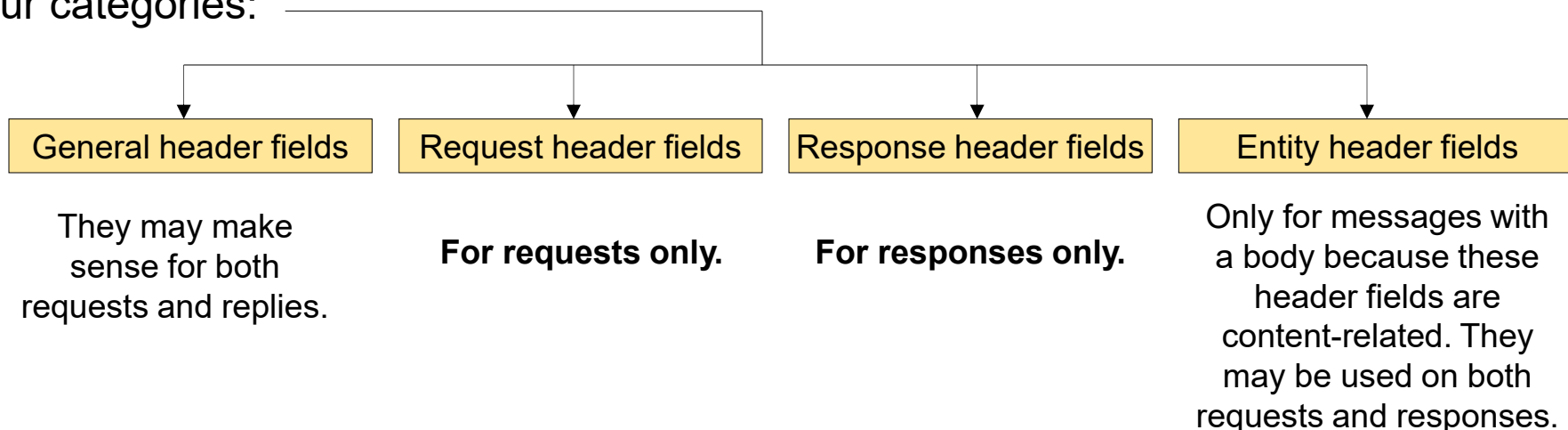
The general form is:



Where **Field name** is a standard case-insensitive identifier with special meaning for HTTP, it is 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 fields have been divided into four categories:



# HTTP 1.1 general-header fields

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

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

# 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:

<b>Allow</b>	Informs about supported methods to access a resource. It will be included in a <b>405 Method Not Allowed</b> error response.
<b>Content-Encoding</b>	Informs the message's receiver about some coding that was applied to the content, for instance: <b>Content-Encoding: gzip</b>
<b>Content-Language</b>	Describes the natural language of the intended audience for the content.
<b>Content-Length</b>	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.
<b>Content-MD5</b>	Holds the result of applying the Message Digest 5 algorithm to the content, used for content integrity checking.
<b>Content-Range</b>	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: <b>Content-Range: bytes 21010-47021/47022</b>
<b>Content-Type</b>	Informs the receiver about the content media, thus, how the content should be interpreted and ultimately displayed to the end-user.
<b>Expires</b>	Contains a date/time after which cached copies of the content are no longer valid.
<b>Last-Modified</b>	Contains the date/time the content was last modified. If the content comes from a file, it's often the file's last modification time.

# HTTP 1.1 request-header fields

Used on HTTP request messages only. Some most often used request-header fields are:

<b>From</b>	Contains the personal e-mail address of the user at the client application side.
<b>Host</b>	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: <b>Host: www.dei.isep.ipp.pt:8080</b>
<b>Referer</b> (yes, it's misspelled)	Contains the URL of the document from where the present request was followed (referred by). This field will not exist for directly user typed requests. Example: <b>Referer: http://www.dei.isep.ipp.pt/index.html</b>
<b>User-Agent</b>	A string identification for the client application, usually a browser. Example: <b>User-Agent: Mozilla/5.0 (Linux; Android 4.0.4; Galaxy Nexus Build/IMM76B)</b>
<b>Authorization</b>	User authentication data, usually username/password. Must be included following a <b>401 Unauthorized</b> error response.
<b>Cookie</b>	Contains a pair, name and value, provided by the counterpart in a previous response. Example: <b>Cookie: sessionToken=ts12325</b>

# HTTP 1.1 request-header fields – conditional requests

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

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

# HTTP 1.1 response-header fields

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

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

# 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 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
-----	-------	----------------	-------	----------	-------

Used to obtain (download) the content pointed by the URI. The URI may refer to a static content (e.g. stored in a file), it may also be a dynamic content, meaning it's dynamically generated by the server for each request. The URI may be associated to some code execution on the server side, such as on each request that code is executed.

For instance, the technique known as CGI (Common Gateway Interface), specifies how a standard server can execute external programs or scripts and return their output as response message content.

In many cases, dynamic contents require input data to be provided by the client (e.g. collected in an HTML form), however, a GET method request can't have a body to carry such data. This is overcome by appending to the URI a query-string. For instance:

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

A URI is obviously not the best-suited place to place data; only plain text data is supported and there are maximum length issues. One other problem is that data will be visible in the URL.

These issues are solved by using either the POST or PUT methods, in both cases the request may have a body, and thus it can be used to carry all sorts of data.

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

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

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

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

The POST request purpose is sending data to an URI; the URI will normally represent some code to be executed on the server side. Unlike with the GET method, data is placed on the message's body, therefore, there are no restrictions whatsoever on the data content type 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. Originally, it was intended to upload a content to a file named by the URI, however, it may also be used with similar purposes to POST, in such case the URI refers to some code to be executed on the server side.

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

Used to request the removal of a resource on the server side. The URI represents the name of the resource to be removed; it could be a file or something else.

# 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. For instance, the **101 Switching Protocols** is used when the server wants to upgrade to a higher HTTP version (see the **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, usually following a PUT request.

**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
----------	-------	-----	-------	--------------------------	-------

Alerts 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 requested 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 error on the request. Examples:

**400 Bad Request** – the server simply did not understand the request.

**401 Unauthorized** – the server demands user authentication to perform the requested operation.

**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 access to the 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 it has a problem and was unable to fulfill the request. Some examples are:

**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's 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 field line. If the server supports it, then it will also include the same header line in the response.

Contrariwise, 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 dialogue sequences.

Even so, to avoid any ambiguity, clients should include the **Connection: keep-alive** header field line in 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 field 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 immediately closed.

In principle, persistent connections are maintained until the client sends a request with the **Connection: close** header field line. 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

As mentioned, in HTTP a persistent TCP connection can be used for several request/response dialogue sequences. Under HTTP 1.1, 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 enforce a **keep alive timeout**, if no request/response is sent during that time, the connection is closed.

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

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

This informs the counterpart about its current settings; two parameters are currently supported for the **Keep-Alive:** header field, they are **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 (man-in-the-middle attack).

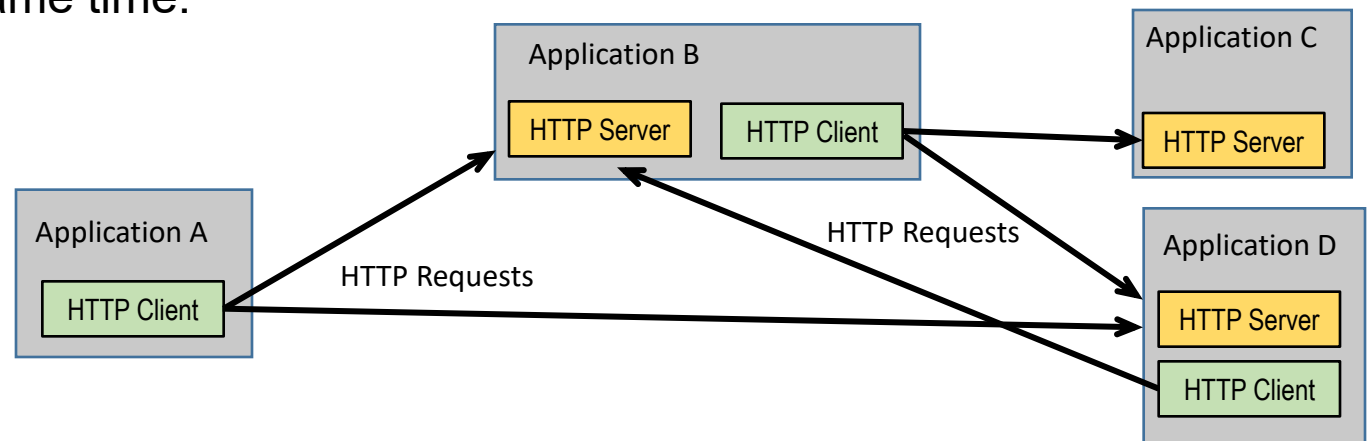
While the standard HTTP service port number is 80, for HTTPS it's port number 443. The browser will assume as default these port numbers by looking at the initial section the URL, correspondingly **http://** or **https://**. Default port numbers may be overridden if explicitly specified, for instance: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

# HTTP/HTTPS as general-purpose application protocol

Designing and implementing a new application layer protocol for some distributed applications architecture is a rather significant effort and investment. This investment could be avoided if an already existing application layer protocol was reused and eventually adapted.

We must bear in mind this may not be the best technical option; however, it may be the best option under investment point of view. Some adaptations to the original protocol usage may be required because it was designed with a different purpose, nevertheless, the protocol specification itself must be kept.

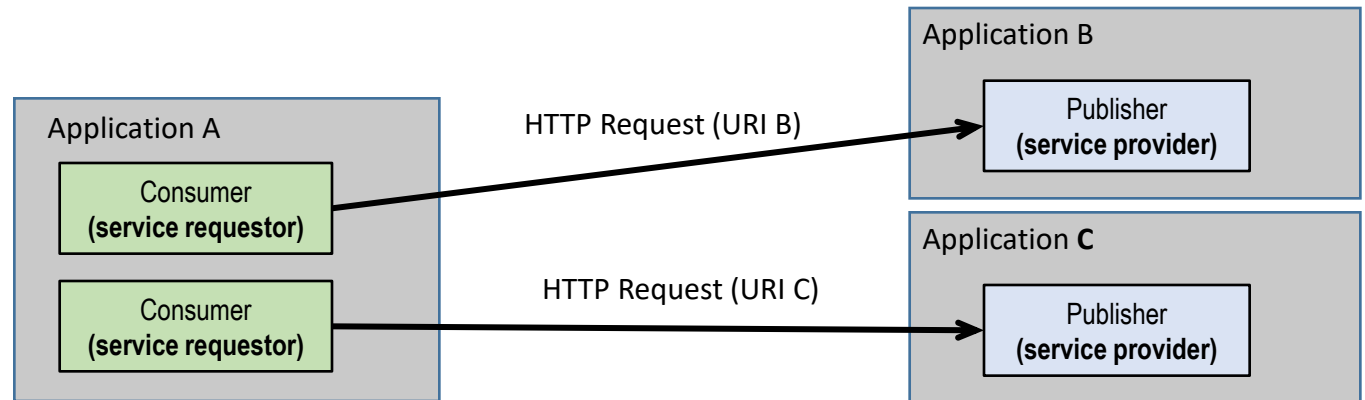
Because it's standard, simple and flexible, HTTP/HTTPS is widely adopted for this purpose. To achieve this, each application contains an HTTP server, an HTTP client, or both. The client-server model is preserved, though one application can be client and server at the same time.



# Web Services

The central concept on web services is the use of HTTP for application-to-application communications without direct human participation.

To make use of a web service, one application (**service requestor or consumer**) assumes the HTTP client's role and the other (**service provider or publisher**) the HTTP server's role. The web service is made available to service requestor applications by the service provider application.



From this central concept, some typical distributed systems issues rise. One issue is about **data representation**; it should be independent of individual local systems so that received data can be understood on any kind of node. Another issue is about **publishers' identification** by requestors, that will encompass the publisher's node address or DNS name, and also, the resource itself within that node address.

# Web Services – data representation

As it is, the web services concept is very wide. As far as the HTTP protocol is fulfilled and respected, all kind of information exchanges between applications can be implemented as web services.

Concerning data representation, the most widely used solutions are extensible markup language (XML), and JavaScript Object Notation (JSON).

Both represent data in a human readable text format, yet also suitable for automated parsing. In each case the appropriate content-type specification should be used, correspondingly, **application/xml** and **application/json**.

Also, some higher-level standards have been established on more details about how applications can communicate through web services, two examples are SOAP (Simple Object Access Protocol) and XML-RPC (Remote Procedure Calls in XML format through HTTP).

Due to the client-server model, implicit by HTTP, requestors must know where to find publishers, and then, what services are provided by that publisher.

# Web Services – resources identification

Resources are identified by an URL, an URL identifies a resource, and also, how to access it. So, an URL starts by an access protocol name, for web services **http://** or **https://**, then it identifies the node's address, usually through a DNS host name. Optionally it may also specify a port number preceded by a colon. If the port number is not specified, then the protocol's default port number is presumed.

What has been described is called the **origin** part of the URL. The remaining part of the URL identifies the resource within that origin; it starts by a slash and may reflect an internal hierarchical resources organization with names separated by a slashes.

Regarding the origin part, it shouldn't be hardcoded into applications because it depends on the running environment, they should be provided to applications as runtime configuration data. Each resource's local identification within the origin, on the other hand, may be hardcoded into requestor applications.

Usually, requestors know what resources are provided by a publisher, nevertheless, standards have been established on how publishers can inform requestors about that. Web Services Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI) make that information available to requestors in XML format.

# Web Services and Web Browsers

From the web services concept, which excludes direct end-users' interaction, it could be assumed web browsers are out of scope. Nevertheless, modern web browsers are themselves able to run applications, namely in JavaScript language. This makes them able to take part in web services architecture.

Current web browsers support the **XMLHttpRequest** object, in essence it's an HTTP client and allows a web page to, whenever it desires, make an HTTP request, retrieve data, and typically use that data to update parts of the page being displayed. This may be done without actually reloading the page, by using the HTML DOM (Document Object Model).

Requests with the **XMLHttpRequest** object should be asynchronous, this means, before triggering the request, a response handling function is defined. Then, the request itself will not block the web browser on waiting for the response, if, and when, the response arrives, the response handling function is executed in background by the browser.

This technique is called **AJAX** (Asynchronous JavaScript and XML), by using it, the traditional web pages' behavior, requiring a reload or submission for an update with fresh data from the server, is overcome.

# RESTful web services design model

REST stands for Representational State Transfer, it's a constrained resource-based usage for web services, main principles (constrains) are:

- Clients request operations over server-side resources (identified by URIs), such operations are: Create, Read, Update and Delete (CRUD), and they are mapped to HTTP request methods.
- Resource contents are transferred in XML, HTML or JSON representations.
- Servers are stateless in the sense they do not store information about clients' dialogue context. Therefore, on every request clients must provide all required context data.
- If the server has a state, then that state context must be represented by an addressable resource (URI), clients may then refer that state context on requests.

RESTful web services consumer applications can request the following four operations over a URI:

Operation	HTTP methods
Create a resource	POST; PUT
Read/retrieve a resource	GET
Update/Modify a resource	PUT
Delete/remove a resource	DELETE

# RESTful – resources and collections

The only **safe method** is GET, meaning it doesn't change the resource or the server state. Methods PUT, GET, and DELETE are regarded as **idempotent methods**; this means making more than one repeated identical request over the same URI has no additional effects beyond the effect of the first request.

A URI may refer to a single resource or a collection of resources, **singular names** are to be used for single resources, **plural names** for a collection of resources. Depending on being a single resource or a resources collection, HTTP methods will have slightly different meanings:

HTTP method	Single resource (singular name URI)	Resources collection (plural name URI)
<b>GET</b>	Retrieve the resource.	List the of resources items in the collection. Retrieved data is a list of resources' URIs and optionally other resources' data.
<b>PUT</b>	Replace the resource, if it does not exist, create it.	Replace the whole collection with another collection.
<b>POST</b>	<b>Not used</b> because the URI would be regarded as a collection and a new collection item would be created within it.	Create a new resource item within the collection. The new resource URI is automatically assigned.
<b>DELETE</b>	Delete the resource.	Delete the entire collection.

# RESTful - URI naming guidelines and best practices

- Singular names for single resources or items/elements belonging to a collection.
- Plural names for collections of resources.
- Verbs for controllers and functions.
- Camel casing for resources and lower case for URIs.
- Hyphens instead of underscores (spinal case instead of snake case).
- Avoid CRUD names (Create/Read/Update/Delete) in URIs.
- URI path elements should represent resources' hierarchical structure on the server side.
- Use URI path components to represent variable values.
- A query component may be added to the URI.

# Hypermedia As The Engine Of Application State (HATEOAS)

HATEOAS is a constraint of the REST application architecture. It means by accessing and retrieving a resource, a REST client also retrieves a **list of links** representing alternative actions from that point on. This is very similar to human web usage: when a web page is reached, several alternative links to follow from that point are provided.

This strategy makes the API discoverable by REST clients, though it's state dependent. Only after accessing a URI follow up links are provided, they represent possible state transitions from the initial state and may depend on the resource itself or other factors, like for instance user authentication used. Example using XML:

```
GET /accounts/1111 HTTP/1.1
Host: example.com
Accept: application/xml
...
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
  <account_number>1111</account_number>
  <balance currency="usd">100.00</balance>
  <link rel="deposit" href="/accounts/12345/deposit" />
  <link rel="withdraw" href="/accounts/12345/withdraw" />
  <link rel="transfer" href="/accounts/12345/transfer" />
  <link rel="close" href="/accounts/12345/close" />
</account>
```

```
GET /accounts/1112 HTTP/1.1
Host: example.com
Accept: application/xml
...
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
  <account_number>1112</account_number>
  <balance currency="usd">0.00</balance>
  <link rel="deposit" href="/accounts/12345/deposit" />
  <link rel="close" href="/accounts/12345/close" />
</account>
```

This is a classical example, because account 1112 has a zero balance, available actions are only deposit and close.

# Same-origin policy

Web browsers implement a security concept called **same-origin policy**. As we have seen, the origin is the URL part representing the protocol, hostname and port number. If a document loading into the browser requires the retrieval of resources from several different origins this is called **cross-origin**.

The **same-origin policy** requires all resources to be retrieved from the same origin and avoid possible malicious actions based on interactions between different origins.

**One thing this policy guarantees is that when a web service is being accessed by a browser, the access is being made from a page loaded from the web service's origin.**

There's no universal standard **same-origin policy** for browsers, usually this policy is enforced only for **JavaScript and other scripting languages**. Cross-origin for static embedded resources like script sources, images and other media files is allowed. Writing operations are usually allowed, including form submissions. Cross-origin read accesses are not allowed.

However, if within a page, a script accesses a REST API on a different origin, it's a clear same-origin policy violation and it will be blocked by the browser.

# Changing the origin

Within some restrictions, scripts are allowed to change the page's origin of the document. They are allowed to change the hostname part of the document origin as far as it's changed to an upper-level domain name (a super domain). In JavaScript this is done by setting the **document.domain** property.

Let's take for instance the following scenario:

- A page is loaded with origin **https://www.dei.isep.ipp.pt**
- This page has a script wanting to call the web service **https://isep.ipp.pt/users**. This is a same-origin policy violation and is going to be blocked by the browser.
- However, the script is allowed to change its page origin hostname from **www.dei.isep.ipp.pt** to **isep.ipp.pt**. Now there's no same-origin policy violation any longer.

Notice: some browsers may have issues/bugs around **document.domain** property handling.

# Cross-origin resource sharing (CORS protocol)

The **same-origin policy** enforced by web browsers can be overridden by using the CORS protocol. When a CORS capable web browser detects a cross-origin request it will look for specific header fields on the response to know if the access should or not be granted.

When a cross-origin request is made, the response may include an **Access-Control-Allow-Origin:** header line, it may be an “\*” authorizing all origins, or one specific origin, if it matches the calling origin then the browser will allow it, otherwise blocks it due to a same-origin policy violation.

If the **Access-Control-Allow-Origin:** header line returned by the server is “\*”, meaning any origin is allowed, or the origin value specified in the request, then access is granted by the browser, otherwise it's blocked.

For methods GET, HEAD and POST the query about access-control can be included in the request itself (Origin: header line). For other methods, a request with the OPTIONS method is triggered by the browser in the first place, this is called CORS pre-flight.

Access allowed by CORS is method dependent. Without a pre-flight, the allowed access applies only to the method used by the request.

# CORS pre-flight request

Under CORS, requests with methods GET, HEAD and POST are called simple requests, they require no preflight. Requests with methods PUT, DELETE, CONNECT, OPTIONS, TRACE and PATCH always require a pre-flight. Though, depending on other factors, even for simple requests, browsers may be required to perform a pre-flight.

The CORS preflight is a test to anticipate if the real request will be allowed and in what conditions. It's identical to the real request with the following differences:

- The **OPTIONS** method is used instead.
- The method to be used in the real request is declared on the **Access-Control-Request-Method**: request header line.
- Header lines to be included in the real request are declared at the **Access-Control-Request-Headers**: request header line.
- The **Origin**: header line contains the origin corresponding to the real request.
- Has no content.

The response to the preflight request lets the browser know if the real request should be allowed. Usually, the same persistent connection is used for the preflight and then for real request.

# CORS pre-flight response

The CORS pre-flight response holds a set of Access-Control response headers:

Response header line	
Access-Control-Allow-Origin:	Allowed origin specification, possibly equal to Origin: request header line or "*" (any origin allowed).
Access-Control-Allow-Methods:	A coma separated list of allowed methods, possibly including the requested one (Access-Control-Request-Method).
Access-Control-Allow-Headers:	A coma separated list of allowed header lines from the requested list (Access-Control-Request-Headers).
Access-Control-Max-Age:	For how long can this response be kept in cache.
Access-Control-Allow-Credentials:	Holds the <b>true</b> value if credentials can be used, otherwise this header is absent. Credentials can be cookies, authorization headers or TLS client certificates.
Access-Control-Expose-Headers:	A list of additional response headers to be made available to clients. Response headers Cache-Control, Content-Language, Content-Type, Expires, Last-Modified, and Pragma are always exposed.