HTTP: the language of Web communication

At times we use ☝ and 👇 to make it clear whether an explanation belongs to the code snippet above or below the text. The ‼ sign is added to code examples you should run yourself. When you see a 🐛, we offer advice on how to debug your code with the browser's and VSC's tooling - these hints are solely to help you with your programming project and not exam material! Paragraphs with a 🚩 are just for your information and not exam material.

Table of Contents

Required & recommended readings and activities

- Required readings: *none*
- Recommended activity:
    - 🎧 Listen to this podcast on how to learn new things quickly in the Web technology world.
    - 📺 A short video (in Dutch) by CWI (Centrum Wiskunde & Informatica) to celebrate 30 years since the Netherlands was connected to the public Internet!
- Recommended readings:
    - 📕 Chapters 1, 2 and 3 of HTTP: The Definite Guide (O'REILLY 2002).
    - MDN overview of HTTP.
    - Developer tools overview of your favorite web browser (e.g. Firefox or Chrome).
    - A brief history of HTTP.
    - Browser fingerprinting showcases how seemingly innocuous data can identify users.
    - A crash course on HTTP and DNS by Mozilla (among other things).
    - A detailed overview of the history of web standards and how to make use of them.
- Relevant scientific publications:
    - Ihm, S. and Pai, V.S., 2011. Towards understanding modern web traffic. In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (pp. 295-312). ACM.
    - Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., Papagiannaki, K. and Steenkiste, P., 2014. The cost of the S in HTTPS. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (pp. 133-140).
    - Falaki, H., Lymberopoulos, D., Mahajan, R., Kandula, S. and Estrin, D., 2010. A first look at traffic on smartphones. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (pp. 281-287).

Web standards

Image sourced from the linked video below.

Take a look at this video pitch from the World Wide Web Consortium, also known as the **W3C**: what are web standards and what makes web standards so important?

Learning goals

- Describe how web servers and clients interact with each other.
- Write HTTP messages that request web resources from web servers and understand the responses.
- Describe the different components of URLs and their purpose.
- Understand and employ HTTP authentication.

World Wide Web vs. Internet

A brief history of the web

The vision of the World Wide Web was already developed in the 1940s by Vannevar Bush, an American engineer who described his idea of a *memex* (a combination of memory and index) in the article As We May Think. The web can simply be described as **a system of interconnected hypertext documents, available via the Internet**.

In the 1960s, the first steps from vision to reality were made by DARPA, the *Defense Advanced Research Projects Agency* of the US department of defense. The so-called ARPANET was built for mail and file transfer and designed to withstand the loss of a portion of the network; as long as some connections remain, the remaining connected parties should still be able to communicate.

It took about 30 years before the Internet was opened to the public (in the late 1980s) and among the first non-military participants were universities and organizations such as CERN, the *European Organisation for Nuclear Research*. In fact, at CERN, Tim Berners-Lee **created** the World Wide Web: he was the first to successfully implement client-server communication on the Internet via the **hypertext transfer protocol** (or HTTP). **This protocol (and how it enables the web to function as we know it today) is what this lecture is all about.** Tim Berners-Lee remains an important figure in the web community today, in fact, he is the current director of the Word Wide Web Consortium.

In the early days of the web, browsers looked nothing like they do today; one of the earliest one was Lynx, a text-based browser that is functioning to this day. Here is how `google.com` and `amazon.com` are rendered on Lynx:

Browsers with graphical user interfaces started to appear in 1994, the front-runner being Netscape, quickly followed by Microsoft. The first version of Mozilla Firefox was released in 2002, Google Chrome started out in 2008. The late 90s and early 2000s were hampered by the so-called browser wars - the browser companies actively working against each other to gain a competitive advantage. Instead of adhering to a shared standard (as published by the Word Wide Web Consortium), different browser vendors implemented very different features and the labels *Best viewed with Netscape* or *Best viewed with Internet Explorer* were a common occurrence.

Key aspects of the Internet

The web is built on top of the Internet. The Internet describes the hardware layer: it is spanned from **interconnected computer networks around the globe that all communicate through one common standard**, the so-called TCP/IP protocol suite. The different sub-networks function autonomously, they do not depend on each other. There is not a single master - no machine or sub-network is in control of the whole network. It is very easy for machines or even entire sub-networks to join and leave the network without interrupting the flow of data among the remaining network. All devices interact with each other through **agreed-upon open standards** which are easy to use. These standards are implemented in a wide range of open-source server and client software.

To show how far we have come, here is the state of the Internet in 1973.

Two important organizations

The web and the Internet are not static, they are continuously changing. This development is led by two organizations:

- The Internet Engineering Task Force (**IETF**) leads the development of the Internet.
- The World Wide Web Consortium (**W3C**) leads the development of the web.

To many, the IETF is a lesser known organization, and while you may not often come across the IETF acronym, you will time and again encounter so-called RFCs. RFCs are **Request for Comments**, released by the IETF. They describe the Internet standards in detail. As an example, RFC 2822 is the document describing the Internet Message Format, aka email format, in about 50 pages.

HTTP messages

Important versions of the protocol are the following:

- **HTTP/0.9** was the first version the protocol (very limited in power, developed between 1989-1991).
- **HTTP/1.1** is governed by RFC 2068; it was standardized in 1997. HTTP/1.1 is the first **standardized** version of http.
- **HTTP/2** is governed by RFC 7540; it was standardized in 2015.

- **HTTP/3** has not been standardized yet.

HTTP/2 is the first new version of HTTP since HTTP/1.1. It originated at Google where it was developed as SPDY protocol (*speedy protocol*); more details here. As HTTP/1.1 is still the dominant protocol type on the web, we focus on it in this lecture.

We do not cover HTTP/3 in this lecture, for those interested, here is a history lesson in the form of a Twitter thread about how HTTP/3 came to pass and what issues of earlier HTTP versions it addresses.

Web servers and clients

On the web, clients and servers communicate with each other through **HTTP requests** and **HTTP responses**. If you open a web browser and type in the URL of your email provider, e.g. `https://gmail.com/` your web browser is acting as the **client** (sending an HTTP request). The **server** is your email provider (sending an HTTP response).

HTTP request and response

How does the communication between the two devices work? Servers wait for data requests continuously and are able to serve many client requests at the same time. Servers host **web resources**, that is any kind of content with an identity on the web. This can be static files, web services, but also dynamically generated content. As long as they are accessible through an identifier, they can be considered as web resources. The **client initiates the communication**, sending an **HTTP request** to the server, e.g. to access a particular file. The server sends an **HTTP response** - if indeed it has this file and the client is authorized to access it, it will send the file to the client, otherwise it will send an error message. The client, i.e. most often the web browser, will then initiate an action, depending on the type of content received - HTML files are rendered, music files are played and executables are executed. **HTTP proxies** are part of the Internet's infrastructure - there are many devices between a client and server that forward or process (e.g. filtering of requests, caching of responses) the HTTP requests and responses.

Network communication

Where does HTTP fit into the **network stack**, i.e. the set of protocols ("stacked" on top of each other) that together define how communication over the Internet happens? A very common representation of the network stack is the **OSI model**, the *Open Systems Interconnection model*:

Zimmermann's OSI model

Image sourced from the OSI reference model paper

It is a simplification of the true network stack, and today mostly a textbook model, but it shows the main idea of network communication very well. Network protocols are matched into different layers, starting at the bottom layer, the **physical layer**, where we talk about bits, i.e. 0s and 1s that pass through the physical network, and ending at the **application layer**, were we deal with **semantic units** such as video segments and emails.

Many network protocols exist, to us only three are of importance:

- the Internet Protocol (**IP**),

- the Transmission Control Protocol (**TCP**), and
- the HyperText Transfer Protocol (**HTTP**).

HTTP is at the top of the stack, and TCP builds on top of IP. Important to know is that HTTP is **reliable** - it inherits this property from TCP, which is reliable (in contrast to IP, which is not). This means, that the data appears **in order** and **undamaged**! This guarantee allows video streaming and other applications: HTTP **guarantees** that the video segments arrive at the client in the correct order; without this guarantee, all segments of a video would have to be downloaded and then assembled in the right order, before you could watch it!

🚩 Note, that this setup will change in HTTP/3, which resides on top of UDP, a protocol without guarantees of message delivery and packet order. HTTP/3 itself will be responsible to ensure reliable transmissions in a highly efficient manner (which in turn will lead to speedups over HTTP/1.1 and HTTP/2).

‼️ Activity

Open a modern browser and use its built-in **web development tools** to see what **HTTP messages** are exchanged when loading a web site. In this course, all tooling examples are using the Firefox Developer Tools; very similar tooling set exists for Chrome and Edge.

The Firefox Developer Tools can be reached from within the browser by heading to the toolbar and navigating to `Tools >> Web Developer`. There are several panels, we here take a look at the **Network panel** and how it looks after the user requested the web page residing at the URL https://www.tudelft.nl/:

Browser built-in web dev tools

☝️ You can see that the resource initially requested (`/`, i.e. the page residing at the URL https://www.tudelft.nl/) links to a myriad of additional web resources, which are then automatically requested by the web browser, leading to a **cascade of resource requests** (31 to be exact). Another vital piece of information when developing resource-intensive web applications is the timing information (broken down into different stages) available for each http request. It allows us to identify resources that are (too) slow to load. In this example we see that it takes more than three seconds to receive a response from the server when requesting the page residing at https://www.tudelft.nl/ and nearly five seconds to complete all requests. This is actually not a lot of resources; head over to a site like Volkskrant, NYTimes or the Guardian and look at the cascade of resource requests - at the end a few hundred resources will have been requested.

Each resource is requested through an **HTTP request**. How exactly such a request looks like can be seen in the *Headers panel* (which appears within the Network panel) when clicking on a particular resource row 👇 :

Browser built-in web dev tools

☝️ Note that above the request header is neatly organized, toggle the *Raw Headers* button to see how the request and response headers look in their raw format (i.e. how the header information is transferred).

Now that you have a first idea of what HTTP messages are about, let's dive into the details!

HTTP request message

Below is a typical HTTP request message:

```
GET / HTTP/1.1
Host: www.tudelft.nl
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:31.0)
Gecko/20100101 Firefox/31.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: __utma=1.20923577936111.16111.19805.2;utmcmd=(none);
```

HTTP/1.1 is a **plain text protocol** and **line-oriented**. The first line indicates what this message is about. In this case the keyword `GET` indicates that we are requesting something. The version number `1.1` indicates the highest version of HTTP that an application supports.

What are we requesting? Line 2 answers this question, we are requesting the web resource at `www.tudelft.nl`. The `host` header enables several domains to reside at the same IP address (i.e. server colocation). The client sending this request also provides additional information, such as which type of content it accepts, whether or not it is able to read encoded content, and so on.

In the last line, you can see that in this request, a cookie is sent from the client to server.

🚩 HTTP/2 (and subsequent versions) have been switched to a binary protocol.

HTTP response message

The server that received the above HTTP request may now assemble the following response:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2019 13:35:55 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 5994
Connection: keep-alive
Set-Cookie: fe_typo_user=d5e20a55a4a92e0; path=/; domain=tudelft.nl
[..other header fields..]
Server: TU Delft Web Server

[..body..]
```

Here, `[..xx..]` indicates other message content we are not interested in at the moment.

The first line indicates the status of the response. In this case, the requested **resource exists** and the client is authorized to access it. Thus, the server sends back the status `200 OK`: everything is okay, the resource was found, you are allowed to receive it. The response is then structured into **response header fields** in the `name:value` format, and the **response body** which contains the actual content. The body is optional - if the requested resource is not found, an error status code without a body would be returned to the client. The header fields contain important information for the client to understand the data being sent, including the type of content, the length and so on. Without this information, the client would be unable to process the data in the correct manner.

HTTP headers dissected

Well-known header fields

More than fifty header fields exist, a number of well-known ones (though to some extent this remains an arbitrary choice) are the following:

Header field	Description
Content-Type	Entity type
Content-Length	Length/size of the message
Content-Language	Language of the entity sent
Content-Encoding	Data transformations applied to the entity
Content-Location	Alternative location of the entity
Content-Range	For partial entities, range defines the pieces sent
Content-MD5	Checksum of the content
Last-Modified	Date on which this entity was created/modified
Expires	Date at which the entity will become stale
Allow	Lists the legal request methods for the entity
Connection & Upgrade	Protocol upgrade

The header fields **in bold** will be covered below in more detail. Let's briefly describe the other fields:

- `Content-Language` indicates the language the resource (also known as entity) is in, which can be English, Dutch or any other language.
- The `Content-Location` field can be useful if loading times are long or the content seems wrong; it can point to an alternative location where the same web resource resides.
- `Content-Range` is vital for entities that consist of multiple parts and are sent partially across different HTTP responses; without this information, the client would be unable to piece together the whole entity.
- `Allow` indicates to the client what type of requests can be made for the entity in question; `GET` is only one of a number of methods, it may also be possible to alter or delete a web resource.

Header field Content-Type

This header field informs the client (*for the purpose of this class the client is commonly a web browser*) what type of content is send. We use **MIME types** for this purpose.

MIME stands for *Multipurpose Internet Mail Extensions* (governed by RFCs 2045 and 2046) and was designed to solve problems when moving messages between electronic mail systems; it worked well and was adopted by HTTP to label its content.

MIME **types** determine how the client reacts - html is rendered, videos are played, and so on. The pattern is always the same: each MIME type has a **primary object type** and a **subtype**. Here are a few typical examples: `text/plain`, `text/html`, `image/jpeg`, `video/quicktime`, `application/vnd.ms-powerpoint`. As you can see in the `text/*` cases, the primary object type can have several subtypes.

Diverse MIME types exist. Below is a list of some of the most and least popular MIME types among 170K web resources sampled from a 2019 large-scale web crawl:

Most popular	Least popular
text/html	video/quicktime
application/xhtml+xml	text/rdf+n3
application/pdf	application/postscript
application/rss+xml	application/x-bzip2
image/jpeg	application/msword
application/atom+xml	application/coreldraw
text/plain	audio/midi
application/xml	application/vnd.ms-powerpoint
text/calendar	application/pgp-signature

You should be able to recognise most of the popular types apart from `application/rss+xml` and `application/atom+xml` - those are two popular types of web feed standards.

If a server does not include a specific MIME type in the HTTP response header, the default setting of `unknown/unknown` is used.

Header field Content-Length

This header field contains the **size of the entity body** in the HTTP response message. It has two purposes:

1. To indicate to the client whether or not the entire message was received. If the message received is less than what was promised, the client should make the same request again.

2. The header is also of importance for so-called **persistent connections**. Building up a TCP connection costs time. Instead of doing this for every single HTTP request/response cycle, we can reuse the same TCP connection for multiple HTTP request/response messages. For this to work though, it needs to be known when a particular HTTP message ends and when a new one starts.

Header field Content-Encoding

Content is often encoded, and in particular **compressed**. Four common encodings are:

- `gzip`
- `compress`
- `deflate`

- `identity` (this encoding indicates that no encoding should be used)

How do client and server negotiate acceptable encodings? If the server would send content in an encoding for which the client requires specific software that is not available, the client would receive a blob of data that it cannot interpret. To avoid this situation, the client sends in its HTTP request a list of encodings it can deal with. This happens in the `Accept-Encoding` HTTP request header, e.g. `Accept-Encoding: gzip, deflate`.

But why bother with encodings at all? If an image or video is compressed by the server before it is sent to the client, **network bandwidth is saved**. There is a **tradeoff**, however: compressed content needs to be decompressed by the client, which **increases the processing costs**.

Header field Content-MD5

Data corruption occurs regularly, the Internet spans the entire globe, billions of devices are connected to it. To route a message it has to pass through several devices, all of which run on software. And software is buggy (which in the worst case can lead to catastrophic errors).

MD5 stands for **message digest** and acts as a sanity check: the HTTP message content is hashed into a 128 bit value (the checksum). *Hashing* simply means that data of arbitrary size is mapped to data of fixed size in a deterministic manner. The MD5 checksum algorithm is described in RFC 1321. Once a client receives an HTTP response message that contains an MD5 checksum (computed by the server based on the message body), it then computes the checksum of the messages body as well and compares it with the server-provided value. If there is a mismatch, the client should assume that the content has been corrupted along the way and thus it should request the web resource again.

`Content-MD5` remains in use today as a simple checking mechanism (e.g. Amazon's S3 service relies on it), although it has been removed in the HTTP/1.1 revision of 2014, as indicated in RFC 7231, Appendix B:

```
The Content-MD5 header field has been removed because it was
inconsistently
implemented with respect to partial responses.
```

This example shows that once something is established (and entered protocol implementations), it is almost impossible to "take back" as the web has no centralized authority that can force its participating entities to adhere to a specific standard version and update its implementations accordingly. We will see this time and again in this course, especially once we start discussing the JavaScript language!

Header field Expires

Web caches make up an important part of the Internet. They cache **popular copies** of web resources. This reduces the load on the original servers that host these popular resources, reduces network bottlenecks and increases the responsiveness (web resources are delivered with less delay). But how does a web cache know for how long a web resource is valid? Imagine a web cache caching `nytimes.com` from the **origin server** (i.e. the server hosting `nytimes.com`) - this copy will quickly become stale and outdated. On the other hand, an RFC page that rarely changes may be valid in the cache for a long time.

This is where the `Expires` header field comes in. It indicates to a web cache when a fetched resource is no longer valid and needs to be retrieved from the origin server.

Web cache

Header field Cache-Control

There is another header that is similar to the `Expires` header: `Cache-Control`. For our purposes, the most important difference is the manner they indicate staleness to the web cache: `Expires` uses an **absolute expiration date**, e.g. *December 1, 2021*, while `Cache-Control` uses a **relative time**, `max-age=<seconds>` since being sent. If both header fields are set, `Cache-Control` takes precedence.

Enabling the origin server to fix in advance how quickly a cached version of a resource goes stale was an important design decision. The alternative would have been to solely rely on web caches to query the origin server to determine whether or not the cached resources are out of date - this would be inefficient though as these kind of enquiries would have to happen very frequently.

Here is an example of the header settings of https://www.theguardian.com/uk 👇 :

Cache-Control Guardian UK

Thus, the Guardian homepage goes stale after sixty seconds in a web cache, a sensible timing, given the nature of the news web site. You also see here that `Cache-Control` directives can contain more than just the seconds-until-stale though most of these directives are beyond the scope of this lecture (for more information, check the `Cache-Control` MDN page).

Finally, we note that modern browsers have an HTTP cache as well, which works analogously to the introduced web cache, relying on the same HTTP header fields to make caching decisions. The browser's cache can be considered as a *private cache* as it exists only on the local machine. The directive `private` in the Guardian's `Cache-Control` settings ☝️ tells the caches which ones are allowed to cache the response: in this case only the browser cache. In contrast, the directive `public` means that any cache can store a copy of the response.

Header field Last-Modified

The `Last-Modified` header field contains the date when the web resource was last altered. There is no header field though that indicates **how much** the resource has changed. Even if only a whitespace was added to a plain-text document, the `Last-Modified` header would change.

It is often used in combination with the `If-Modified-Since` header. When web caches actively try to revalidate web resources they cache, they only want the web resource sent by the origin server if it has changed since the `Last-Modified` date. If nothing has changed, the origin server simply returns a `304 Not Modified` response; otherwise the updated web resource is sent to the web cache.

`Last-Modified` dates has to be taken with a grain of salt. They are not always reliable, and can be manipulated by the origin server to ensure high cache validation rates for instance.

Header fields Connection & Upgrade

In HTTP/1.1 the client **always** initiates the conversation with the server via an HTTP request. For a number of use cases though this is a severe limitation. Take a look at these two examples from the New York Times website and Twitter respectively:

New York Times live polling

Twitter update

In both examples, the encircled numbers are updated *on the fly*, without the user having to manually refresh the page.

This can be achieved through **polling**: the client **regularly** sends an HTTP request to the server, the server in turn sends its HTTP response and if the numbers have changed, the client renders the updated numbers. This of course is a wasteful approach - the client might send hundreds or thousands of HTTP requests (depending on the chosen update frequency) that always lead to the same HTTP response.

An alternative to polling is **long polling**: here, the client sends an HTTP request as before, but this time the server holds the request open until new data is available before sending its HTTP response. Once the response is sent, the client immediately sends another HTTP request that is kept open. While this avoids wasteful HTTP request/response pairs, it requires the server backend to be significantly more complex: the backend is now responsible for ensuring the right data is sent to the right client and scaling becomes an issue.

Both options are workarounds to the requirement of **client-initiated** HTTP request/response pairs. The IETF recognized early on that such solutions will not be sufficient forever and in 2011 standardized the **WebSocket protocol** (RFC 6455). The RFC 6455 abstract read as follows 👇 :

```
The WebSocket Protocol enables two-way communication between a client
running untrusted code in a controlled environment to a remote host
that has opted-in to communications from that code.  The security
model used for this is the origin-based security model commonly used
by web browsers.  The protocol consists of an opening handshake
followed by basic message framing, layered over TCP.  The goal of
this technology is to provide a mechanism for browser-based
applications that need two-way communication with servers that does
not rely on opening multiple HTTP connections (e.g., using
XMLHttpRequest or <iframe>s and long polling).
```

WebSockets finally enable **bidirectional communication** between client and server! The server no longer has to wait for an HTTP request to send data to a client, but can do so *any time* - **as long as both client and server agree to use the WebSocket protocol**.

Client and server agree to this new protocol as follows: the client initiates the protocol *upgrade* by sending a HTTP request with at least two headers: `Connection: Upgrade` (the client requests an upgrade) and `Upgrade: [protocols]` (one or more protocol names in order of the client's preference). Depending on the protocol the client requests, additional headers may be sent. The server then either responds with `101 Switching Protocols` if the server agrees to this upgrade or with `200 OK` if the upgrade request is ignored.

For a concrete example, explore the HTTP request/response headers of the word guesser demo game. It relies on WebSockets to enable bidirectional communication between client and server. The browser's network panel allows you once more to investigate the protocol specifics 👇 :

Network monitor WebSockets

The client sends the following HTTP headers to request the upgrade to the WebSocket protocol:

```
Host: localhost:3000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:61.0)
Gecko/20100101 Firefox/61.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://localhost:3000
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key: ve3NDibwD/111x/ZKV0Phw==
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

As you can see, besides `Connection` and `Upgrade` a number of other information is sent, including the `Sec-WebSocket-Key`.

The server accepts the protocol with the following headers:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: b3yldD7Y6THeWnQGTJYzO1l4F3g=
```

The `Sec-WebSocket-Accept` value is derived from the hashed concatenation of the `Sec-WebSocket-Key` the client sent and the *magic string* `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` (i.e. a fixed string, as stated in the WebSocket RFC); if the server sends the correct `Sec-WebSocket-Accept` response, the client has assurance that the server actually supports WebSockets (instead of wrongly interpreting HTTP header fields).

Lastly it is worth to mention that besides switching to the WebSocket protocol, another common switch is from HTTP/1.1 to HTTP/2.

Status codes

To finish off this part about HTTP header fields, we take a look at the **response status codes**. You have already seen the `304` status code, sent by the origin server in the response after a request from a web cache enquiring about an updated copy of a web resource.

If you look at the first HTTP response example again, you will see that the status code is a very prominent part of the HTTP response - it appears in the first line of the response. In this case the status code is `200 OK`.

Different status codes exist that provide the client with some level of information on what is going on. Response status codes can be classified into five categories:

Status codes	
`1xx`	Informational
`2xx`	Success
`3xx`	Redirected
`4xx`	Client error
`5xx`	Server error

Status codes starting with 100 provide information to the client, e.g. `100 Continue` tells the client that the request is still ongoing and has not been rejected by the server.

Status code `200 OK` is the most common one - it indicates that the HTTP request was successful and the response contains the requested web resource (or a part of it).

Status codes in the three hundred range most often point to a redirect: a resource that was originally under URL `A` can now be found under URL `B`. These redirects are automatically resolved by the browser - you only notice a slightly longer loading time, otherwise redirects do not affect browser users. The network monitor shows you what exactly this delay amounts to 👇 :

Network monitor redirect

Here ☝️, status code `301 Moved Permanently` indicates that the resource at http://volkskrant.nl has been moved elsewhere for good. The `Location` header tells us the new location (https://volkskrant.nl).

Status codes starting with 4 indicate an error on the client side - most well known here is `404: Not Found`, i.e. the web resource or entity the client requests, does not exist on the server.

Errors on the server side start with 5; one relatively common status code is `502: Bad gateway`.

HTTP methods

Consider the first line of our introductory HTTP request message example:

```
GET / HTTP/1.1
```

So far, we have only seen `GET` requests, i.e. requests to get access to some web resource. `GET` however is only one of a number of **HTTP methods**.

Common HTTP methods

The following are the most common HTTP methods:

- `GET` you have already seen.
- `HEAD` returns the header of a HTTP response only (not the content)
- `POST` sends data from the client to the server for processing
- `PUT` saves the body of the request on the server; if you have ever used ftp you are already familiar with put
- `TRACE` can be used to trace where a message passes through before arriving at the server
- `OPTIONS` is helpful to determine what kind of methods a server supports and finally
- `DELETE` can be used to remove documents from a web server

This is not an exhaustive list of methods and not all servers enable or implement all the methods shown here.

‼️ Activity

Let's see how this protocol works in practice. One of the learning goals of this lecture is to be able to make HTTP requests yourself.

One tool to practice HTTP request writing is `telnet`. Telnet is a protocol defined in RFC 15; this low RFC numbering should tell you that it is a very old standard - it is from 1969.

Software that implements the client-side part of the protocol is also called telnet. Telnet **opens a TCP connection to a web server** (this requires a port number, for now just take this information as-is, you will learn more about port numbers in a bit) and anything you type into the telnet terminal is sent to the server. The server treats telnet as a web client and all returned data is displayed on the terminal.

While telnet is easy to use (as you will see in this activity), it is not capable of dealing with the https (i.e. secure http) protocol. Keep this in mind when you are trying this activity with web resources of your own. If you want to practice HTTP requests over https, you need to use openssl instead of telnet (this though is beyond the scope of this lecture).

Ok, back to telnet!

Try out the following examples yourself. Every line of the protocol is completed with a carriage return (i.e. press `<Enter>`). The protocol also has *empty lines*, those are indicated below with a `<carriage return>` tag (again, just press `<Enter>`). **All indented lines are returned by the server and do not have to be typed out.**

In order to close a telnet session, enter the telnet prompt (press `Ctrl` + `]`) and then use the `quit` command.

We are conducting our telnet activity on ard.de, the official homepage of one of Germany's largest public broadcasters - we simply use it as it is one of the most popular sites we are aware off that still offers content over http (instead of only over https).

Use `HEAD` to get information about the page

```
telnet ard.de 80
    Trying 83.125.35.3...
    Connected to ard.de.
    Escape character is ‘^]’
HEAD / HTTP/1.1
host:ard.de
<carriage return>
    HTTP/1.1 301 Moved Permanently
    [...]
    Location: http://www.ard.de/
```

Use `HEAD` to see what is at the moved location (and note that `ard.de` and `www.ard.de` are two different lcoations)

```
telnet www.ard.de 80
    Trying 23.34.184.239...
    Connected to e7671.e6.akamaiedge.net.
    Escape character is ‘^]’
HEAD / HTTP/1.1
host:www.ard.de
<carriage return>
    HTTP/1.1 301 Moved Permanently
    [...]
    Location: http://www.ard.de/home/ard/ARD_Startseite/21920/index.html
```

We continue to follow the new location (note the change in resource path now!)

```
telnet www.ard.de 80
    Trying 23.34.184.239...
    Connected to e7671.e6.akamaiedge.net.
    Escape character is ‘^]’
HEAD /home/ard/ARD_Startseite/21920/index.html HTTP/1.1
host:www.ard.de
<carriage return>
    HTTP/1.1 200 OK
    [...]
    Content-Type: text/html;charset=UTF-8
```

Use `GET` to retrieve the content (the printout on the terminal is not very insightful to us, it is just the HTML code of the requested resource)

```
telnet www.ard.de 80
    Trying 23.34.184.239...
    Connected to e7671.e6.akamaiedge.net.
    Escape character is ‘^]’
GET /home/ard/ARD_Startseite/21920/index.html HTTP/1.1
host:www.ard.de
<carriage return>
    HTTP/1.1 200 OK
    [...]
    Content-Type: text/html;charset=UTF-8
    [...]
```

Finally, for those that want to see to what extremes people go to make fun (or use) of telnet, try out the following

```
telnet towel.blinkenlights.nl
```

and wait for the movie to start. After a while you should see something like this ...

```
      .....                    @@@@@    @@@@@            ...........
      ......                  @     @  @     @           ..........
      .......                    @@@   @     @           .........
      ........                 @@      @     @            .......
       ........               @@@@@@@   @@@@@  th         ......
         .......            -----------------------       ......
           ......             C  E  N  T  U  R  Y          ....
             .....          -----------------------        ....
                ...         @@@@@ @@@@@ @   @ @@@@@        ...
                  ==          @   @      @ @    @          ==
                __||__        @   @@@@    @     @        __||__
               |      |       @   @      @ @    @       |      |
      _________|______|_____  @   @@@@@ @   @   @  _____|______|_________
```

From domain to IP address

Have you noticed something in the activity you just completed? We connected to the domain `ard.de` on port `80` and immediately got the response: `Trying 23.34.184.239`. This is called an **IP address** or *Internet Protocol address*.

The Internet maintains two principal namespaces: the **domain name hierarchy** and the **IP address system**. While domain names are handy for humans, the IP address system is used for the communication among devices.

The entity responsible for translating the domain into an IP address is called the **Domain Name System server** or DNS server. Several exist, a popular one is operated by Google, called Public DNS. Version 4 IP addresses (IPv4), just as the one shown above, consist of 32 bits; they are divided into 4 blocks of 8 bits each. 8 bit can encode all numbers between 0 and 255. This means, that in this format, a total of **2^32 unique IP addresses** or just shy of 4.3 billion unique IP addresses can be generated.

This might sound like a lot, but just think about how many devices you own that connect to the Internet ... This problem was foreseen already in the early 1990s and over time a new standard was developed by the IETF and published in 1998: the **IPv6** standard. An IPv6 address consists of 128 bit, organised into 8 groups of four hexadecimal digits. This means, that up to 2^128 unique addresses can be generated, that is such a large number that meaningful comparisons are hard to come by. In decimal form, 2^128 is a number with 39 digits! A large enough address space to last us almost forever.

Why are we still using IPv4? Because transitioning to the new standard takes time - a lot of devices require software upgrades (and nobody can force the maintainers to upgrade) and things still work, so there is no immediate sense of urgency.

Google keeps track of the percentage of users using its services through IPv6. As of late 2019 about 30% of users rely on IPv6, a slow and steady increase - it is just a matter of years until IPv4 is replaced by IPv6.

Uniform Resource Locators (URLs)

Let's now take a closer look at the format of *Uniform Resource Locators*, more commonly known by their abbreviation URLs. You are probably typing those into your browser at least a few times a day, let's see how well you know them! To get you started, here is a short quiz.

How many of the following URLs are valid?

- `mailto:c.hauff@tudelft.nl`
- `ftp://anonymous:mypass@ftp.csx.cam.ac.uk/gnu;date=today`
- `http://www.bing.com/?scope=images&nr=1#top`
- `https://duckduckgo.com/html?q=delft`
- `http://myshop.nl/comp;typ=c/apple;class=a;date=today/index.html;fr=delft`
- `http://правительство.рф`

Find out the answer!

URLs are the common way to access any resource on the Internet; the format of URLs is standardized. You should already be relatively familiar with the format of URLs accessing resources through HTTP and HTTPS. Resource access in other protocols (e.g. `ftp`) is similar, with only small variations.

In general, a URL consists of up to 9 parts:

```
<scheme>://<user>:<password>@<host>:<port>/<path>;<params>?<query>#<frag>
```

From back to front:

- `<frag>`: The name of a piece of a resource. Only used by the client - the fragment is not transmitted to the server.
- `<query>`: Parameters passed to gateway resources, i.e. applications [identified by the path] such as search engines.
- `<params>`: Additional input parameters applications may require to access a resource on the server correctly. Can be set per path segment.
- `<path>`: the local path to the resource

- `<port>`: the port on which the server is expecting requests for the resource (ports enable multiplexing: multiple services are available on one location)
- `<host>`: domain name (host name) or numeric IP address of the server
- `<user>:<password>`: the username/password (may be necessary to access a resource)
- `<scheme>`: determines the protocol to use when connecting to the server.

URL syntax: query

One of the most important URL types for us is the syntax for a `query`. What does that mean? Let's consider this URL:

```
https://duckduckgo.com/html?q=delft
```

This is an example of a URL pointing to the Duckduckgo website that - as part of the URL - contains the `q=delft` query. This query component is passed to the application accessed at the web server - in this case, Duckduckgo's search system and returned to you is a list of search results for the query `delft`. This syntax is necessary to enable interactive application. By convention we use `name=value` to pass application variables. If an application expects several variables, e.g. not only the search string but also the number of expected search results, we combine them with an `&`: `name1=value1&name2=value2&....`.

Answer: All URLs are valid. Return to the URL section.

Schemes: more than just HTTP(S)

`http` and `https` are not the only protocols that exist. `http` and `https` differ in their encryption - `http` does not offer encryption, while `https` does. `mailto` is the email protocol, `ftp` is the file transfer protocol. The local file system can also be accessed through the URL syntax as `file://<host>/<path>`, e.g. to view `tmp.html` in the directory `/Users/my_home` in the browser, you can use `file:///Users/my_home/tmp.html`.

Relative vs. absolute URLs

URLs can either be **absolute** or **relative**. Shown below are examples of both types:

Absolute (our base URL):

```
https://www.tudelft.nl/studenten/
```

Relative:

```
<h1>Resources</h1>
<ol>
    <li><a href="brightspace">Brightspace</a></li>
    <li><a href=" ../disclaimer">Disclaimer</a></li>
</ol>
```

Those relative URLs in combination with the base URL above lead to:

```
https://www.tudelft.nl/studenten/brightspace
https://www.tudelft.nl/disclaimer
```

An absolute URL can be used to retrieve a web resource without requiring any additional information. The two relative URLs by themselves do not provide sufficient information to resolve to a specific web resource. They are embedded in HTML markup which, in this case, resides within the web page pointed to by the absolute URL.

Relative URLs require a **base URL** to enable their conversion into absolute URLs. By default, this base URL is derived from the absolute URL of the web page the relative URLs are found in. The base URL of a resource is everything up to and including the last slash in its path name.

The base URL is used to convert the relative URLs into absolute URLs. The conversion in the first case (`brightspace`) is straightforward, the relative URL is appended to the base URL. In the second case (`../disclaimer`) you have to know that the meaning of `..` is to move a directory up, thus in the base URL the `/studenten` directory is removed from the base and then the relative URL is added.

Note, that this conversion from relative to absolute URL can be governed by quite complex rules, they are described in RFC 3986.

URL design restrictions

When URLs were first developed they had two basic design goals:

1. to be **portable across protocols**;
2. to be **human readable**, i.e. they should not contain invisible or non-printing characters.

The development of the Internet had largely been driven by US companies and organization and thus it made sense - at the time - to limit URL characters to the ASCII alphabet: this alphabet includes the Latin alphabet and additional reserved characters (such as `!`, `(`, `)`, etc.). The limitation is already apparent in the name: ASCII stands for *American Standard Code for Information Interchange* and thus heavily favours the English language.

Later, **character encoding** was added, e.g. a whitespace becomes `%20`. That means, that characters that are not part of ASCII can be encoded through a combination of ASCII characters. Character encodings are not sufficient though, what about languages that are not based on the Latin alphabet (what about URLs like `http://правительство.рф`)? Ideally, URLs should allow non-Latin characters as well, which today boils down to the use of unicode characters.

IETF comes once again to the rescue! **Punycode** (RFC 3492) was developed to allow URLs with unicode characters that are then translated uniquely and reversibly into an ASCII string. Quoting the RFC abstract:

```
Punycode is a simple and efficient transfer encoding syntax designed
for use with Internationalized Domain Names in Applications (IDNA).
```

```
It uniquely and reversibly transforms a Unicode string into an ASCII
string.  ASCII characters in the Unicode string are represented
literally, and non-ASCII characters are represented by ASCII
characters that are allowed in host name labels (letters, digits, and
hyphens).
```

The cyrillic URL example above transforms into the following ASCII URL: `http://xn--80aealotwbjpid2k.xn--p1ai/`. A URL already in ASCII format remains the same after Punycode encoding.

One word of caution though: **mixed scripts** (i.e. using different alphabets in a single URL) are a potential security issue! Consider the following URL: https://paypal.com. It looks like https://paypal.com, the well-known e-payment website. It is not! Notice that the Russian letter *r* looks very much like a latin *p* and a potential attacker can use this knowledge to create a fake paypal website (to gather credit card information) and lead users on with the malicious, but on first sight correctly looking paypal URL.

Authentication

The last topic we cover in this first lecture is authentication.

Authentication is any process by which a system verifies the identity of a user who wishes to access it.

So far, we have viewed HTTP as an anonymous and **stateless** request/response protocol. This means that the same HTTP request is treated in exactly the same manner by a server, independent of who or what entity sent the request. We have seen that each HTTP request is dealt with independently, the server does not maintain a state for a client which makes even simple forms of tracking (e.g. how often has a client requested a particular web resource) impossible.

But of course, this is not how today's web works: servers **do** identify devices and users, most web applications indeed track their users very closely. We now cover the mechanisms available to us on the web to identify users and devices. We consider four options:

- HTTP headers
- client IP addresses
- user login
- fat URLs

If you already know a bit about web development you will miss in this list cookies and sessions. We cover these concepts in a later lecture.

We now cover each of the four identification options listed above in turn.

Authentication by user-related HTTP header fields

The HTTP header fields we have seen so far were only a few of all possible ones. Several HTTP header fields can be used to provide information about the user or her context. Some are shown here:

Request header field

Request header field	
`From`	User's email address
`User-Agent`	User's browser
`Referer`	Resource the user came from
`Client-IP`	Client's IP address
`Authorization`	Username and password

All of the shown header fields are request header fields, i.e. sent from the client to the server.

The first three header fields contain information about the user such as an email address, the identifying string for the user's device (though here device is rather general and refers to a particular type of mobile phone, not the specific phone of this user), and the web page the user came from.

In reality, users rarely publish their email addresses through the `From` field, this field is today mostly used by web crawlers; in case they break a web server due to too much crawling, the owner of the web server can quickly contact the humans behind the crawler via email. The `User-Agent` allows device/browser-specific customization, but not more. Here is a concrete example of the `User-Agent` a Firefox browser may send: `Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:68.0) Gecko/20100101 Firefox/68.0`. A number of services (e.g. userstack) exist that provide interpretations of the user-agent string. The `Referer` is similarly crude: it can tell us something about a user's interests (because the user had just visited that resource) but does not enable us to uniquely identify a user.

To conclude, the HTTP headers `From`, `Referer` and `User-Agent` are not suitable to track the modern web user. We cover the `Client-IP` header next and `Authorization` in a later section.

Authentication by Client-IP address tracking

We can also authenticate users via client IP address tracking, either extracted from the HTTP header field `Client-IP` or the underlying TCP connection. This would provide us with an ideal way to authenticate users **IF** every user would be assigned a distinct IP address that rarely or never changes.

However, we know that IP addresses are assigned to machines, not users. Internet service providers do not assign a unique IP to each one of their user-device combinations, they dynamically assign IP addresses to users' devices from a common address pool. A user device's IP address can thus change any day.

Today's Internet is also more complicated than just straightforward client and servers. We access the web through firewalls which obscure the users' IP addresses, we use proxies and gateways that in turn set up their own TCP connections and come with their own IP addresses.

To conclude, in this day and age, IP addresses cannot be used anymore to provide a reliable authentication mechanism.

Authentication by fat URLs

That brings us to fat URLs. The options we have covered so far are not good choices for authentication today, fat URLs on the other hand are in use to this day.

The principle of fat URLs is simple: users are tracked through the generation of **unique URLs for each user**. If a user visits a web site for the first time, the server recognizes the URL as not containing a *fat element* and assumes the user has not visited the site before. It generates a unique ID for the user. The server then redirects the user to that fat URL. Crucially, in the last step, the server **on the fly rewrites the HTML** for every single user, adding the user's ID to each and every hyperlink. A rewritten HTML link may look like this: `<a href="/browse/002-1145265-8016838">Gifts</a>` (note the random numbers string at the end).

In this manner, different HTTP requests can be tied together into a single **logical session**: the server is aware which requests are coming from the same user through the ID inside the fat URLs.

Let's look at this concept one more time, based on the following toy example 👇 :

Fat URL toy example

On the left ☝️ you see a shop web site, consisting of the entry page `my-shop.nl` and two other pages, one for books and one for gifts. The entry page links to both of those pages. These URLs do not contain a fat element. The first time a user requests the entry page `my-shop.nl`, the server recognizes the lack of an identifier in the URL and generates one. Specifically for that user, it also rewrites the HTML of the entry page: its hyperlinks now contain the unique ID. The server then redirects the user to `my-shop.nl/43233` and serves the changed HTML content. In this manner, as long as the user browses through the shop, the user remains authenticated to the server. This authenticaion approach is still to weak though as the user can simply navigate to `my-shop.nl` again and to receive a new unique identifier.

Fat URLs have issues:

1. First of all, they are ugly, instead of short and easy to remember URLs you are left with overly long ones.
2. Fat URLs should not be shared - you never know what kind of history information you share with others if you hand out the URLs generated for you!
3. Fat URLs are also not a good idea when it comes to web caches - these caches rely on the *one page per request* paradigm; fat URLs though follow the *one page per user* paradigm.
4. Dynamically generating HTML every time a user requests a web resource adds to the server load.
5. All of this effort does not completely avoid loosing the user: as soon as the user navigates away from the web site, the user's identification is lost.

To conclude, fat URLs are a valid option for authentication as long as you are aware of the issues they have.

Authentication by HTTP basic authentication

Let's move on to the final authentication option we consider here: **HTTP basic authentication**. You are already familiar with this type of authentication: the server asks the user **explicitly** for authentication by requesting a valid username and a password. HTTP has a built-in mechanism to support this process through the `WWW-Authenticate` and `Authorization` headers. Since HTTP is **stateless**, once a user has logged in, the login information has to be resend to the server with every single HTTP request the user's device is making.

Here is a concrete example of HTTP basic authentication 👇 :

Basic authentication example

We have the usual server and client setup ☝️ . The client sends an HTTP request to access a particular web resource, in this case the `index.html` page residing at `www.microsoft.com`.

The server sends back a `401` status code, indicating to the client that this web resource requires a login. It also sends back information about the supported authentication scheme (in this case: `Basic`). There are several authentication schemes, but we will only consider the basic one here. The *realm* describes the protection area: if several web resources on the same server require authentication within the same realm, a single user/password combination should be sufficient to access all of them.

In response to the `401` status code, the client presents a login screen to the user, requesting the username and password. The client sends username and password encoded (**but not encrypted**) to the server via the HTTP `Authorization` header field.

If the username/password combination is correct, and the user is allowed to access the web resource, the server sends an HTTP response with the web resource in question in the message body.

For future HTTP requests to the site, the browser **automatically sends along** the stored username/password. It does not wait for another request.

As just mentioned, the username/password combination are encoded by the client, before being passed to the server. The **encoding scheme** is simple: the username and password are joined together by a colon and converted into **base-64 encoding** (described in detail in RFC 4648). It is a simple *binary-to-text* encoding scheme that ensures that only HTTP compatible characters are entered into the message.

For example, in base-64 encoding `Normandië` becomes `Tm9ybWFuZGnDqw==` and `Delft` becomes `RGVsZnQ=`.

It has to be emphasized once more that encoding has nothing to do with encryption. The username and password sent via basic authentication can be decoded trivially, they are sent over the network *in the clear*. This by itself is not critical, as long as users are aware of this. However, users tend to be lazy, they tend to reuse the same or similar login/password combinations for a wide range of websites of highly varying criticality. Even though the username/password for site `A` may be worthless to an attacker, if the user only made a slight modification to her usual username/password combination to access site `B`, let's say her online bank account, the user will be in trouble.

Overall, basic authentication is the best of the four authentication options discussed; it prevents accidental or casual access by curious users to content where privacy is desired but not essential. Basic authentication is useful for personalization and access control within a friendly environment such as an intranet.

In the wild, i.e. the general web, basic authentication should only be used in combination with secure HTTP (most popular variant being https with URL scheme `https`) to avoid sending the username/password combination in the clear across the network. Here, request and response data are encrypted before being sent across the network.

Secure HTTP

So far we have seen *lightweight authentication* approaches. Those are not useful for bank transactions or confidential data. Secure HTTP should provide:

- Server authentication (client is sure to talk to the right server)
- Client authentication (server is sure to talk to the right client)
- Integrity (client and server are sure their data is intact)
- Encryption
- Efficiency
- Adaptability (to the current state of the art in encryption)

HTTPS is the most popular secure form of HTTP. The URL scheme is `https` instead of `http`. Now, request and response data are **encrypted** before being sent across the network. In the layered network architecture, an additional layer is introduced: the Secure Socket Layer (SSL):

HTTPS

Note, that client and server have to **negotiate** the cryptographic protocol to use (the most secure protocol both sides can handle). The encryption employed is only as secure as the weaker side allows: if the server has the latest encryption protocols enabled but the client has not been updated in years, a weak encryption will be the result.

Self-check

Here are a few questions you should be able to answer after having followed the lecture:

1. What are the main advantage and disadvantage of using compression?
2. What is the main problem of HTTP's plain text format compared to a binary format?
3. What is commonly compressed: HTTP headers and/or HTTP responses? Why?
4. In what circumstance might `HEAD` be useful?
5. In what order do the following operations occur when you enter http://www.microsoft.com in the browser's address bar?
   - Convert domain to IP address
   - Send HTTP request
   - Receive HTTP response
   - Establish a TCP connection
6. What is the main advantage of relative URLs over absolute URLs?
7. What is the main weakness of URLs as they are in use today?
   - URLs are unnecessarily long.
   - We are running out of URL space for ASCII-based URLs.
   - URLs point to a location instead of a web resource.
   - URLs point to a web resource instead of a location.
8. Which of the following statements about the hypertext transfer protocol are TRUE?
   - `HEAD` can be used to determine whether a given URL refers to an existing web resource.
   - The HTTP header field `Last-Modified` is used in an HTTP request that informs the server of the client's latest version of a given web resource.
   - The information retrieved via `HEAD` can also be retrieved via `GET`.
   - The `Content-Length` header is used in an HTTP request to inform the server which parts of a web resource a client wants to receive.
9. Which of the following statements about web caches are TRUE?

- Web caches increase the processing power of origin servers.
- Web caches are the web's backup: they keep a copy of every resource on the web.
- Web caches rely on the `Content-Range` header field to determine when a copy becomes invalid.
- Web caches lead to reduced distance delay.

10. Which of the following statements about IPv6 are FALSE?
    - IPv6 has approximately ten times as much address space available as IPv4.
    - Most Internet traffic today makes use of IPv6 (instead of IPv4).
    - IPv6 addresses do not have an associated domain name in the Domain Name System registry.
    - IPv6 addresses are 128 bit long.
11. What issues do fat URLs have?
12. What is the purpose of HTTPS?