CSE1500 Web Technology Final

Contents

Week 6. HTTP lecture, HTML and web design lecture; first web technology assignment

Lecture 1. HTTP: the language of web communication

Web standards

Web standards are the formal, non-proprietary standards and other technical specifications that **define and describe aspects of the World Wide Web**. In recent years, the term has been more frequently associated with the trend of endorsing a set of **standardized best practices** for building web sites, and a **philosophy** of web design and development that includes those **methods**.

World Wide Web vs The Internet

The **web is a system of interconnected hypertext documents, available via 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.

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

RFCs are **Request for Comments**, released by the IETF. They describe the **Internet standards** in detail. (i.e. RFC 2822 email format standard)

HTTP messages

Hypertext transfer protocol.

- HTTP/0.9 was the first version of 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, though an RFC draft already exists.

HTTP/1.1 is still the dominant protocol version on the web, we focus on it in this lecture.

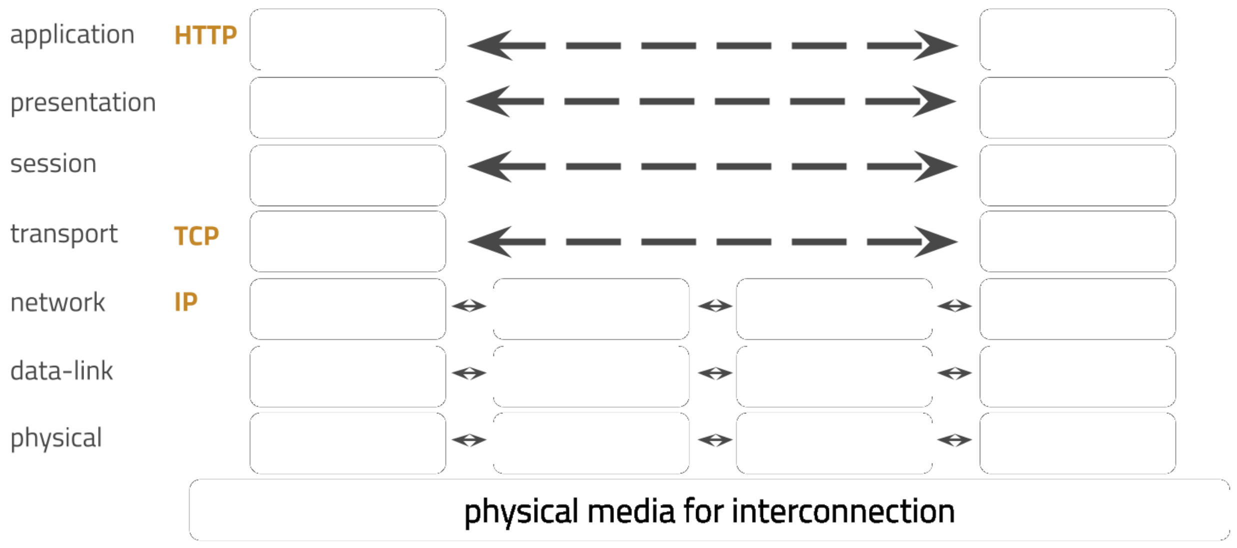
Web servers and clients

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. Your email provider is acting as the server, sending an HTTP response.

The client always 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

i.e. the set of protocols ("stacked" on top of each other) that together define how communication over the Internet happens. A common representation of the network stack is the Open Systems Interconnection model (or OSI model)



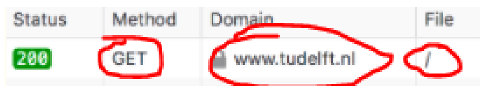
For our purposes the two outer stacks can be considered client and server, the middle ones are HTTP proxies. An HTTP message travels down the network stack on the device being transformed in every layer potentially into multiple messages which are then transmitted via the physical network. At the other end, these messages travel up the device's network stack again, being transformed in every layer and then at the final layer the HTTP message is reassembled.

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:

- Internet Protocol (IP)
- Transmission Control Protocol (TCP)
- HyperText Transfer Protocol (HTTP)

HTTP is reliable (like TCP) 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!



Headers

When you visit a website via a browser, the first "resource" is the url itself, (File: /), which can link to multiple other resources, leading to a cascade of resource requests. Each resource is requested through an HTTP request.

HTTP request message

HTTP request messages are always sent by the client. Below is a typical HTTP request message (the numbering of lines is not part of the actual message):

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

HTTP/1.1 is a plain text protocol and line-oriented.

1. 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.
2. We are requesting the web resource at www.tudelft.nl We need the host header, because it enables several domains to reside at the same IP address.

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 response message

Sent by the server, a typical response message:

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

1. 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.

Common header fields

Header field	Description
Content-Type	Entity type
Content-Length	Length/size of the message
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 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 what type of content is being sent. We use MIME types for this purpose. MIME stands for Multipurpose Internet Mail Extensions (governed by RFCs 2045 and 2046).

MIME types determine how the client reacts: html is rendered, videos are played, and so on. Each MIME type has a primary object type and a subtype. Here are a few typical examples

Popular MIME types:

- text/html
- application/xhtml+xml
- application/pdf
- application/rss+xml
- image/jpeg
- application/atom+xml
- text/plain
- application/xml
- text/calendar

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**:

- 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.
- 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, that is The client only understands content that is not compressed)

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

Encoding pro: If an image or video is compressed by the server before it is sent to the client, **network bandwidth is saved**.

Encoding con: compressed content needs to be decompressed by the client, which increases the **processing costs**.

Header field Content-MD5

MD5 stands for message digest and acts as **a sanity check**: the HTTP message content is hashed into a 128 bit value (the checksum). It's used to check if there is a mismatch between the server and the client checksum, if so the client may 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 although **it has been removed in the HTTP/1.1 revision of 2014**.

Header field Expires

It indicates to a web cache when a fetched resource is no longer valid and needs to be retrieved from the origin server.

Header field Cache-Control

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**.

There are 2 types of cache browser cache (private) and any other type (public). A browser's HTTP cache is useful as it reduces the load on the origin server; for instance, a click on the browser's back button does not typically result in a new HTTP request, instead the cached resource is served.

A browser's HTTP cache is at times the source of immense frustration for web developers. If you are updating the code of your web application, deploying it and testing it in your browser, it may appear to not have any effect. And then a round of debugging starts. However, there may be nothing wrong with your code, instead your browser may be relying on the cached version of your web application source code files. For development purposes, the browser's HTTP cache should be switched off.

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 returns a 304 Not Modified response; otherwise the updated web resource is sent to the web cache.

Last-Modified dates have 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. This in turn indicates that the origin server is serving novel content regularly which is rumored to help search engine rankings (it is only a rumor as search engines continuously update their result ranking algorithms).

Header fields Connection & Upgrade

In HTTP/1.1 the client always initiates the conversation with the server via an HTTP request. But in some occasions (think of a live stock price) content is 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.

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. *This avoids wasteful constant polling but increases server back-end complexity.*

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).

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]

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.

An example below:

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

They appear in the first line of the response.

Status codes' categories

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. 101 status code indicates that the server is switching to the protocol setup requested by the client.

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 starting with 4 indicate an error on the client side - most well known here is 404: Not Found, that is, 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

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.

From domain to IP 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 a domain name into an IP address is called the Domain Name System server or DNS server; it is essentially a world-wide distributed directory service that needs to be continuously updated and synced.

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.

An IPv6 address consists of 128 bit, organized into 8 groups of four hexadecimal digits. This means, that up to 2^128 unique addresses can be generated.

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 August 2020 about 33% 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)

More commonly known by their abbreviation "URLs", a URL consists of up to 9 parts:

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

- <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

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. 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 ampersand (&): name1=value1&name2=value2& ...

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
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 /students directory is removed from the base and then the relative URL is added.

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:

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

At first they only supported ASCII characters but later adopted Unicode characters. Punycode (RFC 3492) was developed to allow URLs with unicode characters to be translated uniquely and reversibly into an ASCII string.

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! Don't take our word for it, try it out in your own browser. 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

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

Let's consider four options:

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

Authentication by user-related HTTP header fields

Several HTTP header fields can be used to provide information about the user or her context. Such as:

Request header field	Description
`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

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 (and it is not always reliable). 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

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.

The HTTP headers From, Referer and User-Agent are not suitable to track the modern web user.

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, 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.

This authentication approach is still to weak though as the user can navigate to my-shop.nl again and to receive a new unique identifier.

Fat URLs have issues:

- First of all, they are “ugly”, instead of short and easy to remember URLs you are left with overly long ones.
- 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!
- 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.
- Dynamically generating HTML every time a user requests a web resource adds to the server load.
- 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.

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 Unauthorized 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

The previous approaches are not useful for bank transactions or confidential data. Secure HTTP provides:

- server authentication (client is sure to talk to the right server);
- client authentication (server is sure to talk to the right client);
- integrity (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). It sits below the HTTP layer and above the TCP layer.

Importantly, 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.

Lecture 2. HTML: the language of the web

Required readings HTML – Intro (Skeleton)

https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started

Getting started with HTML

In this article we cover the absolute basics of HTML. To get you started, this article defines elements, attributes, and all the other important terms you may have heard. It also explains where these fit into HTML. You will learn how HTML elements are structured, how a typical HTML page is structured, and other important basic language features. Along the way, there will be an opportunity to play with HTML too!

Prerequisites:	Basic computer literacy, basic software installed, and basic knowledge of working with files.
Objective:	To gain basic familiarity with HTML, and practice writing a few HTML elements.

What is HTML?

HTML (Hypertext Markup Language) is not a programming language. It is a *markup language* that tells web browsers how to structure the web pages you visit. It can be as complicated or as simple as the web developer wants it to be. HTML consists of a series of elements, which you use to enclose, wrap, or *mark up* different parts of content to make it appear or act in a certain way. The enclosing tags can make content into a hyperlink to connect to another page, italicize words, and so on. For example, consider the following line of text:

My cat is very grumpy

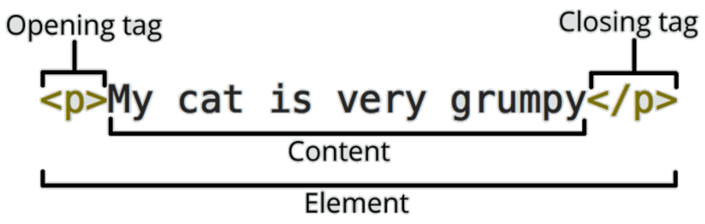
If we wanted the text to stand by itself, we could specify that it is a paragraph by enclosing it in a paragraph (<p>) element:

<p>My cat is very grumpy</p>

Note: Tags in HTML are case-insensitive. This means they can be written in uppercase or lowercase. For example, a <title> tag could be written as <title>, <TITLE>, <Title>, <TiTlE>, etc., and it will work. However, it is best practice to write all tags in lowercase for consistency, readability, and other reasons.

Anatomy of an HTML element

Let's further explore our paragraph element from the previous section:



The anatomy of our element is:

- **The opening tag:** This consists of the name of the element (in this example, *p* for paragraph), wrapped in opening and closing angle brackets. This opening tag marks where the element begins or starts to take effect. In this example, it precedes the start of the paragraph text.
- **The content:** This is the content of the element. In this example, it is the paragraph text.
- **The closing tag:** This is the same as the opening tag, except that it includes a forward slash before the element name. This marks where the element ends. Failing to include a closing tag is a common beginner error that can produce peculiar results.

The element is the opening tag, followed by content, followed by the closing tag.

Active learning: creating your first HTML element

Edit the line below in the "Editable code" area by wrapping it with the tags <em> and </em>. To *open the element*, put the opening tag <em> at the start of the line. To *close the element*, put the closing tag </em> at the end of the line. Doing this should give the line italic text formatting! See your changes update live in the *Output* area.

If you make a mistake, you can clear your work using the *Reset* button. If you get really stuck, press the *Show solution* button to see the answer.

Nesting elements

Elements can be placed within other elements. This is called *nesting*. If we wanted to state that our cat is **very** grumpy, we could wrap the word *very* in a <strong> element, which means that the word is to have strong(er) text formatting:

<p>My cat is <strong>very</strong> grumpy.</p>

There is a right and wrong way to do nesting. In the example above, we opened the p element first, then opened the strong element. For proper nesting, we should close the strong element first, before closing the p.

The following is an example of the *wrong* way to do nesting:

<p>My cat is <strong>very grumpy.</p></strong>

The **tags have to open and close in a way that they are inside or outside one another**. With the kind of overlap in the example above, the browser has to guess at your intent. This kind of guessing can result in unexpected results.

Block versus inline elements

There are two important categories of elements to know in HTML: block-level elements and inline elements.

- Block-level elements form a visible block on a page. A block-level element appears on a new line following the content that precedes it. Any content that follows a block-level element also appears on a new line. Block-level elements are usually structural elements on the page. For example, a block-level element might represent headings, paragraphs, lists, navigation menus, or footers. A block-level element wouldn't be nested inside an inline element, but it might be nested inside another block-level element.
- Inline elements are contained within block-level elements, and surround only small parts of the document's content (not entire paragraphs or groupings of content). An inline element will not cause a new line to appear in the document. It is typically used with text, for example an <a> element creates a hyperlink, and elements such as <em> or <strong> create emphasis.

Consider the following example:

```
<em>first</em><em>second</em><em>third</em>


<p>fourth</p><p>fifth</p><p>sixth</p>
```

<em> is an inline element. As you see below, the first three elements sit on the same line, with no space in between. On the other hand, <p> is a block-level element. Each *p* element appears on a new line, with space above and below. (The spacing is due to default CSS styling that the browser applies to paragraphs.)

Note: HTML5 redefined the element categories: see Element content categories. While these definitions are more accurate and less ambiguous than their predecessors, the new definitions are a lot more complicated to understand than *block* and *inline.* This article will stay with these two terms.

Note: The terms *block* and *inline*, as used in this article, should not be confused with the types of CSS boxes that have the same names. While the names correlate by default, changing the CSS display type doesn't change the category of the element, and doesn't affect which elements it can contain and which elements it can be contained in. One reason HTML5 dropped these terms was to prevent this rather common confusion.

Note: Find useful reference pages that include lists of block and inline elements. See Block-level elements and Inline elements.

Empty elements

Not all elements follow the pattern of an opening tag, content, and a closing tag. Some elements consist of a single tag, which is typically used to insert/embed something in the document. For example, the <img> element embeds an image file onto a page:

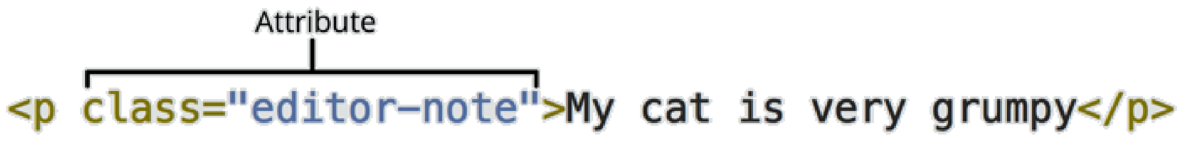
```
<img src="https://raw.githubusercontent.com/mdn/beginner-html-site/gh-pages/images/firefox-icon.png">
```

This would output the following:

Note: Empty elements are sometimes called *void elements*.

Attributes

Elements can also have attributes. Attributes look like this:



```
<p class="editor-note">My cat is very grumpy</p>
```

Attributes contain extra information about the element that won't appear in the content. In this example, the **class** attribute is an identifying name used to target the element with style information.

An attribute should have:

- A space between it and the element name. (For an element with more than one attribute, the attributes should be separated by spaces too.)
- The attribute name, followed by an equal sign.
- An attribute value, wrapped with opening and closing quote marks.

Active learning: Adding attributes to an element

Another example of an element is <a>. This stands for *anchor*. An anchor can make the text it encloses into a hyperlink. Anchors can take a number of attributes, but several are as follows:

- **href**: This attribute's value specifies the web address for the link. For example: href="https://www.mozilla.org/".
- **title**: The title attribute specifies extra information about the link, such as a description of the page that is being linked to. For example, title="The Mozilla homepage". This appears as a tooltip when a cursor hovers over the element.
- **target**: The target attribute specifies the browsing context used to display the link. For example, target="_blank" will display the link in a new tab. If you want to display the linked content in the current tab, just omit this attribute.

Edit the line below in the *Input* area to turn it into a link to your favorite website.

1. Add the <a> element.
2. Add the href attribute and the title attribute.
3. Specify the target attribute to open the link in the new tab.

You'll be able to see your changes update live in the *Output* area. You should see a link—that when hovered over—displays the value of the title attribute, and when clicked, navigates to the web address in the href attribute. Remember that you need to include a space between the element name, and between each attribute.

If you make a mistake, you can always reset it using the *Reset* button. If you get really stuck, press the *Show solution* button to see the answer.

Boolean attributes

Sometimes you will see attributes written without values. This is entirely acceptable. These are called Boolean attributes. Boolean attributes can only have one value, which is generally the same as the attribute name. For example, consider the disabled attribute, which you can assign to form input elements. (You use this to *disable* the form input elements so the user can't make entries. The disabled elements typically have a grayed-out appearance.) For example:

```
<input type="text" disabled="disabled">
```

As shorthand, it is acceptable to write this as follows:

```
<!-- using the disabled attribute prevents the end user from entering text into the input box -->

<input type="text" disabled>



<!-- text input is allowed, as it doesn't contain the disabled attribute -->

<input type="text">
```

For reference, the example above also includes a non-disabled form input element.The HTML from the example above produces this result:

Omitting quotes around attribute values

If you look at code for a lot of other sites, you might come across a number of strange markup styles, including attribute values without quotes. This is permitted in certain circumstances, but it can also break your markup in other circumstances. For example, if we revisit our link example from earlier, we could write a basic version with *only* the href attribute, like this:

```
<a href=https://www.mozilla.org/>favorite website</a>
```

However, as soon as we add the title attribute in this way, there are problems:

```
<a href=https://www.mozilla.org/ title=The Mozilla homepage>favorite website</a>
```

As written above, the browser misinterprets the markup, mistaking the title attribute for three attributes: a title attribute with the value *The*, and two Boolean attributes, Mozilla and homepage. Obviously, this is not intended! It will cause errors or unexpected behavior, as you can see in the live example below. Try hovering over the link to view the title text!

Always include the attribute quotes. It avoids such problems, and results in more readable code.

Single or double quotes?

In this article you will also notice that the attributes are wrapped in double quotes. However, you might see single quotes in some HTML code. This is a matter of style. You can feel free to choose which one you prefer. Both of these lines are equivalent:

```
<a href="http://www.example.com">A link to my example.</a>
```

<a href='http://www.example.com'>A link to my example.</a>

Make sure you don't mix single quotes and double quotes. This example (below) shows a kind of mixing quotes that will go wrong:

<a href="http://www.example.com'>A link to my example.</a>

However, if you use one type of quote, you can include the other type of quote *inside* your attribute values:

<a href="http://www.example.com" title="Isn't this fun?">A link to my example.</a>

To use quote marks inside other quote marks of the same type (single quote or double quote), use HTML entities. For example, this will break:

<a href='http://www.example.com' title='Isn't this fun?'>A link to my example.</a>

Instead, you need to do this:

<a href='http://www.example.com' title='Isn&apos;t this fun?'>A link to my example.</a>

Anatomy of an HTML document

Individual HTML elements aren't very useful on their own. Next, let's examine how individual elements combine to form an entire HTML page:

```
<!DOCTYPE html>

<html>

  <head>

    <meta charset="utf-8">

    <title>My test page</title>

  </head>

  <body>

    <p>This is my page</p>

  </body>

</html>
```

Here we have:

1. <!DOCTYPE html>: The doctype. When HTML was young (1991-1992), doctypes were meant to act as links to a set of rules that the HTML page had to follow to be considered good HTML. Doctypes used to look something like this:
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

More recently, the doctype is a historical artifact that needs to be included for everything else to work right. <!DOCTYPE html> is the shortest string of characters that counts as a valid doctype. That is all you need to know!

3. <html></html>: The <html> element. This element wraps all the content on the page. It is sometimes known as the root element.

4. <head></head>: The <head> element. This element acts as a container for everything you want to include on the HTML page, **that isn't the content** the page will show to viewers. This includes keywords and a page description that would appear in search results, CSS to style content, character set declarations, and more. You'll learn more about this in the next article of the series.

5. <meta charset="utf-8">: This element specifies the character set for your document to UTF-8, which includes most characters from the vast majority of human written languages. With this setting, the page can now handle any textual content it might contain. There is no reason not to set this, and it can help avoid some problems later.

6. <title></title>: The <title> element. This sets the title of the page, which is the title that appears in the browser tab the page is loaded in. The page title is also used to describe the page when it is bookmarked.

7. <body></body>: The <body> element. This contains *all* the content that displays on the page, including text, images, videos, games, playable audio tracks, or whatever else.

Whitespace in HTML

In the examples above, you may have noticed that a lot of whitespace is included in the code. This is optional. These two code snippets are equivalent:

```
<p>Dogs are silly.</p>


<p>Dogs     are

    silly.</p>
```

No matter how much whitespace you use inside HTML element content (which can include one or more space character, but also line breaks), the HTML parser reduces each sequence of whitespace to a single space when rendering the code. So why use so much whitespace? The answer is readability. It can be easier to understand what is going on in your code if you have it nicely formatted. In our HTML we've got each nested element indented by two spaces more than the one it is sitting inside. It is up to you to choose the style of formatting (how many spaces for each level of indentation, for example), but you should consider formatting it.

Entity references: Including special characters in HTML

In HTML, the characters <, >,",' and & are special characters. They are parts of the HTML syntax itself. So how do you include one of these special characters in your text? For example, if you want to use an ampersand or less-than sign, and not have it interpreted as code.

You do this with character references. These are special codes that represent characters, to be used in these exact circumstances. Each character reference starts with an ampersand (&), and ends with a semicolon (;).

Literal character	Character reference equivalent
<	&lt;
>	&gt;
"	&quot;
'	&apos;
&	&amp;

The character reference equivalent could be easily remembered because the text it uses can be seen as less than for '&lt;' , quotation for ' &quot; ' and similarly for others. To find more about entity reference, see List of XML and HTML character entity references (Wikipedia).

In the example below, there are two paragraphs:

<p>In HTML, you define a paragraph using the <p> element.</p>

<p>In HTML, you define a paragraph using the &lt;p&gt; element.</p>

In the live output below, you can see that the first paragraph has gone wrong. The browser interprets the second instance of <p> as starting a new paragraph. The second paragraph looks fine because it has angle brackets with character references.

Note: You don't need to use entity references for any other symbols, as modern browsers will handle the actual symbols just fine as long as your HTML's character encoding is set to UTF-8.

HTML comments

HTML has a mechanism to write comments in the code. Browsers ignore comments, effectively making comments invisible to the user. The purpose of comments is to allow you to include notes in the code to explain your logic or coding. This is very useful if you return to a code base after being away for long enough that you don't completely remember it. Likewise, comments are invaluable as different people are making changes and updates.

To write an HTML comment, wrap it in the special markers <!-- and -->. For example:

<p>I'm not inside a comment</p>

<!-- <p>I am!</p> -->

As you can see below, only the first paragraph displays in the live output.

Required readings – Headings, paragraphs and lists

https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/HTML_text_fundamentals

The basics: headings and paragraphs

Most structured text consists of headings and paragraphs, whether you are reading a story, a newspaper, a college textbook, a magazine, etc.

Structured content makes the reading experience easier and more enjoyable.

In HTML, each paragraph has to be wrapped in a <p> element, like so:

<p>I am a paragraph, oh yes I am.</p>

Each heading has to be wrapped in a heading element:

<h1>I am the title of the story.</h1>

There are six heading elements: <h1>, <h2>, <h3>, <h4>, <h5>, and <h6>. Each element represents a different level of content in the document; <h1> represents the main heading, <h2> represents subheadings, <h3> represents sub-subheadings, and so on.

Implementing structural hierarchy

For example, in this story, the <h1> element represents the title of the story, the <h2> elements represent the title of each chapter, and the <h3> elements represent sub-sections of each chapter:

<h1>The Crushing Bore</h1>

<p>By Chris Mills</p>

<h2>Chapter 1: The dark night</h2>

<p>It was a dark night. Somewhere, an owl hooted. The rain lashed down on the ...</p>

<h2>Chapter 2: The eternal silence</h2>

<p>Our protagonist could not so much as a whisper out of the shadowy figure ...</p>

<h3>The specter speaks</h3>

<p>Several more hours had passed, when all of a sudden the specter sat bolt upright and exclaimed, "Please have mercy on my soul!"</p>

It's really up to you what the elements involved represent, as long as the hierarchy makes sense. You just need to bear in mind a few best practices as you create such structures:

- Preferably, you should use a single <h1> per page—this is the top level heading, and all others sit below this in the hierarchy.
- Make sure you use the headings in the correct order in the hierarchy. Don't use <h3> elements to represent subheadings, followed by <h2> elements to represent sub-subheadings—that doesn't make sense and will lead to weird results.

- Of the six heading levels available, you should aim to use no more than three per page, unless you feel it is necessary. Documents with many levels (i.e., a deep heading hierarchy) become unwieldy and difficult to navigate. On such occasions, it is advisable to spread the content over multiple pages if possible.

Why do we need structure?

- Users looking at a web page tend to scan quickly to find relevant content, often just reading the headings, to begin with. (We usually spend a very short time on a web page.) If they can't see anything useful within a few seconds, they'll likely get frustrated and go somewhere else.
- **Search engines indexing your page consider the contents of headings as important keywords for influencing the page's search rankings. Without headings, your page will perform poorly in terms of SEO (Search Engine Optimization).**
- Severely visually impaired people often don't read web pages; they listen to them instead. This is done with software called a screen reader. This software provides ways to get fast access to given text content. Among the various techniques used, they provide an outline of the document by reading out the headings, allowing their users to find the information they need quickly. If headings are not available, they will be forced to listen to the whole document read out loud.
- To style content with CSS, or make it do interesting things with JavaScript, you need to have elements wrapping the relevant content, so CSS/JavaScript can effectively target it.

Therefore, we need to give our content structural markup.

Active learning: Giving our content structure

Let's jump straight in with a live example. In the example below, add elements to the raw text in the *Input* field so that it appears as a heading and two paragraphs in the *Output* field.

If you make a mistake, you can always reset it using the *Reset* button. If you get stuck, press the *Show solution* button to see the answer.

Why do we need semantics?

Semantics are relied on everywhere around us—we rely on previous experience to tell us what the function of an everyday object is; when we see something, we know what its function will be. So, for example, we expect a red traffic light to mean "stop," and a green traffic light to mean "go." Things can get tricky very quickly if the wrong semantics are applied. (Do any countries use red to mean "go"? We hope not.)

In a similar vein, we need to make sure we are using the correct elements, giving our content the correct meaning, function, or appearance. In this context, the <h1> element is also a semantic element, which gives the text it wraps around the role (or meaning) of "a top level heading on your page."

<h1>This is a top level heading</h1>

By default, the browser will give it a large font size to make it look like a heading (although you could style it to look like anything you wanted using CSS). More importantly, its semantic value will be used in multiple ways, for example by search engines and screen readers (as mentioned above).

On the other hand, you could make any element *look* like a top level heading. Consider the following:

<span style="font-size: 32px; margin: 21px 0; display: block;">Is this a top level heading?</span>

This is a <span> element. It has no semantics. You use it to wrap content when you want to apply CSS to it (or do something to it with JavaScript) without giving it any extra meaning. (You'll find out more about these later on in the course.) We've applied some CSS to it to make it look like a top level heading, but since it has no semantic value, it will not get any of the extra benefits described above. It is a good idea to use the relevant HTML element for the job.

Lists

Now let's turn our attention to lists. Lists are everywhere in life—from your shopping list to the list of directions you subconsciously follow to get to your house every day, to the lists of instructions you are following in these tutorials! Lists are everywhere on the web, too, and we've got three different types to worry about.

Unordered

Unordered lists are used to mark up lists of items for which the order of the items doesn't matter. Let's take a shopping list as an example:

milk

eggs

bread

hummus

Every unordered list starts off with a <ul> element—this wraps around all the list items:

```
<ul>

milk

eggs

bread

hummus

</ul>
```

The last step is to wrap each list item in a <li> (list item) element:

```
<ul>

  <li>milk</li>

  <li>eggs</li>

  <li>bread</li>

  <li>hummus</li>
```

```
</ul>
```

Active learning: Marking up an unordered list

Try editing the live sample below to create your very own HTML unordered list.

Ordered

Ordered lists are lists in which the order of the items *does* matter. Let's take a set of directions as an example:

Drive to the end of the road

Turn right

Go straight across the first two roundabouts

Turn left at the third roundabout

The school is on your right, 300 meters up the road

The markup structure is the same as for unordered lists, except that you have to wrap the list items in an <ol> element, rather than <ul>:

```
<ol>
  <li>Drive to the end of the road</li>
  <li>Turn right</li>
  <li>Go straight across the first two roundabouts</li>
  <li>Turn left at the third roundabout</li>
  <li>The school is on your right, 300 meters up the road</li>
</ol>
```

Active learning: Marking up an ordered list

Try editing the live sample below to create your very own HTML ordered list.

Active learning: Marking up our recipe page

So at this point in the article, you have all the information you need to mark up our recipe page example. You can choose to either save a local copy of our text-start.html starting file and do the work there or do it in the editable example below. Doing it locally will probably be better, as then you'll get to save the work you are doing, whereas if you fill it in to the editable example, it will be lost the next time you open the page. Both have pros and cons.

If you get stuck, you can always press the *Show solution* button, or check out our text-complete.html example on our github repo.

Nesting lists

It is perfectly ok to nest one list inside another one. You might want to have some sub-bullets sitting below a top-level bullet. Let's take the second list from our recipe example:

```
<ol>
  <li>Remove the skin from the garlic, and chop coarsely.</li>
  <li>Remove all the seeds and stalk from the pepper, and chop coarsely.</li>
  <li>Add all the ingredients into a food processor.</li>
  <li>Process all the ingredients into a paste.</li>
  <li>If you want a coarse "chunky" hummus, process it for a short time.</li>
  <li>If you want a smooth hummus, process it for a longer time.</li>
</ol>
```

Since the last two bullets are very closely related to the one before them (they read like sub-instructions or choices that fit below that bullet), it might make sense to nest them inside their own unordered list and put that list inside the current fourth bullet. This would look like so:

```
<ol>
  <li>Remove the skin from the garlic, and chop coarsely.</li>
  <li>Remove all the seeds and stalk from the pepper, and chop coarsely.</li>
  <li>Add all the ingredients into a food processor.</li>
  <li>Process all the ingredients into a paste.
    <ul>
      <li>If you want a coarse "chunky" hummus, process it for a short time.</li>
      <li>If you want a smooth hummus, process it for a longer time.</li>
    </ul>
  </li>
</ol>
```

Try going back to the previous active learning example and updating the second list like this.

Emphasis and importance

In human language, we often emphasize certain words to alter the meaning of a sentence, and we often want to mark certain words as important or different in some way. HTML provides various semantic elements to allow us to mark up textual content with such effects, and in this section, we'll look at a few of the most common ones.

Emphasis

When we want to add emphasis in spoken language, we *stress* certain words, subtly altering the meaning of what we are saying. Similarly, in written language we tend to stress words by putting them in italics. For example, the following two sentences have different meanings.

I am glad you weren't late.

I am *glad* you weren't *late*.

The first sentence sounds genuinely relieved that the person wasn't late. In contrast, the second one sounds sarcastic or passive-aggressive, expressing annoyance that the person arrived a bit late.

In HTML we use the <em> (emphasis) element to mark up such instances. As well as making the document more interesting to read, these are recognized by screen readers and spoken out in a different tone of voice. Browsers style this as italic by default, but you shouldn't use this tag purely to get italic styling. To do that, you'd use a <span> element and some CSS, or perhaps an <i> element (see below).

<p>I am <em>glad</em> you weren't <em>late</em>.</p>

Strong importance

To emphasize important words, we tend to stress them in spoken language and **bold** them in written language. For example:

This liquid is **highly toxic**.

I am counting on you. **Do not** be late!

In HTML we use the <strong> (strong importance) element to mark up such instances. As well as making the document more useful, again these are recognized by screen readers and spoken in a different tone of voice. Browsers style this as bold text by default, but you shouldn't use this tag purely to get bold styling. To do that, you'd use a <span> element and some CSS, or perhaps a <b> element (see below).

<p>This liquid is <strong>highly toxic</strong>.</p>

<p>I am counting on you. <strong>Do not</strong> be late!</p>

You can nest strong and emphasis inside one another if desired:

<p>This liquid is <strong>highly toxic</strong> —

if you drink it, <strong>you may <em>die</em></strong>.</p>

Active learning: Let's be important!

In this active learning section, we've provided an editable example. Inside it, we'd like you to try adding emphasis and strong importance to the words you think need them, just to have some practice.

Italic, bold, underline...

The elements we've discussed so far have clearcut associated semantics. The situation with <b>, <i>, and <u> is somewhat more complicated. They came about so people could write bold, italics, or underlined text in an era when CSS was still supported poorly or not at all. Elements like this, which only affect presentation and not semantics, are known as **presentational elements** and should no longer be used because, as we've seen before, semantics is so important to accessibility, SEO, etc.

HTML5 redefined <b>, <i>, and <u> with new, somewhat confusing, semantic roles.

Here's the best rule of thumb: It's likely appropriate to use <b>, <i>, or <u> to convey a meaning traditionally conveyed with bold, italics, or underline, provided there is no more suitable element. However, it always remains critical to keep an accessibility mindset. The concept of italics isn't very helpful to people using screen readers, or to people using a writing system other than the Latin alphabet.

- <i> is used to convey a meaning traditionally conveyed by italic: foreign words, taxonomic designation, technical terms, a thought...
- <b> is used to convey a meaning traditionally conveyed by bold: key words, product names, lead sentence...
- <u> is used to convey a meaning traditionally conveyed by underline: proper name, misspelling...

A kind warning about underline: **People strongly associate underlining with hyperlinks.** Therefore, on the web, it's best to underline only links. Use the <u> element when it's semantically appropriate, but consider using CSS to change the default underline to something more appropriate on the web. The example below illustrates how it can be done.

Required readings – Sections of a Document

https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure

Basic sections of a document

Webpages can and will look pretty different from one another, but they all tend to share similar standard components, unless the page is displaying a fullscreen video or game, is part of some kind of art project, or is just badly structured:

header:
Usually a big strip across the top with a big heading, logo, and perhaps a tagline. This usually stays the same from one webpage to another.

navigation bar:
Links to the site's main sections; usually represented by menu buttons, links, or tabs. Like the header, this content usually remains consistent from one webpage to another — having inconsistent navigation on your website will just lead to confused, frustrated users. Many web designers consider the navigation bar to be part of the header rather than an individual component, but that's not a requirement; in fact, some also argue that having the two separate is better for accessibility, as screen readers can read the two features better if they are separate.

main content:
A big area in the center that contains most of the unique content of a given webpage, for example, the video you want to watch, or the main story you're reading, or the map you want to view, or the news headlines, etc. This is the one part of the website that definitely will vary from page to page!

sidebar:
Some peripheral info, links, quotes, ads, etc. Usually, this is contextual to what is contained in the main content (for example on a news article page, the sidebar might contain the author's bio, or links to related articles) but there are also cases where you'll find some recurring elements like a secondary navigation system.

footer:
A strip across the bottom of the page that generally contains fine print, copyright notices, or contact info. It's a place to put common information (like the header) but usually, that information is not critical or secondary to the website itself. The footer is also sometimes used for SEO purposes, by providing links for quick access to popular content.

HTML for structuring content

In your HTML code, you can mark up sections of content based on their *functionality* — you can use elements that represent the sections of content described above unambiguously, and assistive technologies like screenreaders can recognise those elements and help with tasks like "find the main navigation", or "find the main content." As we mentioned earlier in the course, there are a number of consequences of not using the right element structure and semantics for the right job.

To implement such semantic mark up, HTML provides dedicated tags that you can use to represent such sections, for example:

- **header:** <header>.
- **navigation bar:** <nav>.
- **main content:** <main>, with various content subsections represented by <article>, <section>, and <div> elements.
- **sidebar:** <aside>; often placed inside <main>.
- **footer:** <footer>.

HTML layout elements in more detail

It's good to understand the overall meaning of all the HTML sectioning elements in detail — this is something you'll work on gradually as you start to get more experience with web development. You can find a lot of detail by reading our HTML element reference. For now, these are the main definitions that you should try to understand:

- <main> is for content *unique to this page.* Use <main> only *once* per page, and put it directly inside <body>. Ideally this shouldn't be nested within other elements.
- <article> encloses a block of related content that makes sense on its own without the rest of the page (e.g., a single blog post).
- <section> is similar to <article>, but it is more for grouping together a single part of the page that constitutes one single piece of functionality (e.g., a mini map, or a set of article headlines and summaries), or a theme. It's considered best practice to begin each section with a heading; also

note that you can break <article>s up into different <section>s, or <section>s up into different <article>s, depending on the context.

- <aside> contains content that is not directly related to the main content but can provide additional information indirectly related to it (glossary entries, author biography, related links, etc.).
- <header> represents a group of introductory content. If it is a child of <body> it defines the global header of a webpage, but if it's a child of an <article> or <section> it defines a specific header for that section (try not to confuse this with titles and headings).
- <nav> contains the main navigation functionality for the page. Secondary links, etc., would not go in the navigation.
- <footer> represents a group of end content for a page.

Each of the aforementioned elements can be clicked on to read the corresponding article in the "HTML element reference" section, providing more detail about each one.

Non-semantic wrappers

Sometimes you'll come across a situation where you can't find an ideal semantic element to group some items together or wrap some content. Sometimes you might want to just group a set of elements together to affect them all as a single entity with some CSS or JavaScript. For cases like these, HTML provides the <div> and <span> elements. You should use these preferably with a suitable class attribute, to provide some kind of label for them so they can be easily targeted.

<span> is an inline non-semantic element, which you should only use if you can't think of a better semantic text element to wrap your content, or don't want to add any specific meaning. For example:

```
<p>The King walked drunkenly back to his room at 01:00, the beer doing nothing to aid

him as he staggered through the door <span class="editor-note">[Editor's note: At this point in the

play, the lights should be down low]</span>.</p>
```

In this case, the editor's note is supposed to merely provide extra direction for the director of the play; it is not supposed to have extra semantic meaning. For sighted users, CSS would perhaps be used to distance the note slightly from the main text.

<div> is a block level non-semantic element, which you should only use if you can't think of a better semantic block element to use, or don't want to add any specific meaning. For example, imagine a shopping cart widget that you could choose to pull up at any point during your time on an e-commerce site:

```
<div class="shopping-cart">

  <h2>Shopping cart</h2>

  <ul>

    <li>
    <p><a href=""><strong>Silver earrings</strong></a>: $99.95.</p>

    <img src="../products/3333-0985/thumb.png" alt="Silver earrings">

  </li>

  <li>

    ...

  </li>

 </ul>

 <p>Total cost: $237.89</p>

</div>
```

This isn't really an <aside>, as it doesn't necessarily relate to the main content of the page (you want it viewable from anywhere). It doesn't even particularly warrant using a <section>, as it isn't part of the main content of the page. So a <div> is fine in this case. We've included a heading as a signpost to aid screenreader users in finding it.

Warning: Divs are so convenient to use that it's easy to use them too much. As they carry no semantic value, they just clutter your HTML code. Take care to use them only when there is no better semantic solution and try to reduce their usage to the minimum otherwise you'll have a hard time updating and maintaining your documents.

Line breaks and horizontal rules

Two elements that you'll use occasionally and will want to know about are <br> and <hr>:

<br> creates a line break in a paragraph <hr> elements create a horizontal rule in the document that denotes a thematic change in the text (such as a change in topic or scene).

Required readings – Web forms

https://developer.mozilla.org/en-US/docs/Learn/Forms/Your_first_form

Web forms are one of the main points of interaction between a user and a web site or application. Forms allow users to enter data, which is generally sent to a web server for processing and storage (see Sending form data later in the module), or used on the client-side to immediately update the interface in some way (for example, add another item to a list, or show or hide a UI feature).

A web form's HTML is made up of one or more **form controls** (sometimes called **widgets**), plus some additional elements to help structure the overall form — they are often referred to as **HTML forms**. The controls can be single or multi-line text fields, dropdown boxes, buttons, checkboxes, or radio buttons, and are mostly created using the <input> element, although there are some other elements to learn about too.

Form controls can also be programmed to enforce specific formats or values to be entered (**form validation**), and paired with text labels that describe their purpose to both sighted and blind users.

Designing your form

From a user experience (UX) point of view, it's important to remember that the bigger your form, the more you risk frustrating people and losing users. Keep it simple and stay focused: ask only for the data you absolutely need.

Active learning: Implementing our form HTML

The <form> element

All forms start with a <form> element, like this:

```
<form action="/my-handling-form-page" method="post">


</form>
```

This element formally defines a form. It's a container element like a <section> or <footer> element, but specifically for containing forms; it also supports some specific attributes to configure the way the form behaves. All of its attributes are optional, but it's standard practice to always set at least the action and method attributes:

- The action attribute defines the location (URL) where the form's collected data should be sent when it is submitted.
- The method attribute defines which HTTP method to send the data with (usually get or post).

The <label>, <input>, and <textarea> elements

Our contact form is not complex: the data entry portion contains three text fields, each with a corresponding <label>:

The input field for the name is a single-line text field.

The input field for the e-mail is an input of type email: a single-line text field that accepts only e-mail addresses.

The input field for the message is a <textarea>; a multiline text field.

In terms of HTML code we need something like the following to implement these form widgets:

```
<form action="/my-handling-form-page" method="post">
 <ul>
  <li>
   <label for="name">Name:</label>
   <input type="text" id="name" name="user_name">
  </li>
  <li>
   <label for="mail">E-mail:</label>
   <input type="email" id="mail" name="user_email">
  </li>
  <li>
   <label for="msg">Message:</label>
   <textarea id="msg" name="user_message"></textarea>
  </li>
 </ul>
</form>
```

Update your form code to look like the above.

The <li> elements are there to conveniently structure our code and make styling easier (see later in the article). For usability and accessibility, we include an explicit label for each form control. Note the use of the for attribute on all <label> elements, which takes as its value the id of the form control with which it is associated — this is how you associate a form control with its label.

There is great benefit to doing this — it associates the label with the form control, enabling mouse, trackpad, and touch device users to click on the label to activate the corresponding control, and it also provides an accessible name for screen readers to read out to their users. You'll find further details of form labels in How to structure a web form.

On the <input> element, the most important attribute is the type attribute. This attribute is extremely important because it defines the way the <input> element appears and behaves. You'll find more about this in the Basic native form controls article later on.

- In our simple example, we use the value <input/text> for the first input — the default value for this attribute. It represents a basic single-line text field that accepts any kind of text input.
- For the second input, we use the value <input/email>, which defines a single-line text field that only accepts a well-formed e-mail address. This turns a basic text field into a kind of "intelligent" field that will perform some validation checks on the data typed by the user. It also causes a more appropriate keyboard layout for entering email addresses (e.g. with an @ symbol by default) to appear on devices with dynamic keyboards, like smartphones. You'll find out more about form validation in the client-side form validation article later on.

Last but not least, note the syntax of <input> vs. <textarea></textarea>. This is one of the oddities of HTML. The <input> tag is an empty element, meaning that it doesn't need a closing tag. <textarea> is not an empty element, meaning it should be closed with the proper ending tag. This has an impact on a specific feature of forms: the way you define the default value. To define the default value of an <input> element you have to use the value attribute like this:

```
<input type="text" value="by default this element is filled with this text">
```

On the other hand, if you want to define a default value for a <textarea>, you put it between the opening and closing tags of the <textarea> element, like this:

```
<textarea>

by default this element is filled with this text

</textarea>
```

The <button> element

The markup for our form is almost complete; we just need to add a button to allow the user to send, or "submit", their data once they have filled out the form. This is done by using the <button> element; add the following just above the closing </ul> tag:

```
<li class="button">

  <button type="submit">Send your message</button>

</li>
```

The <button> element also accepts a type attribute — this accepts one of three values: submit, reset, or button.

- A click on a submit button (the default value) sends the form's data to the web page defined by the action attribute of the <form> element.
- A click on a reset button resets all the form widgets to their default value immediately. From a UX point of view, this is considered bad practice, so you should avoid using this type of button unless you really have a good reason to include one.
- A click on a button button does... nothing! That sounds silly, but it's amazingly useful for building custom buttons — you can define their chosen functionality with JavaScript.

Note: You can also use the <input> element with the corresponding type to produce a button, for example <input type="submit">. The main advantage of the <button> element is that the <input> element only allows plain text in its label whereas the <button> element allows full HTML content, allowing more complex, creative button content.

Sending form data to your web server

The last part, and perhaps the trickiest, is to handle form data on the server side. The <form> element defines where and how to send the data thanks to the action and method attributes.

We provide a name to each form control. The names are important on both the client- and server-side; they tell the browser which name to give each piece of data and, on the server side, they let the server handle each piece of data by name. The form data is sent to the server as name/value pairs.

To name the data in a form you need to use the name attribute on each form widget that will collect a specific piece of data. Let's look at some of our form code again: In our example, the form will send 3 pieces of data named "user_name", "user_email", and "user_message". That data will be sent to the URL "/my-handling-form-page" using the HTTP POST method.

```
<form action="/my-handling-form-page" method="post">

 <ul>

  <li>

    <label for="name">Name:</label>

    <input type="text" id="name" name="user_name" />

  </li>

  <li>

    <label for="mail">E-mail:</label>

    <input type="email" id="mail" name="user_email" />

  </li>

  <li>

    <label for="msg">Message:</label>

    <textarea id="msg" name="user_message"></textarea>

  </li>



  …
```

On the server side, the script at the URL "/my-handling-form-page" will receive the data as a list of 3 key/value items contained in the HTTP request. The way this script will handle that data is up to you. Each server-side language (PHP, Python, Ruby, Java, C#, etc.) has its own mechanism of handling form data. It's beyond the scope of this guide to go deeply into that subject, but if you want to know more, we have provided some examples in our Sending form data article later on.

Lecture

Web sites vs. web applications vs. the web as a platform

The Web Platform Working Groupis responsible for a number of web technologies that move us closer towards the vision of the browser as the operating system, including:

- client-side database and offline applications;
- file and filesystem APIs;
- WebSockets;
- Web Workers (enables web applications to spawn background processes);
- DOM & HTML;
- Canvas (for drawing);
- Web components (a component model for the web).

Cross-platform applications

Do web technologies also help you to create desktop apps or mobile apps?

Indeed, they do! Electron is an open-source project that enables you to build **cross-platform desktop apps** for Windows, Mac and Linux with HTML, JavaScript and CSS - the very technologies you learn about in this course.

Electron itself uses Node.js, the server-side JavaScript runtime we cover in a later lecture together with Chromium, an open-source web runtime.

The major benefit of Electron should be clear: instead of writing three separate desktop variants (one for each operating system) you only have to write and maintain one. New features are integrated in one application instead of three, which reduces feature delivery time. For these reasons, many well-known applications today are built on Electron, including Visual Studio Code (the IDE we recommend you to use), the Slack app, Atom and many, many more.

While Electron focuses on **cross-platform desktop applications**, initiatives with a similar goal for the mobile world (i.e. generating native apps for Android/iOS from HTML/CSS/JavaScript) exist as well. By far the most popular is React Native. Before you can use it though you will have to learn React, a very popular JavaScript library for building user interfaces that is being developed by Facebook.

Web design basics

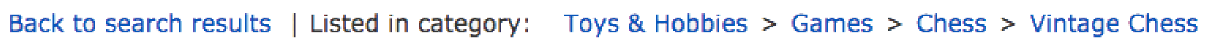
Rule: Don't make me think

The way a web site or web application (I tend to use the terms interchangeably here; the rules apply to both apps and sites) works should be self-evident; the user should not have to expend cognitive effort to understand what she can do.

When naming and formatting links, buttons, section headers, etc. adhere to **established standards** and **be clear instead of clever**. For instance, a company's web site that has a link to its current job offers should use as link text Jobs or Vacancies (clear to the user what this link is about) instead of Interested? or Join us! (less clear).

Similarly, there are established style standards of how to format a link: in the early years of the web, blue underlined text was synonymous with a link and thus we are now stuck with the saying *10 blue links* as a synonym for web search results.

Users should also **not get lost within a web site**. A site should provide users with information on where they are and on how they arrived at that point. Ebay for instance leaves so-called **breadcrumbs**


Back to search results | Listed in category: Toys & Hobbies > Games > Chess > Vintage Chess

Lastly, it should be easy for the user to **distinguish different parts of a site** such as advertisement vs. content. Disguising ads as content is known as a **dark pattern**.

Rule: Minimize noise and clutter

Rule: If you cannot make it self-evident, make it self-explanatory

When a site is not self-explanatory, a small amount of explanatory text can go a long way. In today's mobile world where a lot of content is accessed *mobile first*, it is vital to keep the mobile user in mind who has to deal with a small screen, a touch-based interface and possibly many distractions while surfing the web.

The entry page (or home page) of a web application should answer a number of essential questions, none of which are answered in our example:

- What **is** this?
- What can I **do** here?
- Why **should** I be here?
- What do they **have** here?

Lastly, **avoid happy talk**, that is text without any content for the sake of adding some text (e.g. a welcome message).

Aesthetics matter (of course!)

Usability testing: expectations vs. reality

Our expectations of web users are often *not* grounded in reality. We may expect users to be **rational**, **attentive** and having a **clear goal** in mind.

Instead, the average web user:

- quickly scans a web page (not even reading it);
- decides within seconds whether or not a site is worth their attention (research on so-called *dwell time* has shown this again and again);
- clicks on the first link they find;
- depends a lot on the browser's back button (which can causes issues for web applications);
- does not read instructions.

A web application should be designed based on **user reality**. **Usability testing** is an important step to create a well-designed web application. The **development cycle** consists of *designing-testing-reviewing*:

$$\begin{tikzcd} & \text{design} \arrow[dr] & \\ \text{review} \arrow[ur] & & \text{test} \arrow[ll, bend left] \end{tikzcd}$$

In a **usability test**, a user is given a **typical task**, such as:

- Create a user account.
- Retrieve a lost password.
- Change the current credit card information.
- Delete a user account.
- Find an article in the archive.
- Edit a posting made in a forum.
- Start a game with three players.

While the user is busy with the task, her actions towards completing the task are being recorded. These actions are then translated into performance metrics. Performance metrics depend on the task, it could be the **number of clicks** required to complete the task, the **time taken** or the **number of wrongly clicked elements**.

Usability testers should be a mix of target audience and average web users; 2-3 testers per iteration tend to be sufficient.

A typical usability setup has the following roles:

- The **participant** (our tester) sits in front of the device (laptop, mobile phone, tablet).
- The **facilitator** sits next to her and guides her through the test.
- The **observers** (developers of the app, managers, etc.) watch the usability test and discuss the tester's performance (and how to improve it) afterwards.

The result of a usability test are a set of issues. Each of those issues should be assigned a **priority** (low, medium, high) and the next iteration of the development should focus on the high priority problems. No new issues should be added to the list until the most severe issues are fixed.

Performance metrics

Once web applications become complex, besides general usability we also need to consider various **performance metrics**, ideally metrics that are *user-centric*. A basic question to ask is: *how long does it take for the entire application to load?*. This is of course a rather crude metric and it is not always clear what exactly this means, due to polling, etc. We thus need more fine-grained performance metrics. The web vitals section at Google's web.dev resource provides a good overview of what questions to ask in order to measure aspects of an application's performance that are *relevant* to the user. Let's focus on the three metrics that Google considers to be core web vitals:

- Largest contentful paint (LCP): LCP measures the time it takes to load a page's main content, which correlates very well with users' perceived load speed.
- First input delay (FID): FID measures the time between a user first interacting with a page and the browser processing corresponding event handlers.

- Cumulative layout shift (CLS): CLS measures a page's visual stability by aggregating to what extent visible elements shift in the layout unexpectedly.

As an example of a popular online web metrics tool let's take a look at Google's https://web.dev/measure/. An alternative service is webhint.io

Beyond online tools, browser extensions such as Lighthouse and web performance libraries such as Perfume.js exist as well that allow you to gain more specific insights into your application's performance.

HTML5

HTML5 is a set of related technologies (core HTML5, CSS, JavaScript) that together enable **rich web content**:

- **Core HTML5**: mark up content;
- **CSS**: control the appearance of marked-up content;
- client-side **JavaScript**: manipulate the contents of HTML documents and respond to user interactions.

Modern web application development requires knowledge of all three technologies. In practice, it also requires a whole set of additional frameworks and tools to go from prototype code to production code, such as build tools, transpilers, code coverage tools and so on. This is the case even "just" for frontend coding.

Before HTML5 we had **XHTML** and HTML 4.01. XHTML is a reformulation of HTML 4 as an XML 1.0 application and stands for **Extensible HyperText Markup Language**. It looks as follows (taken straight from the W3C XHTML recommendation):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
   <title>Virtual Library</title>
  </head>
  <body>
   <p>Moved to <a href="http://example.org/">example.org</a>.</p>
  </body>
</html>
```

XHTML was designed to make processing of web pages **easier for machines** by having a very strict set of rules. The problem though was that (X)HTML is written by developers, not machines and it turned out to be too much hassle to write valid XHTML. Moreover, browsers were and are able to render invalid XHTML pages properly (so why even try to write valid XHTML?) and thus XHTML was eventually abandoned in favor of HTML5, which is not only less strict but also added a host of new features to the language.

With this introduction of new features **browser compatibility** issues returned: some browser vendors are faster than others in implementing W3C standards (in addition to implementing their own non-standardized features). A good resource to check which browser versions support which HTML5 feature and to what extent is https://caniuse.com/.

A polyfill is a code snippet that brings a modern browser feature (such as a calendar picker) to browsers that do not natively support that feature. Here is an example of a polyfill for Safari (among others). Sometimes when looking for polyfills one can also come across the term *shim*. These concepts are not quite interchangeable, though they often used in the same contexts.

The move towards HTML5

The initial list of HTML tags (1991/92) was **static**: <title> <a> <isindex> <plaintext> <listing> <p> <h1> <address> <hp1> <dl> <dt> <ul>. JavaScript was created within 10 days (*which explains many JavaScript quirks*) in May 1995. It was the beginning of client-side **dynamic** scripting for the browser.

Plugins were created to go beyond what at the time was possible with HTML. Probably the most famous plugin remains Adobe Flash, which was introduced in 1996. HTML5 is a drive to return rich content **directly** into the browser, without the need for plugins or addons.

HTML5 introduced a number of **semantic HTML elements** including <article> <footer> <header> <main> <aside> <section> <output>. As a guideline, when creating an HTML document, it is always best to select the **most specific** element to represent your content (instead of only using <div>'s). Semantic elements provide **meaning** but do not force a particular presentation. Older HTML elements (pre-HTML5) often do force a particular presentation, e.g. <b> (bold) or <i> (italics). At the same time, those heavily used HTML elements cannot be moved to an obsolete state - as this would inevitably break a large portion of the web. For the browser vendors, backwards compatibility is a necessity, not an option. It should be pointed out that **semantic HTML** is quite different from the grand vision of the Semantic Web:

The Semantic Web is a Web of data — of dates and titles and part numbers and chemical properties and any other data one might conceive of.

Who decides the HTML standard

HTML is widely used, which makes standardization a slow process. Many different stakeholders are part of W3C's Web Platform Working Group (Microsoft, Google, Mozilla, Nokia, Baidu, Yandex, etc.). The standardization process of the W3C is elaborate, as a wide variety of stakeholders have to build consensus. Confusingly, a **W3C recommendation** is the highest level of standardization possible, before achieving it, a number of steps leading up to the recommendation are required:

1. **Working Draft**: *a document that W3C has published for review by the community, including W3C Members, the public, and other technical organizations.*

2. **Candidate Recommendation**: *a document that W3C believes has been widely reviewed and satisfies the Working Group's technical requirements. W3C publishes a Candidate Recommendation to gather implementation experience.*

3. **Proposed Recommendation**: *a mature technical report that, after wide review for technical soundness and implementability, W3C has sent to the W3C Advisory Committee for final endorsement.*

4. **W3C Recommendation**: *a specification or set of guidelines that, after extensive consensus-building, has received the endorsement of W3C Members and the Director. W3C recommends the wide deployment of its Recommendations. Note: W3C Recommendations are similar to the standards published by other organizations.*

Week 7. JavaScript lecture, Node.js lecture; second web technology assignment;

Lecture 3. JavaScript: the language of browser interactions

Required readings - A first splash into JavaScript

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash

The place where we'll be adding all our code is inside the <script> element at the bottom of the HTML:

```
<script>

  // Your JavaScript goes here

</script>
```

Adding variables to store our data

- use 'let' for variables
- 'const' for constants i.e. to store references to parts of html

Javascript operators

- == is ===
- != is !==
- Math and concatenation are same as Java
- conditionals are the same as Java

Events

Events are things that happen in the browser — a button being clicked, a page loading, a video playing, etc. — in response to which we can run blocks of code. The constructs that listen out for the event happening are called **event listeners**, and the blocks of code that run in response to the event firing are called **event handlers**.

```
<input type="submit" value="Submit guess" class="guessSubmit">

...

<script>

...

guessSubmit.addEventListener('click', checkGuess);
```

Loops

```
for (let i = 1 ; i < 21 ; i++) { console.log(i) }
```

Some methods

```
guessField.value = 'Hello'; //this is for buttons

guesses.textContent = 'Where is my paragraph?'; //this is for html paragraphs

guesses.style.backgroundColor = 'yellow';
guesses.style.fontSize = '200%';

guesses.style.padding = '10px';

guesses.style.boxShadow = '3px 3px 6px black';
```

Required readings – Variables

Use let instead of var as let is an improved version that prevents you from writing bad code.

Skipped types are the same as in Java

Arrays

In JS you don't need to initialize the length of the array. It grows dynamically. You can initialise an array with just

```
let myArray = [];
```

and you retrieve values like in java myArray[index];

Objects

JS objects are more like variables with subvariables (and subfunctions). All the attributes can be defined in one line.

```
let dog = { name : 'Spot', breed : 'Dalmatian' };
```

There is no need for setters and getters, to retrieve information just use

```
dog.name
```

Dynamic typing

JS does not require you to specify the type, just initalise the variable with a value and JS will do the rest.

Constants

Use constants ‘const’ to assign values that won’t change, such as a reference to a part in html.

Required readings – Srrings

Single quotes vs. double quotes

In JavaScript, you can choose single quotes or double quotes to wrap your strings in.

When you start with “ you dont need to escape ‘ and viceversa. To escpace them use \

```
let sglDbl = 'Would you eat a "fish supper"?';

let dblSgl = "I'm feeling blue.";

let bigmouth = 'I\'ve got no right to take my place...';
```

casting

```
let myNum = Number(myString); \\ will parse it to a number (or NaN)

let myString = myNum.toString(); \\ same as in Java
```

Template literals

They are like the assembly strings, where a variable is explicitly inside the string, denoted by ${ variable } and the string quotes are ` istead of ‘

```
let examScore = 45;

let examHighestScore = 70;

examReport = `You scored ${ examScore }/${ examHighestScore } (${
Math.round((examScore/examHighestScore*100)) }%). ${ examScore >= 49 ? 'Well
done, you passed!' : 'Bad luck, you didn\'t pass this time.' }`;
```

Ternary operator works like in Java. With literals you don’t need to use \n you can literally have a breakline in your code instead.

Template literals are good practice, prefered over standard string literals. Internet Explorer does not support them well though.

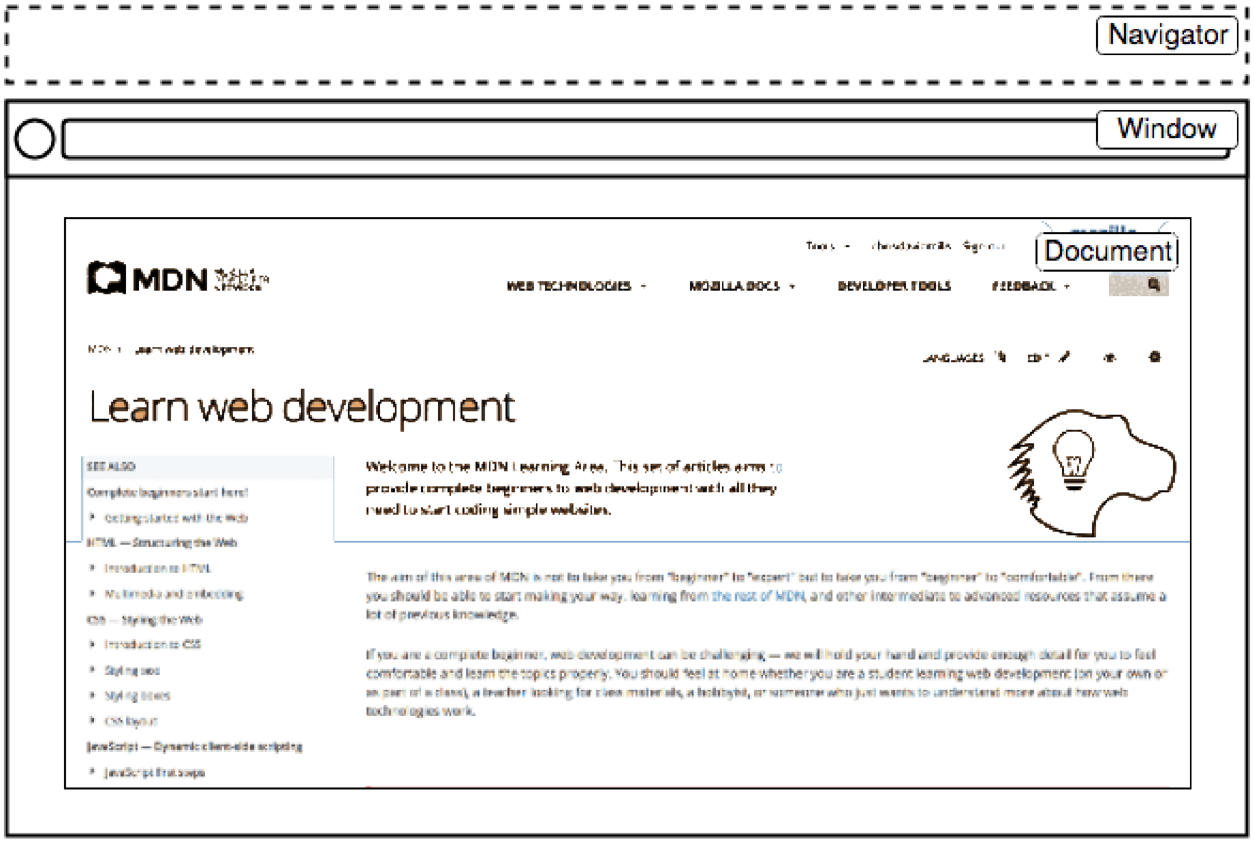
Required readings - Manipulating documents

When writing web pages and apps, one of the most common things you'll want to do is manipulate the document structure in some way. This is usually done by using the Document Object Model (DOM), a set of APIs for controlling HTML and styling information that makes heavy use of the Document object. In this article we'll look at how to use the DOM in detail, along with some other interesting APIs that can alter your environment in interesting ways.

The important parts of a web browser

Web browsers are very complicated pieces of software with a lot of moving parts, many of which can't be controlled or manipulated by a web developer using JavaScript. You might think that such limitations are a bad thing, but browsers are locked down for good reasons, mostly centering around security. Imagine if a web site could get access to your stored passwords or other sensitive information, and log into websites as if it were you?

Despite the limitations, Web APIs still give us access to a lot of functionality that enable us to do a great many things with web pages. There are a few really obvious bits you'll reference regularly in your code — consider the following diagram, which represents the main parts of a browser directly involved in viewing web pages:



- The window is the browser tab that a web page is loaded into; this is represented in JavaScript by the Window object. Using methods available on this object you can do things like return the window's size (see Window.innerWidth and Window.innerHeight), manipulate the document loaded into that window, store data specific to that document on the client-side (for example using a local database or other storage mechanism), attach an event handler to the current window, and more.
- The navigator represents the state and identity of the browser (i.e. the user-agent) as it exists on the web. In JavaScript, this is represented by the Navigator object. You can use this object to retrieve things like the user's preferred language, a media stream from the user's webcam, etc.
- The document (represented by the DOM in browsers) is the actual page loaded into the window, and is represented in JavaScript by the Document object. You can use this object to return and manipulate information on the HTML and CSS that comprises the document, for example get a reference to an element in the DOM, change its text content, apply new styles to it, create new elements and add them to the current element as children, or even delete it altogether.

The document object model

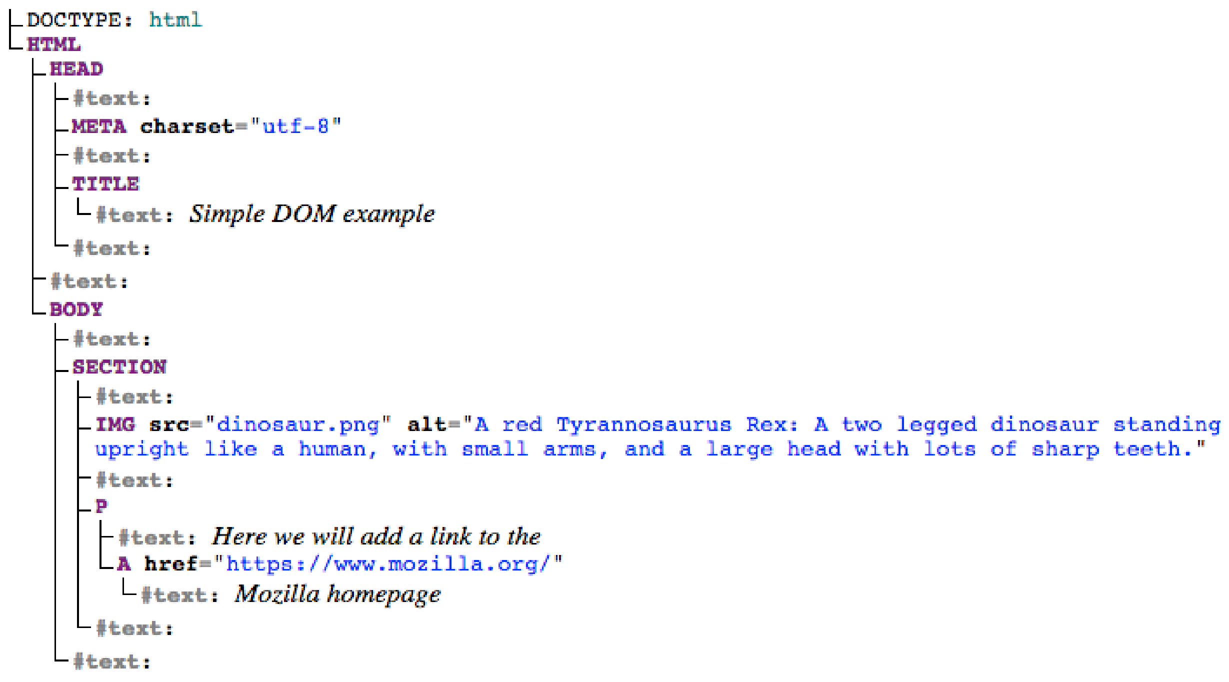
Developers like you can manipulate the DOM (the html) with JavaScript after the page has been rendered.

You can use this tool to visualize html as DOM https://software.hixie.ch/utilities/js/live-dom-viewer/

From this html:

```
<!DOCTYPE html>

<html>

  <head>

    <meta charset="utf-8">

    <title>Simple DOM example</title>

  </head>

  <body>

      <section>

        <img src="dinosaur.png" alt="A red Tyrannosaurus Rex: A two legged dinosaur standing upright like a human, with small arms, and a large head with lots of sharp teeth.">

        <p>Here we will add a link to the <a href="https://www.mozilla.org/">Mozilla homepage</a></p>

      </section>

  </body>

</html>
```

We get the following DOM:



```
├DOCTYPE: html
└HTML
  └HEAD
  │ ├#text:
  │ ├META charset="utf-8"
  │ ├#text:
  │ ├TITLE
  │ │ └#text: Simple DOM example
  │ └#text:
  ├#text:
  └BODY
    ├#text:
    ├SECTION
    │ ├#text:
    │ ├IMG src="dinosaur.png" alt="A red Tyrannosaurus Rex: A two legged dinosaur standing
    │ │ upright like a human, with small arms, and a large head with lots of sharp teeth."
    │ ├#text:
    │ ├P
    │ │ ├#text: Here we will add a link to the
    │ │ └A href="https://www.mozilla.org/"
    │ │   └#text: Mozilla homepage
    │ └#text:
    └#text:
```

You can see here that each element and bit of text in the document has its own entry in the tree — each one is called a **node**. You will also encounter various terms used to describe the type of node, and their position in the tree in relation to one another:

- **Element node**: An element, as it exists in the DOM.
- **Root node**: The top node in the tree, which in the case of HTML is always the HTML node (other markup vocabularies like SVG and custom XML will have different root elements).
- **Child node**: A node *directly* inside another node. For example, IMG is a child of SECTION in the above example.
- **Descendant node**: A node *anywhere* inside another node. For example, IMG is a child of SECTION in the above example, and it is also a descendant. IMG is not a child of BODY, as it is two levels below it in the tree, but it is a descendant of BODY.
- **Parent node**: A node which has another node inside it. For example, BODY is the parent node of SECTION in the above example.
- **Sibling nodes**: Nodes that sit on the same level in the DOM tree. For example, IMG and P are siblings in the above example.
- **Text node**: A node containing a text string

Some methods

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Simple DOM example</title>
  </head>
  <body>
      <section>
        <img src="dinosaur.png" alt="A red Tyrannosaurus Rex">
        <p>Here we will add a link to the <a
href="https://www.mozilla.org/">Mozilla homepage</a></p>
      </section>
        <script>
        const link = document.querySelector('a');
        link.textContent = 'Sergio was here';
        link.href = 'milk.com';
        </script>
  </body>
</html>
```

Document.querySelector() is the recommended modern approach, which is convenient because it allows you to select elements using CSS selectors. The above querySelector() call will match the first <a> element that appears in the document. If you wanted to match and do things to multiple elements, you could use Document.querySelectorAll(), which matches every element in the document that matches the selector, and stores references to them in an array-like object called a NodeList.

here are older methods available for grabbing element references, such as:

- Document.getElementById(), which selects an element with a given id attribute value, e.g. <p id="myId">My paragraph</p>. The ID is passed to the function as a parameter, i.e. const elementRef = document.getElementById('myId').
- Document.getElementsByTagName(), which returns an array-like object containing all the elements on the page of a given type, for example <p>s, <a>s, etc. The element type is passed to the function as a parameter, i.e. const elementRefArray = document.getElementsByTagName('p').

These two work in older browsers than the modern methods like querySelector(), but are not as convenient. Have a look and see what others you can find!

Creating and placing new nodes

```
const sect = document.querySelector('section');


//append something at the end of <section> before </section>

const para = document.createElement('p');

para.textContent = 'We hope you enjoyed the ride.';

sect.appendChild(para);




//append plain text instead of a <p> section (because <p> would be a breakline too)

const firstP = document.querySelector('p');

const text2 = document.createTextNode(' — the premier source for web development knowledge.');

firstP.appendChild(text2);
```

Moving and removing elements

```
//will move firstP to the end

sect.appendChild(firstP);


//wil remove firstP completely

sect.removeChild(firstP);
```

```
//but this easier (although not supported in older browsers)
firstP.remove();


//for old browsers youd have to do:
firstP.parentNode.removeChild(firstP);
```

Lecture

JavaScript in context

It is the most important language of the modern web stack. Today's **JavaScript runtime environments** are highly efficient and a number of them exist:

- V8 is Google's JavaScript engine (used in Chrome and other browsers).
- SpiderMonkey is Mozilla's engine and used in Firefox.
- Chakra is Microsoft's JavaScript runtime engine. It was originally used in Microsoft's Edge browser. In December 2018, Microsoft decided to adopt Chromium (Google's open-source browser project) as its JavaScript runtime in the browser. Chakra is still powering Windows applications that are written in HTML/CSS and JavaScript

the Node.js platform we cover in the next lecture is built on top of V8.

While the browser is the most obvious usage scenario for JavaScript runtime environments, they are also used in other areas such as microcontrollers.

Javascript is an *interpreted* language. That has advantages and disadvantages:

- advantage: quick upstart;
- disadvantage: the program overall *runs* slower than one written in a language (such as Java) requiring compilation.

Today's JavaScript engines both interpret *and* compile by employing so-called **just-in-time (JIT) compilation**. This means that JavaScript code that is run repeatedly such as often-called functions is eventually compiled and no longer interpreted.

JavaScript tracks ECMAScript, the scripting-language specification standardized by Ecma International. While JavaScript is the most popular implementation of the standard, other implementations or dialects (such as ActionScript) exist as well. ECMAScript is updated in a yearly cycle.

In this course we include just a few **ES6** features. It is worthwhile to know that many languages compile into JavaScript. Three of the most well-known languages are CoffeeScript, TypeScript and Dart; all three fill one or more gaps of the original JavaScript language. Once you work on complex projects in collaboration, these higher-level languages can make a difference, especially when it comes to debugging.

Here is one example of what TypeScript offers: JavaScript is a **dynamic language**, this means that you have no way of enforcing a certain **type** on a variable. Instead, a variable can hold any type, a String, a Number, an Array … but of course often you *know* what you want the type to be (for instance function parameters). It is useful to provide this knowledge upfront. TypeScript allows you to do that, by **enabling static type checking**.

Scripting overview

Server-side vs. client-side scripting

Server-side scripting refers to scripts that run on the **web server** (in contrast to the client). Executing the scripts on the server means they are **private** and only the result of the script execution is returned to the client - often an HTML document. The client thus has to trust the server's computations (there is no possibility to validate the code that ran on the server). Server-side scripts can access **additional resources** (most often databases) and they can use **non-standard language features** (when you run a server you know which type of software is installed on it and what type of language features it supports). At the same time, as all computations are conducted on the server, with many clients sending HTTP requests, this can quickly **increase the server's load**. As clients often only receive an HTML document as result of the computation, the app developer does not have to worry about clients' device capabilities - any modern browser can render HTML.

Client-side scripting on the other hand does not return the result of a computation to the client, but instead sends the script (and if necessary the data) to the client which enables the user to dig through the code (and expose you). But a clear advantage of client-side coding is **reduced server load**, as clients execute the scripts, though all data necessary for the scripts (which could be megabytes or gigabytes of data) need to be downloaded and processed by the client.

Modern browsers implement the IndexedDB API. IndexedDB provides a standard for an in-browser database that is transaction-based and stores key-value pairs persistently. While it cannot be queried with SQL directly, libraries such as JSstore exist that act as wrapper around IndexedDB to enable SQL-like querying. The storage limits are browser and device-dependent; in principle it is possible to store gigabytes of data within the browser's database. This can be very useful for instance for games (game objects are stored in the database) as well as client-side in-browser data processing tools.

The <script> tag

As the browser renders the page in a top-down fashion, with DOM elements created in the order they appear in the HTML document, we place the <script> tags right before the closing <body> tag. Thus, the DOM is already complete once the JavaScript is being executed. Interactivity based on the DOM should only start **after** the DOM has been fully loaded; if you decide to place your script's elsewhere, make use of the browser window's load event which is fired once the DOM has loaded.

Scoping, hoisting and this

Scoping

Scoping is the **context in which values and expressions are visible**. In contrast to other languages, JavaScript has very few scopes:

- local;
- global;
- block (introduced in **ES6**).

A *block* is used to group a number of statements together with a pair of curly brackets {...}.

The scopes of values and expressions depend on *where* and *how* they are declared:

	Scope
`var` declared within a function	local
`var` declared outside of a function	global
`let` (ES6)	block
`const` (ES6)	block
variable declaration without `var`/`let`/`const`	global

Before **ES6** there was no **block scope**, we only had two scopes available: local and global. Having only two scopes available resulted in code behavior that does not always seem intuitive. imagine we want to print out the numbers 1 to 10

```
for (var i = 1; i <= 10; i++) {
    console.log(i);
}
```

Let's now imagine that the print outs should happen each after a delay of one second.

```
for (var i = 1; i <= 10; i++) {

    setTimeout(function() {

        console.log(i);

    }, 1000);

}
```

When you run the code you will actually find it to behave very differently: after around one second delay, you will see ten print outs of the number 11.

setTimeout is executed ten times without delay. Defined within setTimeout is a **callback**, i.e. the function to execute when the condition (the delay) is met. After the tenth time, the for loop executes i++ and then breaks as the i<=10 condition is no longer fulfilled. This means i is 11 at the end of the for loop.

As i has **global scope** (recall: var i is declared outside a function), every single callback refers to the same variable. After a bit more time passes (reaching ~1 second), each of the function calls within setTimeout is now being executed. Every single function just prints out i. Since i is 11, we will end up with ten print outs of 11. Workaround:

```
function fn(i) {
    setTimeout(function() {
        console.log(i);
    }, 1000 * i);
}

for (var i = 1; i <= 10; i++)
    fn(i);
```

JavaScript passes the value of a variable in a function; if the variable refers to an array or object, the value is the **reference** to the object. Here, i is a number and thus every call to fn has its own local copy of i.

With the introduction of **ES6** and let, we no longer need this additional function construct as let has block scope and thus every i referred to within setTimeout is a different variable. This now works as expected:

```
for (let i = 1; i <= 10; i++)
    setTimeout( function() {
        console.log(i)
    }, 1000 * i)
```

If libraries used var we would ran out out of variable names to use. Therefore: always use let (or const).

Hoisting
to be executing a function (six() and seven() respectively) before they are defined.

```
var x = six();

//function declaration
function six(){
    return 6;
}

var y = seven();
```

```

//function expression
var seven = function(){
    return 7;
}


console.log(x+" - "+y);
```

while var x = six(); works (i.e., we can call six() before declaring it), var y = seven(); does not.

The difference lies in how we went about defining our six and seven functions:

- var seven = function(){...} is a **function expression** and is only defined when that line of code is reached.
- function six(){...} on the other hand is a **function declaration** and is defined as soon as its surrounding function or script is executed due to the **hoisting principle**: declarations are processed before any code is executed.

So function six(){...} has already been executed whereas var seven = function(){...} not. “Hoisting” is executing a declaration earlier. All declerations are hosted to the top. Expressions are not hoisted

This is not only the case for functions, also variable declarations are hoisted. Consider this example :

```
function f(){
    x = 5;
    y = 3;
};
f();
console.log(x);
console.log(y);
```

Variables x and y have global scope as they are not prefixed by var or let or const. And so the console output will be 5 and 3.

But what happens in this slightly changed piece of code?

```
function f(){
   a = 5;
   b = 3;
   var a, b;
};
f();
console.log(a);
console.log(b);
```

Now we will end up with a ReferenceError: a is not defined as the var a declaration at the end of function f is **hoisted** to the top of the function. The same applies to b. Both variables a and b thus have local scope and are not accessible to the console.log calls.

The keyword this

A **function's this keyword** behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

A property of an execution context (global, function or eval) that, in non–strict mode, is always a reference to an object and in strict mode can be any value.

Global context

In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true


a = 37;
console.log(window.a); // 37
```

non-strict mode

This refers to is dependent on *how* the function containing this was called.

We also have the option to set the value of a function's this independent of how the function was called, using the bind function.

In the code below each time, this refers to a different object. We called the function in three different ways:

- CASE 1: as a method of an object;
- CASE 2: as a property of the global window object;
- CASE 3: as a bound function.

```
//We assume execution in the browser's Web Console, we thus
//know the global window object exists (it is provided by the browser).

//h is now a property of the global `window` variable;
//it can also be accessed as window.h
var name = "Beat Saber";

function game(n){
    this.name = n;
    this.printName = function(){
        console.log(this.name);
    }
}

//CASE 1
//Creating a new object and calling the object's printName() function
var gameObj = new game("Astro Bot Rescue Mission");
gameObj.printName(); // this.name = "Astro Bot Rescue Mission"

//CASE 2
//Copying the printName function;
//printName is now a property of the global window object
var printName = gameObj.printName;
printName(); // this.name = "Beat Saber"

//CASE 3
//Fixing 'this' of the printName function
var boundPrintName = printName.bind({name: "Tetris Effect"});
boundPrintName(); // this.name = "Tetris Effect"
```

Arrow functions

In ES6 so-called **arrow functions** were introduced. Instead of writing :

```
 let sum = function(a,b){return a+b}
```

we can shorten it to: (but the way this behaves in this context is different to that of regular functions)

```
 let sum = (a,b) => {return a+b}
```

JavaScript design patterns

Design pattern is a “reusable solution”. Instead of everyone trying to figure out how to create objects, we use well-known **recipes** (a.k.a. design patterns) that were developed over time and apply them. There are many different design patterns, some are known to work across languages and some are specific to just a small subset of programming languages. What we cover in this lecture is mostly specific to JavaScript. Note that besides **design patterns**, there also exist **anti-patterns**, that are programming recipes which are popular but ineffective at tackling a recurring problem.

JavaScript objects

In JavaScript, **functions are first-class citizens** of the language. This means that **functions can be passed as parameters**, they can be **returned from functions** and they can be **assigned to a variable.**

```
//a function is passed as second parameter
function cutoffStringAt(c, fn){


  //a function is returned
      return function(s){
            fn(s.substring(0, c));
      }
}


//a function is assigned to a variable
var cutoffStringAt5 = cutoffStringAt(5, function(s){
      console.log(s);
});
```

This is quite a difference to Java for example, where functions cannot be passed around.

In JavaScript, **functions are also objects**. Apart from functions, JavaScript also comes with a number of other built-in objects: Strings, arrays and objects specific to the fact that JavaScript was developed to add interaction to HTML. One example is the document object, which only makes sense in the context of an HTML page. The browser is the host application in this case and provides the document object.

JavaScript objects can be created in different ways. This is very much unlike Java where there is essentially only one: you have a class, write a constructor and then use the new keyword to create an object.

When discussing object-oriented JavaScript it needs to be mentioned that since ES6 the class keyword exists in JavaScript. It does not act though in the same way as the class keyword in Java and acts mostly as *syntactic sugar*.

Object creation with new

```
var game = new Object();

game["id"] = 1;

game["player1"] = "Alice";

game.player2 = "Bob";

console.log( game["player2"] ); //prints out "Bob"

console.log( game.player1 ); //prints out "Alice"


game["won lost"] = "1 12";


game.printID = function(){

    console.log( this.id );

}

game["printID"](); // prints out "1"

game.printID(); //prints out "1"
```

“properties” = java attributes; “method” is like java “method”, methods are properties assigned by a funciton.

We first create an empty object with new Object() that we can then assign name/value pairs. Here, id, player1, etc. are the object’s **properties** and their name must be a valid JavaScript identifier (basically a string that does not start with a number). Note, that printID is also an object property, although it is often also referred to as a method because we define a function as part of an object. As seen here, JavaScript makes it easy to add methods, by assigning a function to the property of an object.

We have two ways to set and get an object’s properties: either through the bracket notation ([name]) or the dot notation (.name). It usually does not matter which notation to use, the exception here being property names with whitespaces. Property names that contain whitespaces must be set and accessed through the bracket notation (as in the example above for game["won lost"], the alternative game.won lost or game."won lost" will lead to a SyntaxError).

Object literals

There is a second way to create objects and that is via **object literals**. An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces:

```
var game = {

    id: 1,

    player1: "Alice",
    player2: "Bob",
    "won lost": "1 12",
    printID: function(){
        console.log(this.id);
    }
};
```

This time, "won lost" is a valid property name, but only if enclosed in quotation marks. *Instead of remembering when whitespaces are allowed, it is best to avoid them at all when assigning property names.*

Object literals can be complex, they can contain objects themselves:

```
var paramModule = {
    /* parameter literal */
    Param : {
        minGames: 1,
        maxGames: 100,
        maxGameLength: 30
    },
    printParams: function(){
        console.table(this.Param);
    }
};
```

Design pattern I: Basic constructor

First, let’s quickly recap what classes in Java offer us:

- we can encapsulate private members, i.e. members of the class that are not accessible externally;
- we define constructors that define how to initialize a new object;
- we define methods (public, protected, private).

Here is a **Java** example:

```
public class Game {
    private int id; /* encapsulate private members */


    /* constructor: a special method to initialize a new object */
    public Game(int id){
        this.id = id; /* this: reference to the current object */
    }

    public int getID(){
        return this.id;
    }

    public void setID(int id){
        this.id = id;
    }
}
```

And here is how we do the same in JavaScript:

```
function Game(id){
    this.id = id;
    this.totalPoints = 0;
    this.winner = null;
    this.difficulty = "easy";

    this.getID = function(){ return this.id; };
    this.setID = function(id){ this.id = id; };
}
```

We use functions as constructors and rely on this. We rely on the keyword new to initialize a new object similar to what you have already seen before:

```
var g1 = new Game(1);
g1.getID();
g1.setID(2);
var g2 = new Game(3);
```

```
//:cookie: ES6: object destructuring allows us to extract several object properties at once instead of one-by-one

var {totalPoints,winner,difficulty} = g1;

//:cookie: ES6: template literals to make string concatenations more readable

console.log(`This game reached ${totalPoints} points, was won by ${winner} and had difficulty ${diff}.`);
```

A common error is to forget the new keyword. The JavaScript runtime will not alert you to this mistake, in fact, the JavaScript runtime will execute the function as-is.

It turns out that without an object, in the browser context, this refers to the global window object (which represents the window in which the script is running). If you type window.id you will find the property to exist and hold the value of TWO. Of course, this is not desired as you may accidentally overwrite important properties of the window object.

Lesson here: be sure to know when to use new and what this refers to when.

JavaScript is a **prototype-based language** and here we can actually change our objects on the fly

```
function Game(id){

    this.id = id;

    this.getID = function(){ return this.id; };

    this.setID = function(id){ this.id = id; };

}


var g1 = new Game("1");

g1.player1 = "Alice";


var g2 = new Game("2");

g2.player1 = "Bob";


g1.printPlayer = function(){ console.log(this.player1); } //we add a method on the fly!

g1.printPlayer(); //prints out "Alice"


g2.printPlayer(); //TypeError: g2.printPlayer is not a function (method was added to g1 alone!)
```

```
g1.hasOwnProperty("printPlayer"); //true

g2.hasOwnProperty("printPlayer"); //false


g1.toString(); //"[object Object]" (we never defined it, but it is there)
```

Design pattern 2: Prototype-based constructor

objects come with **default methods**, and so the natural question should be, where do these methods come from? The answer is **prototype chaining**. Objects have a **secret pointer** to another object - the object's prototype. And thus, when creating for instance an object with a basic constructor as just seen, **the properties of the constructor's prototype are also accessible in the new object**. If a property is not defined in the object, the **prototype chain** is followed.

One of the issues in the basic constructor is that **objects do not share functions**. Often we do want objects to share functions and if a function changes that change should be reflected in all objects that have this property/method.

```
function Game(id){

    this.id = id;

}


/* new member functions are defined once in the prototype */

Game.prototype.getID = function(){ return this.id; };

Game.prototype.setID = function(id){ this.id = id; };


//using it

var g1 = new Game("1");

g1.setID("2"); //that works!


var g2 = new Game("2");

g2.setID(3); //that works too!


//g1 and g2 now point to different setID properties

//g2 follows the prototype chain

//g1 has  property setID

g1["setID"] = function(id){
    this.id = "ID"+id;
}
g1.setID(4);

console.log(g1.getID()); //ID4
```

All we have to do to make properties available to all objects is to use the .prototype property to walk up the prototype chain and assign a property to Game.prototype. When the two game objects are created and setID() is called, the JavaScript runtime walks up the prototype chain and “finds” the **first** property that matches the desired property name.

Changes made to the prototype are also reflected in existing objects

```
function Game(id){
    this.id = id;
}

/* new member functions are defined once in the prototype */
Game.prototype.getID = function(){ return this.id; };
Game.prototype.setID = function(id){ this.id = id; };

//using it
var g1 = new Game("1");
g1.setID("2"); //works
console.log( g1.getID() ); //prints out "2"

Game.prototype.setID = function(id){
    console.assert(typeof(id)=="number", "Expecting a number");
    this.id = id;
}

g1.setID("3");//leads to "Assertion failed: Expecting a number"
```

The prototype chaining allows us to set up **inheritance through prototyping**. This requires two steps:

1. Create a new constructor.
2. Redirect the prototype.

when using prototypical inheritance, always set up both the prototype and prototype.constructor; in this manner the wiring is correct, no matter how you will use the inheritance chain later on.

To finish off, here is a summary of the prototype-based constructor:

- Advantages:
    - **Inheritance is easy** to achieve;
    - **Objects share functions**;
- Issue:
    - All members are **public** and **any piece of code can be accessed/changed/deleted** (which makes for less than great code maintainability).

Example: Let's assume we want to inherit from Game to create a more specialized two-player game

```
function Game(id){
    this.id = id;
}

/* new member functions are defined once in the prototype */
Game.prototype.getID = function(){ return this.id; };
Game.prototype.setID = function(id){ this.id = id; };

/* constructor */
function TwoPlayerGame(id, p1, p2){
    /*
     * call(...) calls a function with a given `this` value and arguments.
     */
    Game.call(this, id);
    this.p1 = p1;
    this.p2 = p2;
}
```

```

/* redirect prototype */
TwoPlayerGame.prototype = Object.create(Game.prototype);
TwoPlayerGame.prototype.constructor = TwoPlayerGame;


/* use it */
var TPGame = new TwoPlayerGame(1, "Alice", "Bob");
console.log( TPGame.getID() ); //prints out "1"
console.log( TPGame.p1 ); //prints out "Alice"
```

Design pattern 3: Module

The module pattern has the following goals:

- **Do not declare any global variables** or functions unless required.
- Emulate **private/public** membership.
- Expose only the **necessary** members to the public.

We start with a concrete example of the **module pattern**

```
/* creating a module */
var gameStatModule = ( function() {


    /* private members */
    var gamesStarted = 0;
    var gamesCompleted = 0;
    var gamesAbolished = 0;


    /* public members: return accessible object */
    return {
        incrGamesStarted : function(){
            gamesStarted++;
        },
        getNumGamesStarted : function(){
            return gamesStarted;
        }
    }
})();


/* using the module */
gameStatModule.incrGamesStarted();
console.log( gameStatModule.getNumGamesStarted() ); //prints out "1"
console.log( gameStatModule.gamesStarted ); //prints out "undefined
```

Here we are defining a variable gameStatModule which is assigned a function expression that is immediately invoked. This is known as an *Immediately Invoked Function Expression* (or IIFE).

An IIFE itself is also a design pattern, it looks as follows:

```
(function () {
    //statements
})();
```

The function is **anonymous** (it does not have a name and it does not need a name, as it is immediately invoked) and the final pair of brackets () leads to its immediate execution. The brackets surrounding the function are not strictly necessary, but they are commonly used.

Going back to our gameStatModule, we immediately execute the function. The function contains a number of variables with function scope and a return statement. **The return statement contains the result of the function invocation**. In this case, an *object literal* is returned and this object literal has two methods: incrGamesStarted() and getNumGamesStarted(). Outside of this module, we cannot directly access gamesStarted or any of the other emulated *private* variables, all we will get is an undefined as the returned object does not contain those properties. The returned object though **has access** to them through JavaScript's concept of closures).

VSC offers a simple way to catch TypeError by adding the line //@ts-check

We can also set up VSC to perform type checking automatically for all JavaScript files.

Summarizing the module pattern:

- Advantages:
  - **Encapsulation is achieved**;
  - Object members are either public or private;
- Issues:
  - Changing the type of membership (public/private) takes effort (unlike in Java).
  - Methods added on the fly later cannot access emulated private members

Events and the DOM

In plain JavaScript we use the document object as our entry point to the DOM. It allows us to select DOM elements (either a group or a single element) in various ways:

- document.getElementById
- document.getElementsByClassName
- document.getElementsByTagName
- document.querySelector: returns the first element within the DOM tree according to a **depth-first pre-order traversal** that matches the selector
- document.querySelectorAll: returns all elements that match the selector

The last two ways of selecting DOM elements allow complex selector rules to be specified

```
<!DOCTYPE html>

<html>

    <head>

      <meta charset="UTF-8">

    </head>

    <body>

        <h1>Hide this button</h1>

        <button id="b">Hide me forever</button>


        <p></p>


        <h2>Hide this button too</h2>

        <button class="second">Hide me forever too</button>


        <script>


            /* addEventListener: allows multiple events to be attached to an
element */

            document.getElementById("b").addEventListener("click",
function(){

                document.getElementsByTagName("h1")[0].hidden = true;
            })


            document.getElementById("b").addEventListener("click",
function(){
                this.hidden = true;
            })


            /* ***** */


            /* onclick property: only one handler can be assigned to an
element at a time */
            document.querySelectorAll(".second")[0].onclick = function(){
                this.hidden = true;
            }
            document.querySelector("h2 ~ button").onclick = function(){
                document.getElementsByTagName("h2")[0].hidden = true;
            }
        </script>
    </body>
</html>
```

Here, two elements are rendered by the browser, a heading (*Hide this button*) and a button. Clicking on the button will hide both the heading and the button. Importantly, the browser provides us with an API to access keyboard and mouse events among others. As web developers we can start working from the point of *what happens when a click event has occurred?*. The code snippet thus shows off the **callback principle**, which we come across in all of JavaScript: we define **what happens *when* an event fires**. How can we connect events (e.g. a click) and actions (e.g. hiding an element)? Most often we will use targetEl.addEventListener(eventType, fn) which takes a function (fn) as input that is called whenever the desired eventTyp has been recorded for targetEl. In the example above, it becomes clear that in this manner multiple events can be attached to a single element. This is in contrast to using the onclick property to attach an event handler to an element: only one event can be attached to an element.

When creating a UI element that is responsive to user actions, we typically follow the following steps:

1. Pick a control, e.g. a button.
2. Pick an event, e.g. a click.
3. Write a JavaScript function: what should happen when the event occurs, e.g. an alert message may appear.
4. Attach the function to the event/control combination via addEventListener.

Document Object Model

The DOM is our entry point to interactive web applications. It allows use to:

- **Extract an element's state**:
    - Is the checkbox checked?
    - Is the button disabled?
    - Is a <h1> appearing on the page?
- **Change an element's state**:
    - Check a checkbox.
    - Disable a button.
    - Create an <h1> element on a page if none exists.
- **Change an element's style**:
    - Change the color of a button.
    - Change the size of a paragraph.
    - Change the background color of a web application.

document.getElementById / document.querySelector

We have a page with two elements: a button and a text box. A click on the button will show Hello World! in the text box. As you can see there are different ways (we have listed four here) of pinpointing a DOM element:

```
<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8">

        <title>Example 1</title>

    </head>

    <body>

        <button onclick="
```

```
            //var tb = document.querySelector('#out');
            //var tb = document.querySelector('input')
            //var tb = document.querySelector('input[type=text]')
            var tb = document.getElementById('out');
            tb.value = 'Hello World!'
        ">Say Hello World</button>
        <input id="out" type="text">
    </body>
</html>
```

This code is of course not ideal as we are writing JavaScript code in the middle of HTML elements, so let us refactor to achieve a better code separation:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Example 1</title>
        <script>
            /*
             * The DOM first has to fully load, before event listeners can be
added to its elements.
             */
            window.addEventListener('load', function(){

                /*
                 * We define what happens when button #b is clicked.
                 */
                document.getElementById("b").addEventListener('click',
function(){
                    document.getElementById("out").value = "Hello World!";
                });
            });
        </script>
    </head>

    <body>
        <button id="b">Say Hello World</button>
        <input id="out" type="text">
    </body>
</html>
```

creating new nodes

Note: as all our examples are simple, we will stick to document.getElementById to select DOM elements. In more realistic coding scenarios, document.querySelector and document.querySelector(All) will often be used.

HTML tags and content can be added dynamically in two steps:

1. Create a DOM node.
2. Add the new node to the document as a **child of an existing node**.

To achieve step 2, a number of methods are available to every DOM element:

Name	**Description**
appendChild(new)	places the new node at the end of this node's child list
insertBefore(new, old)	places the new node in this node's child list just before the old child
removeChild(node)	removes the given node from this node's child list
replaceChild(new, old)	replaces the old node with the new one

Examples:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Example 2</title>
    </head>

    <body>
        <button id="b">Add List Element</button>
        <ul id="u"></ul>

        <script>
            document.getElementById("b").addEventListener('click',
addElement);

            function addElement() {
              let ul = document.getElementById('u');
              let li = document.createElement('li');
              let count = ul.childElementCount+1;

              //Concatenating strings (before ES6)
              //li.innerHTML = 'List element ' + (ul.childElementCount+1) +'
';

              //:cookie: ES6: template literals to concatenate strings
              li.innerHTML = `List element ${ul.childElementCount+1}`;
              ul.appendChild(li);
            }
        </script>
    </body>
</html>
```

The HTML initially contains an **empty <ul> element**. Instead of directly adding <li> elements, we could have also added a single child <ul> to the <body> node and then started adding children to it. :flag: The code example also shows off template literals which were introduced in ES6: they allow us to plug variables (here: ul.childElementCount+1) into strings (demarked with *backticks*) in a more readable manner.

We can of course also remove elements

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <meta charset="UTF-8">
        <title>Example 2 (Removal)</title>
    </head>

    <body>
        <button id="bRemoveF">Remove first child</button>
        <button id="bRemoveL">Remove last child</button>
        <ul id="u">
          <li>Item 1</li>
          <li>Item 2</li>
          <li>Item 3</li>
          <li>Item 4</li>
          <li>Item 5</li>
        </ul>

        <script>

          document.getElementById("bRemoveF").addEventListener('click',
removeFirstChild);
          document.getElementById("bRemoveL").addEventListener('click',
removeLastChild);

          function removeLastChild() {
            let ul = document.getElementById('u');
            if(ul.childElementCount>0)
              ul.removeChild(ul.lastElementChild);
          }
          function removeFirstChild() {
            let ul = document.getElementById('u');
            if(ul.childElementCount>0)
              ul.removeChild(ul.firstElementChild);
          }
        </script>
    </body>
</html>
```

Important to note here is that there are often methods available for DOM elements which look similar, but are leading to quite different behaviors. Case in point: in the example we used ul.firstElementChild and ul.lastElementChild. Instead, we could have also used ul.firstChild and ul.lastChild. And this will work to - *at least with every second click*, as those methods also keep track of a node's children that are comments or text nodes, instead of just li nodes as we intend with our code.

Example 3: this

Event handlers are bound to the attached element's objects and the handler function "knows" which element it is listening to (the element pointed to by this). This simplifies programming as a function can serve different objects.

Imagine you want to create a multiplication app that has one text input box and three buttons, each with an arbitrary number on it. A click on a button multiplies the number found in the input with the button's number.

We could write three different functions and then separately attach each of them to the correct button :

```
document.getElementById("button10").addEventListener('click',
computeTimes10);

document.getElementById("button23").addEventListener('click',
computeTimes23);

document.getElementById("button76").addEventListener('click',
computeTimes76);
```

This leads to code duplication, is tedious, error prone and not maintainable (what if you need a hundred buttons …).

We can avoid code duplication with the use of this in order to *read out* the button's label :

```
<!DOCTYPE html>

<html>

    <head>
        <meta charset="UTF-8">
        <title>Example 3</title>
    </head>

    <body>
        <input type="text" id="input">
        <button id="button10">10 times</button>
        <button id="button23">23 times</button>
        <button id="button76">76 times</button>

        <script>
          /* loop over all buttons and attach an event handler */
          let buttons = document.getElementsByTagName("button");
          for(let b of buttons){
            b.addEventListener("click", computeTimes);
          }

          function computeTimes() {
            /*
             * this.innerHTML returns to us "N times",
             * parseInt() then strips out the " times" suffix
             * as it stops parsing at an invalid number character
             */
            let times = parseInt(this.innerHTML);
            let input = parseFloat(document.getElementById("input").value);
            let res = times * input;
            alert("The result is " + res);
          }
        </script>
    </body>
```

```
</html>
```

Check example

Depending on which button is clicked, this refers to the corresponding DOM tree element and .innerHTML allows us to examine the label text. The parseInt function is here used to strip out the " times" string suffix, forcing a conversion to type number.

Example 4: mouse events

A number of different mouse events exist (mouseup, mousedown, mousemove, …) and some are defined as a series of simpler mouse events, e.g.

- A click of the mouse button in-place consists of:
    - mousedown
    - mouseup
    - click
- A click of the mouse button while moving the mouse ("dragging") consists of:
    - mousedown
    - mousemove
    - …
    - ..
    - mousemove
    - mouseup

Let's look at an example of mouseover and mouseout. A timer starts and remains active as long as the mouse pointer hovers over the button element and it resets when the mouse leaves the element. Each of the three buttons has a different timer speed:

```
<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8">

        <title>Example 4</title>

    </head>


    <body>
```

```
<button style="width:500px" id="b1">0</button>
<br>
<button style="width:500px" id="b10">0</button>
<br>
<button style="width:500px" id="b100">0</button>

  <script>

    const buttons = document.getElementsByTagName("button");
    for(let b of buttons){
      b.addEventListener("mouseover", atMouseOver);
      b.addEventListener("mouseout", atMouseOut);
    }

    let intervals = {};

    function updateNum(button){
      let num = parseInt(button.innerHTML) + 1;
      button.innerHTML = num;
    }

    function atMouseOver() {
      let incr = parseInt(this.id.substr(1));
      intervals[this.id] = setInterval(updateNum, 1000/incr, this);
    }

    function atMouseOut()
    {
      clearInterval(intervals[this.id]);
      this.innerHTML = "0";
```

```
          }
        </script>
    </body>
</html>
```

Mouse events can be tricky, the more complex ones are not consistently implemented across browsers.

Check it here

Example 5: a crowdsourcing interface

Here is another event that can be useful, especially for text-heavy interfaces: onselect. Here, we have an interface with a read-only text that the user can select passages in. If enough passages have been selected, the user can submit the selected passages:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Example 5</title>
    </head>

    <body>
      <form>
          <p><em>Task: Write down / mark the 3 most important information nuggets</em>
            <br>

            <textarea id="ta" cols="50" rows="5" readonly>"Hobey Baker (1892-1918) was an American amateur athlete of the early twentieth century, widely regarded by his contemporaries as one of the best athletes of his time."
            </textarea>
            <br>

            <label>Nugget 1: <input type="text" id="n1" autocomplete="off"></label>
            <br>
            <label>Nugget       2:       <input       type="text"       id="n2"
autocomplete="off"></label><br>

            <label>Nugget       3:       <input       type="text"       id="n3"
autocomplete="off"></label><br>

            <button hidden="hidden" id="b">Submit Answers</button><br>

        </form>


        <script>

          document.getElementById("ta").addEventListener('select',
updateNuggets);


          function updateNuggets() {


            let n1El = document.getElementById('n1');

            let n2El = document.getElementById('n2');

            let n3El = document.getElementById('n3');


            /* what has already been selected as nuggets */

            let n1Val = document.getElementById('n1').value;

            let n2Val = document.getElementById('n2').value;

            let n3Val = document.getElementById('n3').value;


            console.log(n1Val);

            console.log(n2Val);

            console.log(n3Val);

            /* extract the selected string */

            let selected = null;

            let myTextArea = document.getElementById('ta');

            if (myTextArea.selectionStart != undefined)

            {

              let p1 = myTextArea.selectionStart;
              let p2 = myTextArea.selectionEnd;
              selected = myTextArea.value.substring(p1, p2);
            }

            //if the selected phrase is already in a nugget, remove it
            if(selected==n1Val) {n1El.value = "";}
            else if(selected==n2Val){n2El.value = "";}
            else if(selected==n3Val){n3El.value = "";}
            //if the first nugget is empty, add it
            else if(n1Val.length==0){n1El.value = selected;}
            //if the second nugget is empty, add it
            else if(n2Val.length==0){n2El.value = selected;}
            //third nugget is treated differently, as now the button becomes
unhidden
            else if(n3Val.length==0)
            {
              n3El.value = selected;
              document.getElementById('b').hidden = false;
            }
            else {
              alert('You have selected three information nuggets. Either
unselect one or manually empty text box.');
            }
          }
        </script>
    </body>
</html>
```

Check here

Example 6: a typing game

The last example is a typing game . Given a piece of text, type it correctly as fast as possible. The interface records how many seconds it took to type and alerts the user to mistyping. In this example we make use of the keypress event type. We start the timer with setInterval (incrementing once per second), which returns a handle that we can later pass to clearInterval to stop the associated callback from executing (thus stopping the clock). You see here too how to determine which key was pressed: we use the KeyboardEvent's key property. The site keycode.info makes it easy for you to find out what KeyboardEvent property (there are several, we care about key) each key of your keyboard assigned to!

In this example we do do make slight use of CSS (to flash a red background and alter the color of the timer in the end), you can recognize those line on the .style properties.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Example 6</title>
    </head>

    <body>
      <form id="f">
          <p>
            <em>Task: Type out the following text correctly:</em>
            <br>

            <textarea    id="given"    cols="50"    rows="5"    readonly=""
autocomplete="off">H.  Baker  was  an  American  amateur  athlete  of  the  20th
century.</textarea>
            <br>

            <textarea    id="typed"    cols="50"    rows="5"    readonly=""
autocomplete="off"></textarea>
            <br>

            <span id="timer">0 seconds</span>
          </p>
      </form>
      <script>
          let currentPos = 0;
          let givenText = "given";
          let typedText = "typed";
          let timerLog = "timer";
          let intervals = {};

          let typedTextEl = document.getElementById(typedText);
          let timerEl = document.getElementById(timerLog);

          document.getElementById("typed").addEventListener('keypress',
checkTextAtKeyPress);

          //e refers to the event (we need it to extract the char typed)
          function checkTextAtKeyPress(e) {

            let textToType = document.getElementById(givenText).value;

            //we reached the end, do nothing
            if(currentPos >= textToType.length) {return;}

            let nextChar = textToType.charAt(currentPos);

            let keyPressed = e.key;
            console.log("Key pressed: "+keyPressed);

            //correct key was pressed
            if(nextChar==keyPressed) {
              //CSS is used here to "style" the text box
              typedTextEl.style.backgroundColor="white";
              typedTextEl.value = textToType.substring(0,currentPos+1);

              currentPos++;

              //first time key was pressed, start counter
              if(currentPos==1) {
                intervals[this.id]=setInterval(function(){
                    let t = 1 + parseInt(timerEl.innerHTML);
                    timerEl.innerHTML = t +" seconds";
                }, 1000);
              }

              //we reached the end
              if(currentPos==textToType.length) {
                clearInterval(intervals[this.id]);
                //CSS is used here to "style" the text box
                timerEl.style.color="orange";
              }
            }
            //incorrect key
            else {
              //CSS is used here to "style" the text box
              typedTextEl.style.backgroundColor="salmon";
            }
```

```
        }
      </script>
  </body>
</html>
```

To conclude this DOM section, here is an overview of important keyboard and text events:

Event	Description
blur	element loses keyboard focus
focus	element gains keyboard focus
keydown	user presses key while element has keyboard focus
keypress	user presses and releases key while element has keyboard focus (a problematic event)
keyup	user releases key while element has keyboard focus
select	user selects text in an element

In the six DOM interaction examples shown here, we did not have to add a lot of code and thus ***for presentation purposes*** *I chose to keep things lean and move all JavaScript code into the HTML file. When writing code that go beyond the few lines I am showcasing here, this should be avoided: your JavaScript code should be in dedicated files and the object-oriented programming paradigm should be employed as much as possible.*

Self-check

Here are a few questions you should be able to answer after having followed the lecture and having worked through the required readings:

```
for (var i = 1; i <= 10; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}
```

What will be the result of executing the above piece of JavaScript in the browser?

- After a one second delay, ten printouts of the number 11 appear on the console. Reason, var i is global scoped.

What does JavaScript's hoisting principle refer to?

- Declarations are processed before any code is executed.

A server-side application uses sessions instead of cookies to track users. What is a common approach to determine the end of a session?

- If x seconds have passed without a request from the client, the server ends the session.

In the module design pattern, methods added on the fly to an already created object cannot access emulated private members.

- True

```
function A(x){

  var y = x * 2;

  return function(y){

    var z = y * 3;

    return function(z){

      return x + y + z;

    }

  }

}

console.log( A(3)(4)(5) );
```

What is the output on the web console when running the above snippet of JavaScript in the browser?

- 12 (currying)

```
var message = "Toy kitchen";
var price = "89.90";
var deal1 = {
    message: "Peppa Pig",
    details: {
        price: "29.95",
        getPrice: function(){
            console.log(this.price);
        }
    }
}

deal1.details.getPrice();
```

What is the output on the web console when running the above snippet of JavaScript in the browser?

- 29.95

Example of a private attribute and a public getter in Javascript:

```
function Car() {

  var speed = 10;


  this.getSpeed = function() {

    return speed;

  }

}


var myCar = new Car();

myCar.speed; // this doesn't work, there is no myCar.speed

myCar.getSpeed() // this works because we explicitly made getSpeed 'public'
```

Lecture 4. Node.js: JavaScript on the server

Suggested Activity – Microsoft's Node tutorial

https://docs.microsoft.com/en-us/learn/modules/intro-to-nodejs/2-what

Node.js, or Node for short, is an open-source, server-side JavaScript runtime environment. You can use Node.js to run JavaScript applications and code in many places outside of a browser, such as on a server.

Node.js is a wrapper around a JavaScript engine called V8 that powers many browsers, including Google Chrome, Opera, and Microsoft Edge. You can use Node.js to run JavaScript by using the V8 engine outside of a browser. Node.js also contains many V8 optimizations that applications running on a server might need. For example, Node.js adds a **Buffer** class that allows V8 to work with files. This feature makes Node.js a good choice for building something like a web server.

Even if you've never used JavaScript as a primary programming language, it might be the right choice for writing robust, modular applications. JavaScript also offers some unique advantages. For example, because browsers use JavaScript, you can use Node.js to share logic like form-validation rules between the browser and the server.

JavaScript has become more relevant with the rise of single-page applications and supports the widely used JavaScript Object Notation (JSON) data-exchange format. Many NoSQL database technologies such as CouchDB and MongoDB also use JavaScript and JSON as a format for queries and schemas. Node.js uses the same language and technologies that many modern services and frameworks use.

You can build the following types of applications by using Node.js:

- HTTP web servers
- Microservices or serverless API back ends

- Drivers for database access and querying
- Interactive command-line interfaces
- Desktop applications
- Real-time Internet of Things (IoT) client and server libraries
- Plug-ins for desktop applications
- Shell scripts for file manipulation or network access
- Machine learning libraries and models

Node.js is fast, high-performing, and able to handle real-time applications and heavy data flows.

The Node.js environment also offers an npm registry that you can use to share your own Node.js library.

npm is the world's largest **Software Registry**, for the JavaScript language. The registry contains over 800,000 **code packages**. **Open-source** developers use **npm** to **share** software. Many organizations also use npm to manage private development. Using npm is Free. You can download all npm public software packages without any registration or logon. **npm** includes a **CLI** (Command Line Client) that can be used to download and install software: Windows Example

```
C:\>npm install <package>
```

Node.js can interpret and run JavaScript code on a host machine outside of a browser, the runtime has direct access to the operating system I/O, file system, and network.

Node.js is based on a single-threaded event loop. This architecture model efficiently handles concurrent operations. *Concurrency* means the ability of the event loop to perform JavaScript callback functions after completing other work.

In this architecture model:

- *Single-threaded* means that JavaScript has only one call stack and can do only one thing at a time.
- The *event loop* runs the code, collects and processes events, and runs the next subtasks in the event queue.

A *thread* in this context is a single sequence of programmed instructions that the operating system can manage independently.

In Node.js, I/O operations such as reading or writing to a file on the disk, or making a network call to a remote server, are considered blocking operations

Node.js REPL

Node.js has a built-in read-eval-print loop (REPL) mode that's useful for quick code evaluation and experimentation. REPL mode is an interactive console environment where you can enter JavaScript code and have Node.js interpret and run the code and then print the output.

The Node.js REPL mode works as follows:

- **Read**: Reads and parses the user's JavaScript code input (or shows an error if the code is invalid).
- **Eval**: Evaluates the entered JavaScript code.
- **Print**: Prints the computed results.
- **Loop**: Loops and waits for the user to enter a new command (or exits if the user enters **ctrl-c** twice).

To start REPL mode, run the node program in Azure Cloud Shell:

```
node
```

The REPL environment opens. You should see the following output:

```
>
```

Try typing the code console.log('Hello World, from the REPL.') inside the REPL console and view the output. This code prints a "Hello World, from the REPL." message in the REPL output:

```
> console.log('Hello World, from the REPL.')

Hello World, from the REPL.
```

To exit the REPL environment, enter **ctrl-c** twice:

```
>

(To exit, press ^C again or type .exit)

>
```

Create a Node.js script

Node.js also supports running code from files.

1. Using Azure Cloud Shell, open a new editor by entering the code command:

```
code
```

2. Inside the shell code editor, create a file by entering the following code:

```
console.log('Hello World, from a script file.');
```

3. Save the file by entering Ctrl+S (Windows, Linux) or Cmd+S (macOS) under index.js.
4. In the command-line shell, enter node followed by the name of the file, in this case index.js:

```
node index.js
```

You should see the following output:

```
$ node index.js

Hello World, from a script file.
```

You have now run your first Node.js JavaScript code.

Summary

Node.js is a runtime that can run JavaScript applications and code outside of a browser.

Node.js is a good choice for building and running JavaScript applications. This can range from building application servers to running real-time applications on Internet of Things (IoT) embedded devices.

Lecture

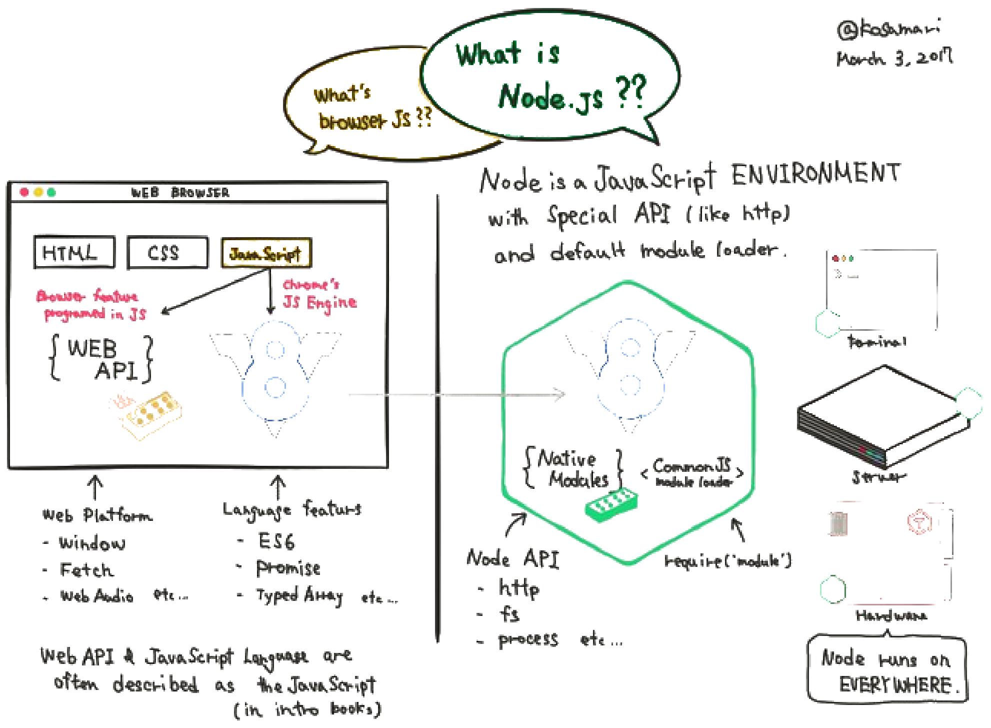
Introduction to Node.js

"Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an **event-driven**, **non-blocking** I/O model that makes it lightweight and **efficient**. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world."

Important milestones between 2008 and 2018 are:

- 2008: Google's JavaScript execution engine (**V8**) is open-sourced (if you are interested what happened in the 10 years since then, check out this blog post from the V8 team).
- 2009: Node.js is released. It builds on V8.
- 2011: Node.js' package manager (npm) is released.
- December 2014: Node.js developers are unhappy with the stewardship of the project and fork io.js.
- May 2015: io.js merges back with Node.js. The Node.js Foundation is set up. It has been steering Node's development ever since.
- 2017: Node becomes a **first-class citizen of V8**. This means that no V8 code change is allowed to break Node.
- August 2018: Node.js has been downloaded more than one billion times.
- May 2018: Deno was first introduced by the creator of Node.js (Ryan Dahl) as an improved alternative to Node.js. Though it has gained traction, Node.js remains the leading JavaScript-based server-side runtime.

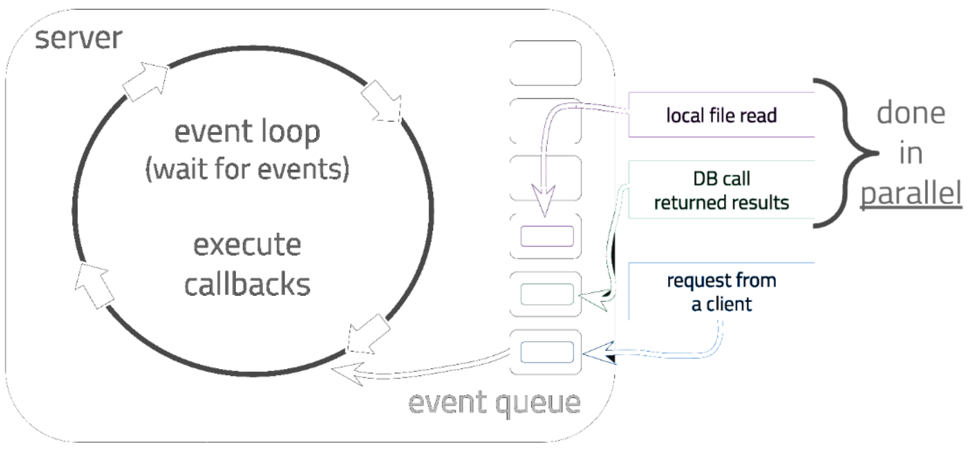
Node.js vs. client-side JavaScript



Shown on the left here is Google Chrome and its JavaScript runtime engine V8 (other browsers may make use of other JavaScript runtime engines). V8 is also the core of Node.js. The main difference between the browser and Node.js are the available APIs. As an example, while JavaScript code deployed in the browser cannot access the local file system for security reasons, Node.js can. It is also important to realize that Node.js runs on many different types of hardware.

Event-driven and non-blocking

One of the core concepts of Node is the **event loop**. Node.js is a single-threaded application that executes **callbacks** in response to an occurring event. Developers write those callbacks. The event loop waits for events to enter the **event queue** and once they have, the events are processed in their order of arrival, i.e. their respective callbacks are executed.



Here, despite the **single-threaded nature** of Node.js, several things are seemingly going on in parallel: a file is read from disk, a database is queried while at the same time an HTTP request from a client comes in. The reason for this is the **asynchronous** nature of file reads, network requests and so on (basically: I/O requests). While the event loop is executed in a single thread, Node maintains a *pool of threads* in order to process I/O requests in parallel. So, it is more correct to say that **Node's event loop is single-threaded**.

File reading example:

```
var fs = require('fs');


//we assume there is a file styles.css in the current directory
fs.readFile('./styles.css', 'utf8', function(err, data) {
  if (err) throw err;
  console.log(data);
});
```

The method readFile takes three parameters: the path of the file to read, the file encoding type, and a **callback function**: the function to execute when the file reading operation has completed. The callback function has two parameters: an error object and a data object. If the file reading operation failed for some reason, we throw the error, otherwise we print out the data to the console. Once the Node runtime encounters this code snippet, it will execute fs.readFile(....) and return immediately to execute the next line of code (this is called **non-blocking**). What happens to the file read operation? The Node runtime has access to a pool of I/O threads and once a thread has completed the file reading operation, an event is entered into the event loop to then execute the callback (in our case printing out the contents to the console).

The Node runtime can also read file contents from disk in a **blocking** manner:

```
let data = fs.readFileSync('./styles.css', 'utf8');
```

The Node runtime will wait until the file read is complete, return the file content in the data variable and then continue with the next line of code. If the file to read is large, this operation will take time and nothing else is executed in the meantime (because the code is **blocking**).

Blocking I/O	**Non-blocking I/O**
1. read request	1. read request
2. process request and access the database	2. process request and make a **callback** to access the database
3. **wait** for the database to return data and process it	3. **do other things**
4. process the next request	4. when the callback returns process it

Node is non-blocking I/O. I/O requests usually require a waiting time - waiting for a database to return results, waiting for a third-party web service or waiting for a connection request. By using callbacks, the Node runtime does not have to wait for slow I/O operations to complete.

Node's focus on making so-called **I/O bound programs** (that is, programs constrained by data access where adding more CPU power will not lead to speedups).

As the typical web application is indeed I/O bound, Node.js has become a popular choice of server-side framework. Another positive side effect of Node is the *reuse* of the language: instead of learning JavaScript for the client-side and PHP (or another language) for the server-side part of an application, we restrict ourselves to a single language and can even *share code* between client and server efficiently.

Node.js in examples

Watching a file for changes

Script that watches a given file for changes and alerts us to any changes by printing out a message on the terminal.

```
const fs = require('fs');


/*
 * Let's make the console output a bit more colorful.
 * These oddly looking strings are ANSI escape codes, they
 * allow us to signal a change of color to the terminal.
 * This is not Node-specific.
 */
let colors = {
    error: "\x1b[31m",
    ok: "\x1b[32m",
    changed: "\x1b[34m",
    reset: "\x1b[0m"
}


//beautifying the console output
function print(type, s){
    console.log(colors[type],s,colors["reset"]);
}
```

```
if(process.argv.length<3){
    print("error", "File to watch required as command line argument.");
    process.exit(1);
}

let file = process.argv[2];

fs.access(file, fs.constants.F_OK, function(err){
    if(!err){
        print("ok", "Now watching " + file);
        fs.watch(file, function(){
            print("changed", "File changed");
        });
    }
    else {
        print("error", "File to watch does not exist!");
        process.exit(1);
    }
});
```

- Line 1 provides us with access to the filesystem object. The corresponding **Node module** is fs. A module is a **self-contained** piece of code that provides **reusable functionality**. The function require() usually returns a JavaScript. In this case, fs is our entry point to the file system.
- You should have recognized that fs.watch is used with two arguments: the path to the file to watch and a **callback** function that is executed when a file change has occurred. The callback function is anonymous (though nothing prevents us from giving the function a name) and executed asynchronously. fs.access (used to determine whether the file exists) takes three arguments in our example.
- As the filesystem access requires operating system specific code, the behavior can vary across file systems; the underlying operating system calls are outlined in the fs.watch documentation.

A note on Node terminology: you will often find references to **Node modules** and **Node packages**. They differ slightly in meaning:

- A **module** is any file or directory that can be loaded by require().
- A **package** is any file or directory that is described by a package.json file.

Our watching.js script above can be considered a module, but not a package, as so far we have not seen the need for a package.json file.

Low-level networking with Node.js

Node.js was originally designed for I/O bound programs, in particular programs requiring **networking** functionalities. For this reason, Node.js has built-in support for **low-level** socket connections (TCP sockets in particular). Sockets are defined by IP address and port number.

TCP socket connections have **two endpoints**:

1. One endpoint **binds** to a numbered port.
2. The other endpoint **connects** to a port.

An analogous example of TCP socket connections are phone lines: One phone *binds* to a phone number. Another phone tries to call that phone. If the call is answered, a connection is established.

The check for changes file but with a network:

```
const fs = require('fs');
const net = require('net');

//Let's make the console output a bit more colorful
let colors = {
    error: "\x1b[31m",
    ok: "\x1b[32m",
    changed: "\x1b[34m",
    reset: "\x1b[0m"
}

function print(type, s){
    console.log(colors[type],s,colors["reset"]);
}

if(process.argv.length<4){
    print("error", "Two command line arguments required: [file to watch] [port]");
    process.exit(1);
}

let file = process.argv[2];
let port = process.argv[3];
let server = net.createServer(function(connection){

    //what to do on connect
    print("ok","Subscriber connected!");

    connection.write(colors["ok"] + "Now watching " + file + " for changes.\n"
+ colors["reset"]);

    var watcher = fs.watch(file, function(){
        connection.write(colors["changed"] + "File " + file + " has changed: "
+ Date.now() + ".\n" + colors["reset"]);
    });

    //what to do on disconnect
    connection.on('close', function(){
        print("ok","Subscriber disconnected!");
        watcher.close();
    });
});

server.listen(port, function(){
    print("ok","Listening to subscribers ...");
});
```

- We here make use of the net module which provides an asynchronous network API. It is one of the core modules and comes prepackaged in Node - we will later see how to install non-core modules.
- The method net.createServer returns a server object and takes as argument a callback function, which is invoked when another endpoint connects.
- With server.listen(port) we **bind** our server to a specific port.
- The callback function contains both client-side and server-side output. All client-side output is "written" to the connection object (which takes care of all the low-level details of actually sending the data), while as we already know the server-side messages are written out to our console object.

To execute the code we now require two command line arguments: the file to watch and the port number we want the server object to listen to:

```
node tcp.js test.txt 3000
```

Any port number between 1024 and 65536 is a good one (ports 1 to 1023 are *system ports* you should avoid them), as long as no other program has already bound to it. If you try to use the same port as another program, Node will throw an EADDRINUSE error (which means *Error: address is in use!*).

Ok, we just started the server. It is now waiting for clients/subscribers on port 3000. The next obvious question is how to start up clients. For this exercise, we will use telnet. Open another terminal (the server has to keep running of course) and type:

```
telnet localhost 3000
```

localhost is the hostname, you can also replace it by IP address 127.0.0.1, which is typically assigned to localhost. It basically means *this computer*, as we start our server in the same machine as our client.

Creating a Hello World! web server with Node.js

By now you will have realized, **Node.js is not a web server**. Node.js provides the **functionality** to implement one!

Let's do that. We start off with another minimal task: whenever our server receives an HTTP request, it should reply with a *Hello World!* string in the HTTP response.

```
const http = require("http");


let port = process.argv[2];


let server = http.createServer(function (req, res) {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello World!");
    console.log("HTTP response sent");
});


server.listen(port, function () {
    console.log("Listening on port " + port);
});
```

Let's assume that script is stored in web.js. This means we can start the script by typing

```
node web.js 3000
```

into the terminal. Now that we are moving up in the network stack and work with HTTP, we can use the browser as our client. Open your browser and use the following URL in the browser's address bar: localhost:3000, or localhost:3000/hello or any other path. The port number in the URL should match the port your server binds to. Each time, you should see a *Hello World!* displayed in the browser window.

- We here utilize Node's core HTTP module which provides us with all necessary functionalities related to HTTP.

- We create a **web server** with the call http.createServer.
- The **callback** function we define has two parameters: an **HTTP request object** and an **HTTP response object**. The callback is executed when an HTTP request comes in.
- Within the callback function we create an HTTP response (making use of the response object, which provides us with the necessary methods such as writeHead to write HTTP header information in JSON format) and sending it with the call to res.end.

To make the code more modular, we can apply some refactoring, which shows off the function-as-parameter paradigm once more.

```
const http = require("http");

let port = process.argv[2];

function simpleHTTPResponder(req, res){
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello World!");
    console.log("HTTP response sent");
}

let server = http.createServer(simpleHTTPResponder);

server.listen(port, function () {
    console.log("Listening on port " + port);
});
```

The task is now to return a greeting for the /greetme path and a 404 Not Found error otherwise. If the URL query (i.e. the part of the URL that assigns values to parameters) contains a parameter named name, we greet by name, and otherwise use Anonymous:

```
const http = require("http");
const url = require("url");

let port = process.argv[2];

function simpleHTTPResponder(req, res) {

    //parse the URL
    var uParts = url.parse(req.url, true);

    //implemented path
    if (uParts.pathname == "/greetme"){
        res.writeHead(200, { "Content-Type": "text/plain" });

        //parse the query
        var query = uParts.query;
        var name = "Anonymous";
        if (query["name"] != undefined){
            name = query["name"];
        }

        res.end(" Greetings "+name);
    }
    //all other paths
    else {
        res.writeHead(404, { "Content-Type": "text/plain" });
        res.end("Only /greetme is implemented.");
    }
}

let server = http.createServer(simpleHTTPResponder);

server.listen(port, function () {
    console.log("Listening on port " + port);
});
```

The code showcases how to make use of another core Node module, the url module; it provides support for URL resolution and parsing. Note that instead of having to manually specify the behaviour for each path, when we do implement a web server, in the Node.js community that most often means making use of the **Express framework**.

Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Express provides a thin layer of fundamental web application features, without obscuring Node.js features …

Node.js has a small core code base; it comes with a number of core modules included such as http and url. Express is not one of the core modules (though it is certainly among the most popular non-core modules with more than 13 million downloads per week) (as of September 2020) and needs to be installed separately.

Once we have found the Node packages we need for our project, how do we go about setting them up for our project? First of all, in order to create a project, we need to create a folder, e.g. node-express-ex. cd into the folder (which is still empty) and execute the following command:

```
npm init -y
```

Now check with the command ls what happened. We have a package.json file in the folder which now contains basic information about the project, all filled with defaults. If you do not want the defaults, run npm init instead, the -y option ensures that you are not asked any questions. Let’s assume you want to install the Express package. Run the following command:

```
npm install express --save
```

We have just installed Express in the folder node_modules, which has automatically been created within your current folder; as Express depends on many other packages, you will find more than just Express in there. The --save option ensures that the package.json file is altered.

```
{
  "name": "node-express-ex",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Note the dependencies entry. It means, that the project is now dependent on Express. Without the --save option, the installation of Express would have occurred as well, however, the package.json file would remain unaltered and the dependencies would not have been saved. When package.json is properly maintained, the application can be installed by anyone with npm install. There is one more file in your folder, package-lock.json, that we are skipping over. If you look into it, you will find it much more detailed than package.json, it provides step by step instructions of how the node_modules tree was generated.

Greetings Express

- The call express() returns an object (usually named app for application) which is our way of making use of Express' functionalities.
- We define three so-called **URL routes**: /greetme, /goodbye and /*, with the latter representing all possible routes. When an HTTP request comes in, the Express framework determines which route to execute - the routes are evaluated **in order of appearance** and the way we set up the code, it is only possible for a single route to be activated per request. Since /greetme appears before /* in our list of routes, we see the intended greeting. If we would move the /* route to be the first in the file, the only response we would ever see, no matter the URL path, would be *Not a valid route ….*

- Lastly, it is no longer necessary to create HTTP headers. Express manages this tedious task for us, all we have to do is call the send() method of the HTTP response object.

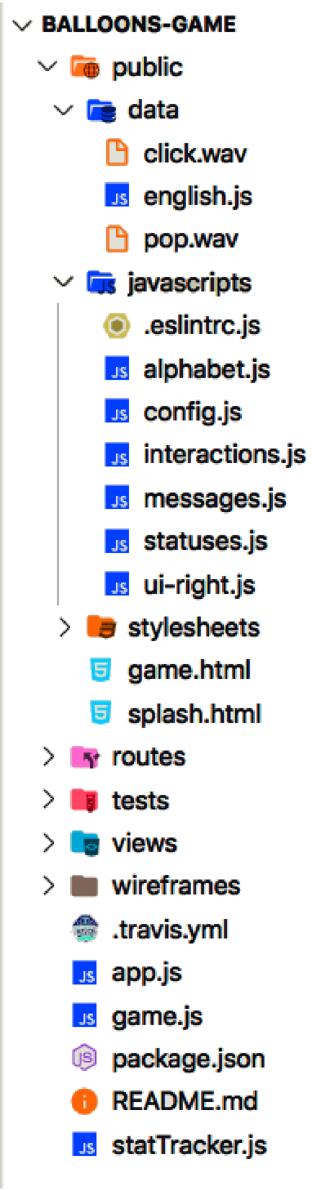
```
const express = require("express");
const url = require("url");
const http = require("http");

let port = process.argv[2];
const app = express();
http.createServer(app).listen(port);

let htmlPrefix = "<!DOCTYPE html><html><head></head><body><h1>";
let htmlSuffix = "</h1></body></html>";

app.get("/greetme", function(req, res){
    var query = url.parse(req.url, true).query;

    var name = (query["name"] != undefined) ? query["name"] : "Anonymous";

    res.send(htmlPrefix + "Greetings " + name + htmlSuffix);
})

app.get("/goodbye", function(req, res){
    res.send(htmlPrefix + "Goodbye to you too!" + htmlSuffix);
})

app.get("/*", function(req, res){
    res.send("Not a valid route ...");
})
```

This code though is still not maintainable, writing HTML code within a Node.js script is poor form and error-prone. For **static files**, i.e. files that are *not* created or changed on the fly, e.g. CSS, client-side JavaScript, HTML, images and audio files, Express offers us a very simple solution. A single line of code is sufficient to serve static files.

```
app.use(express.static(__dirname + "/static"));
```

For this code snippet to work, all static files need to be contained in the directory static (you can of course decide to name that directory differently); __dirname is a globally available string in Node that contains the directory name of the current module. In response to an HTTP request, Express first checks the static files for a given route - if a static file is found, this is served, otherwise the **dynamic routes** are checked until a fitting route is found.

A complete web application



```
BALLOONS-GAME
  public
    data
      click.wav
      english.js
      pop.wav
    javascripts
      .eslintrc.js
      alphabet.js
      config.js
      interactions.js
      messages.js
      statuses.js
      ui-right.js
    stylesheets
    game.html
    splash.html
  routes
  tests
  views
  wireframes
  .travis.yml
  app.js
  game.js
  package.json
  README.md
  statTracker.js
```

Having all the pieces in place (knowledge of HTML, client-side JavaScript, Node.js scripting), we can now write a complete web application. A good development strategy is the following:

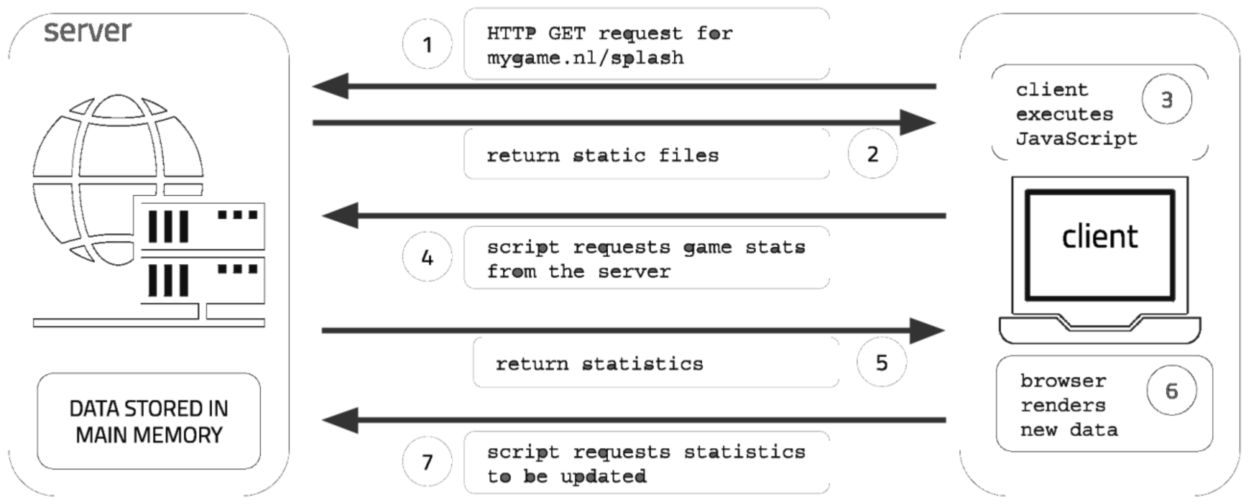
1. Develop the client-side code (HTML, JavaScript, CSS).
2. Place all client-side files into a directory (e.g. /static or /public) **on the server**.
3. Write the **server-side code** using Express.
4. Set Express static path to the directory created in 2
5. Add interactivity between client and server.

The client-side code of the web application resides within the /public folder. Within it, three subfolders exist: one for any data for the client, one for all JavaScript files and one for the stylesheets. Here, as we have only two HTML files (the splash screen and the game screen) we opted to not create a separate subfolder for them. All other .js files (i.e. anything not in /public) refer to server-side code.

Interactivity between client and server via Ajax

The client-server interaction can be implemented with **Ajax** - which is a sensible choice for many application settings. In the board game project the client-server interaction is largely based on WebSockets - which is a good choice due to the bidirectional communication needs of our app. Ajax *is* useful for the splash screen of our board game project, in particular when it comes to updating the game statistics (e.g. games played, games aborted, games ongoing).

The application flow showcases a possible client-server interaction to do just that:


server
DATA STORED IN MAIN MEMORY
1 HTTP GET request for mygame.nl/splash
2 return static files
3 client executes JavaScript
client
4 script requests game stats from the server
5 return statistics
6 browser renders new data
7 script requests statistics to be updated

The game statistics are stored in the server's main memory for simplicity; in any large-scale application they would be stored in a database that the server makes read/write requests to. Important to realize is that initially (step 2) the server only returns static files that do not contain the actual game statistics; in step 3 the client executes a few lines of JavaScript (Ajax!) to request the statistics from the server (step 4); the server sends in step 5 the requested data.

JSON: exchanging data between the client and server

JSON stands for JavaScript Object Notation and is a format that transmits data in human-readable text. It consists of attribute value pairs and array data types. Years ago, XML was used as data exchange format on the web. XML is well defined but not easy to handle. XML is often too bulky in practice; JSON has a much smaller footprint than XML. Importantly, JSON can be parsed with built-in JavaScript functionality (JSON.parse), which turns a JSON string into an object. JavaScript objects can be turned into JSON with the JSON.stringify method.

```
var gameStats = {
    gamesPlayed: 0,
    gamesAbolished: 0,
    gamesOngoing: 0,
    incrGamesPlayed : function(){
        this.gamesPlayed++;
    }
}
gameStats.incrGamesPlayed();

var jsonString = JSON.stringify(gameStats);
//"{\"gamesPlayed\":1,\"gamesAbolished\":0,\"gamesOngoing\":0}"

var jsonObject = JSON.parse(jsonString);
//jsonObject is equivalent to gameStats minus the properties that are functions
```

Two major differences between JSON and JavaScript objects are:

- In JSON, all property names must be enclosed in quotes.
- Objects created from JSON **do not have functions** as properties. If an object contains functions as properties, a call to JSON.stringify will strip them out.

With JSON being a de facto data exchange standard, Express has a dedicated response object method to send a JSON response: res.json(param). The method's parameter is converted to a JSON string using JSON.stringify()

Ajax

Ajax stands for **Asynchronous JavaScript and XML**. XML is in the name, and in the name only. XML is not commonly used as Ajax data exchange format anymore (JSON is!).

Asynchronous Javascript: In synchronous programming, things happen one at a time. When a function needs information from another function, it has to wait for it to finish and this delays the whole program. This is a bad use of your computer's resources, there's no point waiting for a process to finish especially in an era where computer processors are equipped with multiple cores. This is where asynchronous programming steps up. A function that takes too long to finish is separated from the main application and when it is done, it notifies the main program if it was successful or not.

Ajax is a **JavaScript mechanism** that enables the dynamic loading of content **without having to refetch/reload the page manually**. Ajax is a technology that **injects** new data into an existing web document. Ajax is not a language. Ajax is also not a product. Let's take the Bing search engine as an example: once you start typing a query, with each letter typed a new set of query suggestions will become available. If we keep the browser's web dev tools open (in particular the Network Monitor), we observe that each keystroke will lead to a new request/response message pair (and the request contains the currently typed query as visible on the Headers pane).

Selecting **XHR** shows only requests made via Ajax. XHR is short for XMLHttpRequest, which is an object offered by all major browsers that is at the heart of Ajax and allows us to:

1. make requests to the server without a full page reload;
2. receive data from the server.

The axios library hides a lot of the low-level XMLHttpRequest details and is one of the most popular choices for making Ajax calls. We will use it here. In this example code, you will note that the server-side code does not do anything special because Ajax is involved; **to the server, the requests look like any other HTTP request**. This leaves us to look at the client-side. Here, we first consider index.html

```
<!doctype html>
<head>
      <title>Available PlayStation VR Games</title>
</head>

<body>
      <main>
```

```
            <section class="available">
                  <p>Available PlayStation VR Games</p>
                  <ul class="available">
                  </ul>
            </section>
      </main>

      <script                 src="https://unpkg.com/axios/dist/axios.min.js"
type="text/javascript"></script>
      <script src="js/client-app.js"></script>
</body>
</html>
```

On the client-side, we load the axios library and then our application-specific JavaScript code. The domain https://unpkg.com/ is a large content delivery network. We could have also opted to store the current axios version locally, but then we would be responsible for keeping up with maintaining it. Note here, that the games list is **empty**. We have an empty unordered list element (<ul>) that will be filled with the available PlayStation VR games (which those are, we learn from the server) via an Ajax call. Let's find out how to make Ajax requests with the help of axios by looking at js/client-app.js

Let's start at the bottom of this code snippet. In order to retrieve the list of games from the server, we use axios.get(url).then(function(res)).catch(function(err)). This is axios' shorthand for making an HTTP GET request to url. If the request is successful the function specified within then() is executed, and if not, the function within catch() is executed. When the request is successful, res.data will contain the data retrieved from the server. What do we do with that data then? To answer this question, we need to take a look at our function addGamesToList: we first locate in the DOM tree the unordered list placeholder (and since we know there is only one, we can use "ul" as the input to document.querySelector()); for every game we received, we create a list element (<li>) with the corresponding text and append it to our <ul> element.

Ajax - without the high-level encapsulation through axios - works as follows:

1. The web browser creates a XMLHttpRequest object.
2. The XMLHttpRequest object requests data from a web server.
3. The data is sent back from the server.
4. On the client, JavaScript code injects the data into the page.

Importantly, with Ajax, the number of complete page reloads is vastly reduced. **Only the newly available** (e.g. a new set of query suggestions based on the current query) or **changed data** needs to be retrieved from the server, instead of the complete document. Ajax was and remains an important technology to move from web pages to web applications.

In practice, implementing Ajax calls correctly can be tricky, due to Ajax's security model. In our game list code example, we have conveniently requested data from *our* web server. In fact, a security restriction of Ajax (at least by default) is that it can only fetch files or request routes from the same web server as the calling page (this is called **same-origin policy**).

The same-origin policy is fulfilled when the **protocol, port and host** are the same for two pages. Important for debugging: Ajax **cannot** be executed from a web page opened locally from disk.

WebSockets

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. It works closely with the HTTP protocol but with lower overhead. Just need to create a handshake via a normal HTTP request but with header Upgrade: websocket.

We use the popular Node.js WebSocket library, which hides some of the low-level details (similar to axios hiding some of Ajax's low-level details).

Hello World! example for WebSockets: our client initiates a WebSocket connection with the server and sends a first message, the server uses the established connection to send a reply and then closes the connection. Let's look at the client-side:

```
<!DOCTYPE html>
<html>
    <head>
        <title>WebSocket test</title>
    </head>
    <body>
        <main>
            Status: <span id="hello"></span>
        </main>

        <!-- Poor coding standard, only for demonstration purposes.
             JavaScript code should not be part of HTML documents.
        -->
        <script>
            const target = document.getElementById("hello");
            const socket = new WebSocket("ws://localhost:3000");
            socket.onmessage = function(event){
                target.innerHTML = event.data;
            };

            socket.onopen = function(){
                socket.send("Hello from the client!");
                target.innerHTML = "Sending a first message to the server ...";
            };
        </script>
    </body>
</html>
```

We initiate a connection to a WebSocket server running on port 3000 on localhost with the line var socket = new WebSocket("ws://localhost:3000"). Due to the event-based nature of the WebSocket API, we write functions to be executed when particular events occur. In this case, for the open event, we send a message from client to server, while once we receive a message (event message), we grab the message from the event.data field and change the innerHTML property of our <span> element.

The corresponding server-side script looks as follows:

```
const express = require("express");
const http = require("http");
const websocket = require("ws");

const port = process.argv[2];
const app = express();

app.use("/", function (req, res) {
    res.sendFile("client/index.html", { root: "./" });
});

const server = http.createServer(app);

const wss = new websocket.Server({ server });

wss.on("connection", function (ws) {
    /*
     * let's slow down the server response time a bit to
     * make the change visible on the client side
     */
    setTimeout(function () {
        console.log("Connection state: " + ws.readyState);
        ws.send("Thanks for the message. --Your server.");
        ws.close();
        console.log("Connection state: " + ws.readyState);
    }, 2000);

    ws.on("message", function incoming(message) {
        console.log("[LOG] " + message);
    });
});

server.listen(port);
```

Here, we instantiate a WebSocket server object (wss) and define what is to happen in case of a connection event: as connections are always initiated by clients our server simply sends a reply message and closes the connection; we also add a callback for the message event that is defined on the WebSocket object (ws): whenever a message arrives, the message is logged to the console.

This code snippet also shows how we can gather additional information about the *state* of a WebSocket via the readyState read-only property:

readyState	**Description**
0	The connection is not yet open.
1	The connection is open; messages can be send/received.
2	The connection is currently being closed.
3	The connection is closed.

The WebSocket protocol as described in RFC 6455 has four event types:

- open: this event fires once a connection request has been made and the handshake was successful; messages can be exchanged now;
- message: this event fires when a message is received;
- error: something failed;
- close: this event fires when the connection closes; it also fires after an onerror event;

In our client-side code example we saw how simple it is to send data once a connection is established: socket.send().

The WebSocket Inspector can help you to debug your code!

WebSockets for multi-player games

In a multi-player game (such as the board games application implemented throughout the assignments), every player (client) establishes a WebSocket connection to the server. **The server has to keep track of which game each player is assigned to**. When a player in a game with multiple players sends a message to the server (e.g. to *broadcast* her latest move in the game), **the server has to send this message to all other players in the game** - and only to those players. Players active in other games should not receive those messages. *The communication via WebSockets is always between client and server. Two clients cannot communicate directly with each other, they have to go through the server.*

Thus, **the coordination effort lies with the server**: the client needs to maintain only a single WebSocket connection to the server; whenever the client receives a message, this message means an update to the game status (another player's move, another player dropped out of the game, the game has ended, etc.).

The main question then is, *how does the server-side keep track of games and players and their respective WebSocket connections?* One way is showcased in the demo game. Let's walk through the relevant pieces of app.js and game.js.

We keep track of which client is assigned to which game by mapping a WebSocket connection (the *property*) to a game (the *value*).

```
var websockets = {};//property: websocket, value: game
```

We here make use of object properties, but of course could also use JavaScript's Map object.

Our game data structure looks as follows

```
var game = function (gameID) {

    this.playerA = null;

    this.playerB = null;

    this.id = gameID;

    this.wordToGuess = null;

    this.gameState = "0 JOINT";

};
```

The demo game is a word guesser game, so our game data structure holds the WebSocket connections of the two players in a game as well as a numeric game identifier, the word to guess (provided by whoever joined the game first) and the current game state (a set of states we defined based on the game mechanics).

When a client establishes a new WebSocket connection with the server, the server-side script has several tasks:

1. determine whether a new game should be started or whether the current game still requires additional players (and thus our newly connected player should join that game);
2. inform the player about the current game status;
3. request information from the player if necessary (e.g. in the word guessing game, the first player who joins a game is asked for the word to guess).

The relevant code snippet looks as follows:

```
var currentGame = new Game(gameStatus.gamesInitialized++);
var connectionID = 0;//each websocket receives a unique ID

wss.on("connection", function connection(ws) {

    /*
     * two-player game: every two players are added to the same game
     */
    let con = ws;
    con.id = connectionID++;
    let playerType = currentGame.addPlayer(con);
    websockets[con.id] = currentGame;
    /*
     * inform the client about its assigned player type
     */
    con.send((playerType == "A") ? messages.S_PLAYER_A : messages.S_PLAYER_B);

    ...
}
```

We assign every WebSocket connection object a **unique identifier**, add the player to the game currently missing a player and then inform the player about the player type (word guesser or word provider).

Another interesting aspect to mention is the choice of messages to pass back and forth: of course, what messages to pass depends entirely on the game to implement. As a concrete example, in the word guesser game, we have a number of messages which are defined in messages.js:

- GAME-WON-BY
- GAME-ABORTED
- CHOOSE-WORD
- PLAYER-TYPE
- SET-TARGET-WORD
- MAKE-A-GUESS
- GAME-OVER

If you look at the path of messages.js you will find that this JavaScript file is part of the **client-side JavaScript code**. This makes sense, as both client and server need to be able to interpret the messages, so ideally we only create the message types once.

These two lines of code

```
(function(exports){

    ...

}(typeof exports === "undefined" ? this.Messages = {} : exports));
```

Enable us to **share** JavaScript code between our server-side and client-side JavaScript runtime. In our server-side app.js file, we can import this piece of code as usual via

```
const messages = require("./public/javascripts/messages");
```

Self-check

True or False? Node's event loop is single-threaded. In addition, it maintains a pool of threads in order to process I/O requests.

- True.

True or False? Node's event loop is multi-threaded. In addition, it maintains a single thread in order to process I/O requests.

- False.

True or False? In contrast to Ajax, WebSockets do not require entire HTTP messages to be sent between client and server (thus reducing the overhead).

- True.

True or False? WebSockets require the use of the XMLHttpRequest object to request a protocol upgrade.

- False.

True or False? Ajax requires the use of XML as data exchange format.

- False.

True or False? Client and server agree to the WebSocket protocol as follows: the client initiates the protocol upgrade by sending a HTTP request with at least two headers: `Connection:Upgrade` and `Upgrade:websocket`.

- True.

True or False? The WebSocket protocol relies on long polling to enable bidirectional communication between client and server.

- False.

```
var fs = require('fs');

var n = -1;


function f(){

  //asynchronous

  fs.readFile('n.txt', function(err, content){

    n = parseInt(content); //n.txt contains 42

    n++;

  });

}
```

```
f();

console.log(n);
```

Executing the above Node.js code yields which console output?

- -1

```
var fs = require('fs');


function f(done){

   fs.readFile('n.txt', function(err, content){

      var n = parseInt(content); //n.txt contains 42

      n++;

      done(n);

   })

}


f(function(n){

   console.log(n);

});
```

Executing the above Node.js code yields which console output?

- 43

Imagine building a chat application using Ajax (under HTTP/1.1). How is the browser notified of new messages to display?

- The browser polls the server for message updates in short time intervals.

Lecture 5. CSS: the language of web design

Required reading – What is CSS

Browser will style HTML documes using an internal stylesheet that ensures that headings are larger than normal text, links are highlighted, lists are indented, paragraphs are spaced out, etc.

CSS is a language for specifying how documents are presented to user agents (a program that represents a person i.e. browsers) — how they are styled, laid out, etc. Browsers are the main type of user agent we think of when talking about CSS, however, it is not the only one. There are other user agents available — such as those which convert HTML and CSS documents into PDFs to be printed.

CSS is a rule-based language — you define rules specifying groups of styles that should be applied to particular elements or groups of elements on your web page.

The rule opens with a **selector** . This selects the **HTML element that we are going to style**. In this case we are styling level one headings (<h1>). You can target multiple at once by using commas.

Selector

```
h1, h2 {
    color: red;
    font-size: 5em;
}
```

We then have a set of curly braces { }. Inside those will be one or more **declarations**, which take the form of **property** and **value** pairs.

https://developer.mozilla.org/en-US/docs/Web/CSS/Reference contains all CSS properties, which are grouped in modules.

Required reading – Getting started wtih CSS

The very first thing we need to do is to tell the HTML document that we have some CSS rules we want it to use. There are three different ways to apply CSS to an HTML document:

- Create a file in the same folder as your HTML document and save it as styles.css. The .css extension shows that this is a CSS file.
- To link styles.css to index.html add the following line somewhere inside the <head> of the HTML document. Both of these are called “External stylesheets”.

```
<link rel="stylesheet" href="styles.css">
```

- Have the CSS content within <style></style> tags right after opening <head> (Internal stylesheet)

Styling things based on their Class

So far we have styled elements based on their HTML element names. This works as long as you want all of the elements of that type in your document to look the same. You can add a class attribute in the html element tag to target that specific class. **The CSS class selector has a preceding dot i.e. .class**{ color:red;}

You can combine an element and a class. Instead of styling all the classes or all the elements you could:

Combinator:

```
li.special {
  color: orange;
  font-weight: bold;
}
```

style only the lists of ‘special’ class.

Styling things based on their location in a document

To select only an <em> that is nested inside an <li> element I can use a selector called the **descendant combinator**, which takes the form of a space between two other selectors. HTML reference:

```
<li>Item <em>three</em></li>
```

Add the following rule to your stylesheet.

```
li em {

  color: rebeccapurple;

}
```

Result: Item *three*

Styling a paragraph (only) when it comes directly after a heading at the same hierarchy level in the HTML. To do so place a + (an **adjacent sibling combinator**) between the selectors.

```
<body>

    <h1>I am a level one heading</h1>

    <p>This is tyled.</p>

    <p>This is is not styled</p>
```

```
h1 + p {

  font-size: 200%;

}
```

Styling things based on state

When we style a link we need to target the <a> (anchor) element. This has different states depending on whether it is unvisited, visited, being hovered over, focused via the keyboard, or in the process of being clicked (activated). You can use CSS to target these different states. (declared afer ‘:’)

```
a:hover {
  color: pink;
}
```

```
a:visited {
  color: green;
}
```

Combining selectors and combinators

You can combine multiple types together i e. body h1 + p .special

Required reading – How CSS is structured

In some circumstances, internal stylesheets can be useful. For example, perhaps you're working with a content management system where you are blocked from modifying external CSS files. But for sites with more than one page, an internal stylesheet becomes a less efficient way of working.

Inline styles are CSS declarations that affect a single HTML element, contained within a style attribute.

```
<h1 style="color:yellow;border: 1px solid black;">Hello World!</h1>
```

Avoid using CSS in this way, when possible. It is the opposite of a best practice. It is hard to mantain.

Important: CSS properties and values are case-sensitive.

Selectors

* // everything

h1 // all h1 headers

a:link //all default links (non-visited)

.manythings //all things belonging to manythings class

#onething //points to the id attribute, which is unique in the entire html document

Specifity

The most specific selector (id, then class, then tag element) gets implemented in case of conflict, regardless of the order in wich the styling rules were specified.

Cascade

In the event of 2 identical declarations, the last one that takes place is the one implemented.

Functions

CSS can do simple things such as:

```
width: calc(90% - 30px);

transform: rotate(0.8turn);
```

@rules

(pronounced "at-rules") provide instruction for what CSS should perform or how it should behave.

```
@import 'styles2.css';

@media only screen and (max-width: 600px), (max-height: 700px)  {  body {

    background-color: blue;

  }
```

One common @rule that you are likely to encounter is @media, which is used to create media queries. Media queries use conditional logic for applying CSS styling.

Shorthands

Some properties like font, background, padding, border, and margin:

```
padding: 10px 15px 15px 5px;
```

=

```
padding-top: 10px;
padding-right: 15px;
padding-bottom: 15px;
padding-left: 5px;
```

(clockwise starting from 12 o'clock)

Comments

Highly recommended to use, CSS comments begin with /* and end with */

Lecture

There are more than 140 color names that all modern browsers recognize (a nicely formatted list is available on Wikipedia). Besides hex values, colors can also be represented by their rgba values (red, green, blue, alpha). convertingcolors.com as one option to convert between different color formats.

2 types of style sheets: the **browser's style sheets** (also called the *internal stylesheets* or *default styles*) and the **web application style sheets**, the latter overriding the former. There is a third type of style sheets and that are the **user style sheets**: user style sheets allow users to override the web applications' look and feel according to their own needs (e.g. larger font sizes or specific colors).

Pseudo-classes

A **pseudo-class** is a keyword added to a **selector** that indicates *a particular state or type* of the corresponding element. Pseudo-classes allow styling according to, among others, **document external** factors such as mouse movements and user browsing history as well as **element external** factors such as the placement of the element within the document structure.

Pseudo-classes are added to a selector with a colon (:) separating them:

```
selector:pseudo-class {
    property: value;
    property: value;
}
```

nth-child(x) and nth-of-type(x)

Imagine you want to show off a number of game statistics on the splash screen of your web application. To increase readability, the background and font colors of the items should alternate. There are various ways to do this, here are two simple ones:

- We can *hardcode* the CSS rule of every element, ending up with as many rules as we have game statistics to show off. This is not maintainable.
- We write two CSS rules (one per color choice), assign each to a class (e.g. .odd and .even) and then alternate the class assignment of the elements holding the game statistics. Better than the

first option, but still not ideal. What happens if we want to include a game statistic in the middle of the existing list? All subsequent game statistics would have to be assigned a different class to maintain the alternating colors styles.

Ideally, we only create two CSS rules and solve the rest (alternate assignment of colors) with pseudo-classes. And that's exactly what we can do with the two pseudo-classes we introduce now:

- :nth-child(X) is any element that is the Xth **child element** of its parent;
- :nth-of-type(X) is any element that is the Xth **sibling** of its type.

In both cases, X can be an integer or formula, e.g 2n+1, where n represents a number starting at 0 and incrementing. Instead of 2n and 2n+1 we can also use even and odd as shortcuts. If we want to start counting elements in reverse order, we can use the analogous :nth-last-child(X) and :nth-last-of-type(X) pseudo-classes.

If we are aiming at the first and/or last child or sibling element, we also have additional pseudo-classes available to us:

Pseudo-class	**Equivalent to**
:first-child	:nth-child(1)
:last-child	:nth-last-child(1)
:first-of-type	:nth-of-type(1)
:last-of-type	:nth-last-of-type(1)

Variables

```
:root {
  --dark: rgb(37, 40, 58);
}
button:disabled {
  color: var(--dark);
}
```

z

In this example, we create a global CSS variable, i.e. one that is available to all elements in the DOM tree. For this reason, we make use of the pseudo-element :root which represents the <html> element. Variables are defined with the custom prefix -- and can be accessed using the var() function. Non-global CSS variables can be added in the same manner to any element, though they are then only available within their {....} block.

:hover, :active and :visited

- :hover is a selector that becomes active when a mouseover on the element occurs.
- :active is a selector that becomes active when the element is currently being active (e.g. while it is being clicked).
- :visited is a selector that selects links a user has already visited. It is thus more specific than :hover and :active which apply to any type of element, not just links.

:enabled and :disabled

- :enabled is an element that can be clicked or selected.
- :disabled is an element that cannot be clicked or selected.

These two pseudo-classes are not available for all elements (as not all elements are clickable), but they are available, among others, for <button>, <input> and <textarea>

Selector combinations

Selector combination	Description
e1	Selects all <e1> elements
e1 e2	Selects all <e2> elements within <e1>
e1,e2	Selects all <e1> elements and all <e2> elements
e1>e2	Selects all <e2> elements that have <e1> as parent
e1+e2	Selects all <e2> elements that follow <e1> immediately

Pseudo-elements

They provide access to an element's **sub-parts** such as the first letter of a paragraph. In order to distinguish pseudo-classes and pseudo-elements, the double-colon (::) notation was introduced in the specification, though browsers in practice also accept the colon (:) notation.

Two popular examples are the ::first-letter and the ::first-line

Without those pseudo-elements, you would have to wrap the first letter in a <span> (or similar) element.

::before and ::after

Adding (cosmetic) content right before and after an element is achieved through ::before and ::after respectively in combination with the content property

```
a::after {
  content: " (hyperlink)";
}
```

position: absolute

positions an element **relative to its closest positioned ancestor**.

Data attributes

Attributes on any HTML element that are prefixed with data-

CSS can access those data attributes with the attr() function: it retrieves the value of the selected element and data attribute.

```
button::after {
      content: "as of " attr(data-date);
}
```

Only attributes that start with the prefix data- can be surfaced in CSS via the attr() function.

```
<button id="b1" data-date="01/10/2020">12,567 players registered</button>
```

Element positioning

- The property display defines the display type of an element.
- Grids consist of rows and columns and elements can be placed onto them.
- The property position defines how an element is positioned in a document.

when we create HTML documents without any styling, the elements do not appear randomly on the screen, instead they are ordered in some fashion. By default, elements *flow*. **Their order is determined by the order of their appearance in the HTML document.**

By default certain elements are surrounded by line-breaks and others are not. These two types of elements are known as block-level and inline elements respectively.

Block-level elements are **surrounded by line-breaks**. They can contain block-level and inline elements. **Their width is determined by their containing element**

Examples of block-level **elements are <main>, <h1> or <p>**.

Inline elements can be placed within block-level or inline elements. They can contain other inline elements. **Their width is determined by their content.** Examples are **<span>** or **<a>**

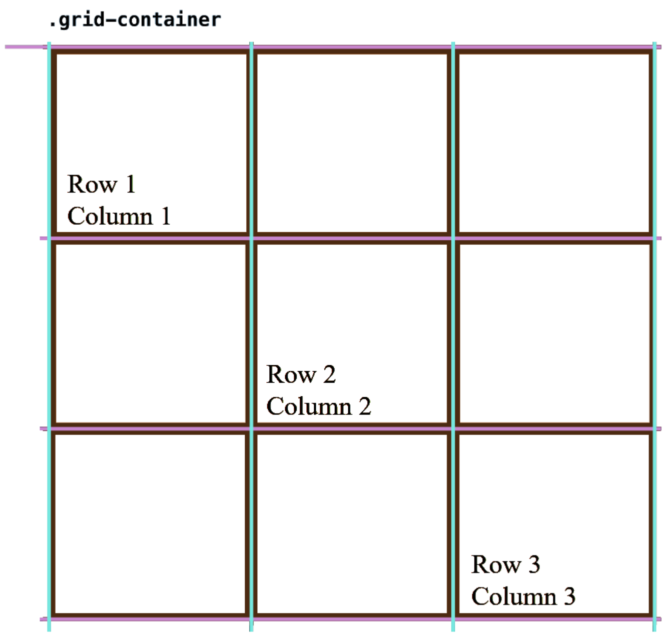
Display Types

The display property enables us to **change the element type (e.g. from block-level to inline),** hide elements from view and determine the layout of its children.

Value	**Description**
display:inline	The element is treat as an inline element.
display:block	The element is treated as a block element (line breaks before and after the element).
display:none	The element (and its descendants) are hidden from view; no space is reserved in the layout.
display:grid	The children of this element are arranged according to the grid layout.

grid Layout

As the name suggests, a display type of grid allows us to create a two-dimensional grid layout within a page. A CSS grid is comprised of rows and columns. Each element that we define within the grid can be likened to as a cell.



```html
<div class="grid-container">
    <div>Row 1, Column 1</div>
    <div>Row 1, Column 2</div>
    <div>Row 1, Column 3</div>
    <div>Row 2, Column 1</div>
    <div>Row 2, Column 2</div>
    <div>Row 2, Column 3</div>
    <div>Row 3, Column 1</div>
    <div>Row 3, Column 2</div>
    <div>Row 3, Column 3</div>
</div>
```

```css
.grid-container {
    display: grid;
    grid-template-rows: 100px 100px 100px;
    grid-template-columns: 100px 100px 100px;
}
```

=

```css
.grid-container {
    display: grid;
    grid-template-rows: repeat(3, 100px);
    grid-template-columns: repeat(3, 100px);
}
```

We can use grid-gap (or grid-row-gap and grid-column-gap) to add spaces between cells.

Position

Value	Description
position:static	the default
position:relative	the element is adjusted on the fly, other elements are **not** affected
position:absolute	the element is taken out of the normal flow (**no space is reserved for it**)
position:fixed	similar to absolute, but fixed to the **viewport** (i.e. the area currently being viewed)
position:sticky	in-between relative and fixed (*we do not consider it further in this class*)

position:absolute

Important to know when using the position property is the direction of the CSS coordinate system: the top-left corner is (0,0). The y-axis extends **downwards**. The x-axis extends to the **right**.

An element with property position:absolute is taken **out of the element flow**. The original space reserved for this element is no longer reserved. Its position is determined relative to its nearest ancestor element that has the absolute position property set as well. If this property is not set for any ancestor, the viewport is considered as ancestor. The horizontal and vertical offset properties have the following effects:

- left: distance to the left edge of the containing block;
- right: distance to the right edge of the containing block;
- bottom: distance between the element's bottom edge and the bottom edge of the containing block;
- top: distance between the element's top edge and the top edge of the containing block.

position:relative

Adding the property/value pair position:relative to an element sets up the element to be moved **relative to its original position**. The original space reserved for this element remains reserved. The horizontal offset from the original position is set through properties left and right, the vertical offset is controlled through top and bottom:

- left moves the element **to the right** of its original position by the given length;
- right moves the element **to the left** of its original position;
- top moves the element **downward** of its original position;
- bottom moves the element **upwards** of its original position.

position:fixed

The position:fixed setting is similar to position:absolute, but now the containing element is **always** the **viewport**. This means that elements with position:fixed always remain visible.

CSS media queries

```
@media (max-width: 500px){
```

CSS media queries enable the use of **device/media-type dependent** style sheets and rules. While the HTML document is written once, the CSS is written once per device type.

Value	**Description**
media all	Suitable for all device types. The default if no media type is provided.
media print	Suitable for documents in print preview.
media screen	Suitable for screens.
media speech	Suitable for speech synthesizers.

In the code example below we link the file large-device.css when at least one of two conditions holds:

- the media type is screen **and** the width is at least 800px but no more than 2000px;
- the width of the device is at least 3000px independent of the device type.

```
<!DOCTYPE html>
<html>
  <head>
    <!-- Example of a logical or (,) in the media query -->
    <link rel="stylesheet" media="screen and (min-width: 800px) and (max-
width: 2000px),
      (min-width: 3000px)" href="large-device.css">

    <style>
      /* more rules can be defined here */
    </style>
  </head>
  <body>
  </body>
</html>
```

CSS animations and transitions

In general, CSS styles (states) are defined by the developer. As concrete examples, in our CSS weather demo, we define:

- how a rain drop looks, its starting point, end point and how quickly it should “fall”;
- the path the leaves travel and their speed;
- the opacity levels of a <div> to simulate flash lightning;
- etc.

The **rendering engine** then takes care of the **transition between styles**. A rendering engine - also known as *browser engine* or *layout engine* - is responsible for translating HTML+CSS (among others) to the screen. The major browsers ship with their own rendering engines, the names of which you will encounter from time to time, especially when using CSS animations and transitions:

Engine	**Browsers**
Gecko	Firefox
Trident	Internet Explorer
EdgeHTML	Microsoft Edge
WebKit	Safari, older versions of Google Chrome
Blink	Google Chrome, Opera

Animations consist of:

- an animation style (e.g. linear, 3 seconds duration, 10 times);
- a number of **keyframes** that act as transition waypoints.

Transitions are animations with a simpler syntax. They consist of exactly **two states**: start and end state.

There are several advantages to using CSS-based instead of JavaScript-based animations

- CSS is relatively easy to use; debugging CSS-based animations is easier than debugging JavaScript code.
- The rendering engines are optimized for CSS-based animations; when implementing an animation from scratch in JavaScript code would have to be optimized to reach the desired frame rate.

Text-heavy animations

- The animation property is a short-hand property that combines a number of animation-* properties in one line. Specifically in our example animation: flicker 3s linear infinite refers to the keyframes (flicker - the animation-name), the duration of one animation cycle (3 seconds - the animation-duration), the animation-timing-function (linear means that the animation moves from start to end in a constant rate) and the animation-iteration-count (here: infinite, i.e. the animation never stops). We defined here two types of flickers: a slow flicker (3 seconds to complete a cycle) and a fast flicker (1 second to complete the cycle). Different letters of our TU Delft string are assigned to different *flicker classes*.
- The @keyframes control the intermediate steps in a CSS animation. In order to achieve flickering we change the opacity of the text string. In one animation cycle, we repeatedly move from an opacity of .99 to .3 and back. Specifically, we define 8 waypoints of our animation (with either opacity of .99 or .3): 0%, 30%, 31%, 33%, 50%, 55%, 56%, 100%. The rendering engine is then responsible to turn this code into an actual animation that resembles flickering.

- By default, all elements with a particular animations are animated *at the same time*. That is often undesired as the animations should look somewhat random (we do not want all letters to flickr at exactly the same time, we do not want all cards to perform a rotation at exactly the same time). The animation-delay property allows us to define the amount of time to wait before starting the animation. If that delay differs for most elements (as it does in our splash screen), we achieve the desired randomized effect.

Property	**Description**
animation-iteration-count	Number of times an animation is executed (default: 1); the value is either a positive number or infinite.
animation-direction	By default the animation restarts at the starting keyframe; if set to alternate the animation direction changes every iteration.
animation-delay	Number of seconds (s) or milliseconds (ms) until the animation starts (default 0s). A **negative** value (e.g. x=-5s) means that the animation starts immediately but already x seconds into the animation cycle
animation-name	Identifies the @keyframes.
animation-duration	The duration of a single animation cycle in seconds (s) or milliseconds (ms), e.g. 2.5s, 500ms.
animation-timing-function	Specifies how a CSS animation progresses between waypoints (common choices are linear, ease-in, ease-out, steps(N)).

Example:

```
/* animation */
#joingame{
    animation: blink 4s linear infinite;
}

@keyframes blink {
    50% {
      background-color: var(--medium); /*does half the blink then repeats*/
    }
  }
```

Lecture 6 - Advanced Node.js

Organization and reusability of Node.js code

A Node.js module is (1) a single file or (2) a directory of files and all code contained in it. By default, *no code in a module is accessible to other modules*. Any property or method that should be visible to other modules has to be **explicitly** marked as such. Modules often depend on a number of other modules.

To install modules, use the command line e.g. “npm install Winston” installs one of the more popular Node logging libraries. You can also use the command line to search for modules, "npm search Winston”.

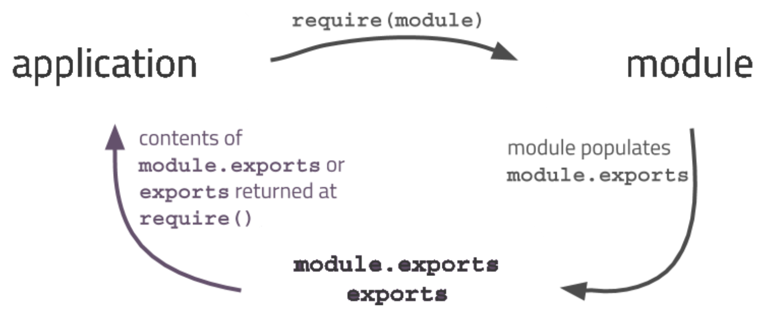
A file-based module system

In Node.js each file is its own module. This means that the code we write in a file does not pollute the global namespace (In Node.js we get this setup “for free”, in the client-side we do pollute the global namespace, module pattern constructors may avoid this).

The module system works as follows: each Node.js file can access its so-called **module definition** through the module object. The module object is your entry point to modularize your code. To make something available from a module to the outside world, module.exports or exports is used.

```
Module {
  id: '.',
  path: '/Users/Node/Web-Teaching',
  exports: {},
  parent: null,
  filename: '/Users/Node/Web-Teaching/myfile.js',
  loaded: false,
  children: [],
  paths: [
    '/Users/Node/GitHub/Web-Teaching/node_modules',
    '/Users/Node/GitHub/node_modules',
    '/Users/Node/node_modules',
    '/Users/node_modules',
    '/node_modules'
  ]
}
```

To execute a node js file write in the terminal “node file.js”


require(module)
application
module
contents of module.exports or exports returned at require()
module populates module.exports
module.exports
exports

We see that in our module object above nothing is currently being *exported* as the exports property has an empty object ({}) as value.

Once you have defined your own module, the globally available require function is used to import a module.

An application uses the require function to import module code. The module itself populates the module.exports variable to make certain parts of the code base in the module available to the outside world. Whatever was assigned to module.exports is then returned to the application when the application calls require().

Let's consider files foo.js:

```
var fooA = 1;                       //LINE 1
module.exports = "Hello!";          //LINE 2
module.exports = function() {       //LINE 3
    console.log("Hi from foo!");    //LINE 4
};                                  //LINE 5
```

and bar.js:

```
var foo = require("./foo");
foo();                        //CASE 1: Hi from foo!
require("./foo")();           //CASE 2: Hi from foo!
console.log(foo);             //CASE 3: [Function]
console.log(foo.toString());//CASE 4: function () {console.log("Hi from
foo!");}
console.log(fooA);            //CASE 5: ReferenceError (you will have to remove
this line to reach the next one)
console.log(module.exports);//CASE 6: {}
```

In bar.js the first line calls the require() function and assigns the returned value to the variable foo. In line 1 we used as argument ./foo instead of ./foo.js - you can use both variants.

Node.js runs the referenced JavaScript file (here: foo.js) in a **new scope** and **returns the final value** of module.exports. What then is the final value after executing foo.js? It is the function we defined in line 3.

As you can see in lines 2 and beyond of bar.js there are several ways to access whatever require returned:

- **CASE 1** We can call the returned function and this results in *Hi from foo!* as you would expect.
- **CASE 2** We can also combine lines 1 and 2 into a single line with the same result.
- **CASE 3** If we print out the variable foo, we learn that it is a function.
- **CASE 4** Using the toString() function prints out the content of the function.
- **CASE 5** Next, we try to access fooA - a variable defined in foo.js. Remember that Node.js runs each file in a new scope and only what is assigned to module.exports is available. Accordingly, fooA is not available in bar.js and we end up with a reference error. 🐛Visual Studio Code flags up this error (fooA is not defined) already at the code writing stage.
- **CASE 6** Finally, we can also look at the module.exports variable of bar.js - this is always available to a file in Node.js. In bar.js we have not assigned anything to module.exports and thus it is an empty object.

require is blocking

This module setup also explains why require is **blocking**: once a call to require() is made, the referenced file's code is executed and only once that is done, does require() return. This is in contrast to the usual *asynchronous* nature of Node.js functions.

In script with several require calls to the same file. The first time the line require(foo.js) is executed, the file foo.js is read from disk (this takes some time). In subsequent calls to require(foo.js), however, the **in-memory object is returned**. Thus, **module.exports is cached**.

module.exports vs. exports

Every Node.js file has access to module.exports. If a file does not assign anything to it, it will be an empty object, but it is **always** present. Instead of module.exports we can use exports as exports is an **alias** of module.exports. This means that the following two code snippets are equivalent:

SNIPPET 1

```
module.exports.foo = function () {
    console.log('foo called');
};

module.exports.bar = function () {
    console.log('bar called');
};
```

SNIPPET 2

```
exports.foo = function () {
    console.log('foo called');
};

exports.bar = function () {
    console.log('bar called');
};
```

In the first snippet, we use module.exports to make two functions (foo and bar) accessible to the outside world. In the second snippet, we use exports to do exactly the same. Note that in these two examples, **we do not assign something to exports directly**, i.e. we do not write exports = function ..... This is in fact **not possible** as exports is only a reference (a short hand if you will) to module.exports: if you directly assign a function or object to exports, then its reference to module.exports will be **broken**. You can only **assign directly** to module.exports, for instance, if you only want to make a single function accessible.

Creating and using a (useful) module

Modules can be either:

- a **single file**, or,
- a **directory of files**, one of which is index.js.

A module can contain other modules (that's what require is for) and should have a specific purpose. For instance, we can create a *grade rounding module* whose functionality is the rounding of grades in the Dutch grading system. Any argument that is not a number between 1 and 10 is rejected:

```
/* not exposed */
var errorString = "Grades must be a number between 1 and 10.";

function roundGradeUp(grade) {
  if (isValidNumber(grade) == false) {
    throw errorString;
  }
  return ( Math.ceil(grade) > 10 ? 10 : Math.ceil(grade));//max. is always 10
}

function isValidNumber(grade) {
  if (
    isNaN(grade) == true ||
    grade < exports.minGrade ||
    grade > exports.maxGrade
  ) {
    return false;
  }
  return true;
}

/* exposed */
exports.maxGrade = 10;
exports.minGrade = 1;
exports.roundGradeUp = roundGradeUp;
exports.roundGradeDown = function(grade) {
  if (isValidNumber(grade) == false) {
    throw errorString;
  }
  return Math.floor(grade);
};
```

We can use the grading module in an Express application as follows:

```
var express = require("express");
var url = require("url");
var http = require("http");
var grading = require("./grades"); // our module file resides in the current
directory
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port);
```

```
app.get("/round", function(req, res) {
  var query = url.parse(req.url, true).query;
  var grade = query["grade"] != undefined ? query["grade"] : "0";

  //accessing module functions
  res.send(
    "UP: " +
      grading.roundGradeUp(grade) +
      ", DOWN: " +
      grading.roundGradeDown(grade)
  );
});
```

Assuming the Node script is started on localhost and port 3000, we can then test our application with several valid and invalid queries:

- http://localhost:3000/round?grade=2.1a
- http://localhost:3000/round?grade=2.1
- http://localhost:3000/round?grade=
- http://localhost:3000/round?grade=10

Middleware in Express

Middleware components are small, self-contained and reusable code pieces across applications.

Middleware components have **three parameters**:

- an HTTP request object;
- an HTTP response object;
- an optional callback function—next()—to indicate that the component is finished; the **dispatcher** which orchestrates the order of middleware components can now move on to the next component.

Middleware components have a number of abilities. They can:

- execute code;
- change the request and response objects;
- end the request-response cycle;
- call the next middleware function in the middleware stack.

Logger example

Our goal is to create a logger that records every single HTTP request made to our application as well as the URL of the request. We need to write a function that accepts the HTTP request and response objects as arguments and next as callback function.

```
var express = require("express");

//a middleware logger component
function logger(request, response, next) {
  console.log("%s\t%s\t%s", new Date(), request.method, request.url);
  next(); //control shifts to next middleware function
}

//a middleware delimiter component
function delimiter(request, response, next) {
  console.log("-----------------");
  next();
}

var app = express();
app.use(logger); //register middleware component
app.use(delimiter);
app.listen(3001);
```

If you start the script and then make the following HTTP requests:

- http://localhost:3001/first
- http://localhost:3001/second
- http://localhost:3001/

your output on the terminal will look something like this:

```
2020-01-14T19:17:16.577Z        GET     /first
-----------------
2020-01-14T19:17:19.111Z        GET     /second
-----------------
2020-01-14T19:17:21.159Z        GET     /
-----------------
```

Importantly, next() enables us to move on to the next middleware component while **app.use(...) registers the middleware component with the dispatcher**.

Up to this point, none of our routes had contained next for a simple reason: all our routes ended with an HTTP response being sent and this completes the request-response cycle; in this case there is no need for a next() call.

Authorization component example

In the node-component-ex example application, we add an authorization component to a simple wishlist application back-end: only clients with the **correct username and password** (i.e. authorized users) should be able to receive their wishlist when requesting it. We achieve this by adding a middleware component that is activated for every single HTTP request and determines:

- whether the HTTP request contains an authorization header (if not, access is denied);
- and whether the provided username and password combination is the correct one.

Before we dive into the code details, install and start the server as explained here. Take a look at app.js before proceeding.

Once the server is started, open another terminal and use curl, a command line tool that provides us with a convenient way to include username and password as you will see:

- Request the list of wishes without authorization, i.e. curl http://localhost:3000/wishlist - you should see an Unauthorized access error.
- Request the list of wishes with the correct username and password (as hard-coded in our demonstration code): curl --user user:password http://localhost:3000/wishlist. The option --user allows us to specify the username and password to use for authentication in the [USER]:[PASSWORD] format. This request should work.
- Request the list of wishes with an incorrect username/password combination: curl --user test:test http://localhost:3000/wishlist. You should receive a Wrong username/password combination error.
- Add a wish to the wishlist:

  curl --user user:password "http://localhost:3000/addWish?type=board%20game&name=Treasure%20Hunt&priority=low" (note that whitespaces are replaced by %20 in URLs and the URL has to appear within quotes due to the special characters & in it). Did it work? If so, another execution of curl --user user:password http://localhost:3000/wishlist should result in a wishlist with now three wishes.

Having found out how the code *behaves*, let us look at the authorization component. We here define it as an anynymous function as argument to app.use:

```
app.use(function (req, res, next) {
    var auth = req.headers.authorization;
    if (!auth) {
        return next(new Error("Unauthorized access!"));
    }

    //extract username and password
    var parts = auth.split(' ');
    var buf = new Buffer(parts[1], 'base64');
    var login = buf.toString().split(':');
    var user = login[0];
    var password = login[1];

    //compare to 'correct' username/password combination
    //hardcoded for demonstration purposes
    if (user === "user" && password === "password") {
        next();
    }
    else {
        return next(new Error("Wrong username/password combination!"));
    }
});
```

This code snippet first determines whether an authorization header was included in the HTTP request (accessible at req.headers.authorization). If no header was sent, we pass an error to the next() function, for Express to catch and process, i.e. sending the appropriate HTTP response. If an authorization header is present, we now extract the username and password (it is base64 encoded!) and determine whether they match user and password respectively. If they match, next() is called and the next middleware component processes the request, which in our app.js file is app.get("/wishlist",...).

Routing

Routing is the mechanism by which requests are routed to the code that handles them. The routes are specified by a URL and HTTP method (most often GET or POST). You have employed routes already - every time you wrote app.get() you specified a so-called **route handler** for HTTP method GET and wrote code that should be executed when that route (or URL) is called.

This routing paradigm is a significant departure from the past, where **file-based** routing was commonly employed. In file-based routing, we access files on the server by their actual name, e.g. if you have a web application with your contact details, you typically would write those details in a file contact.html and a client would access that information through a URL that ends in contact.html. Modern web applications are not based on file-based routing, as is evident by the fact URLs these days do not contain file endings (such as .html or .asp) anymore.

In terms of routes, we distinguish between request **types** (GET /user differs from POST /user) and request **routes** (GET /user differs from GET /users).

Route handlers are middleware. So far, we have not introduced routes that include next as third argument, but since they *are* middleware, we can indeed add next as third argument.

A/B Testing example

```
//clients request their wishlists
app.get("/wishlist", function (req, res, next) {
    //hardcoded “A-B” testing
    if (Math.random() < 0.5) {
        return next();
    }
    console.log("Wishlist in schema A returned");
    res.json(wishlist_A);
});

app.get("/wishlist", function (req, res,next) {
    console.log("Wishlist in schema B returned");
    res.json(wishlist_B);
});
```

We define two route handlers for the same route /wishlist. Both anonymous functions passed as arguments to app.get() include the next argument. The first route handler generates a random number between 0 and 1 and if that generated number is below 0.5, it calls next() in the return statement. If the generated number is >=0.5, next() is not called, and instead a response is sent to the client making the request. If next was used, the dispatcher will move on to the second route handler and here, we do not call next, but instead send a response to the client. What we have done here is to hardcode so-called *A/B testing*. Imagine you have an application and two data schemas and you aim to learn which schema your users prefer. Half of the clients making requests will receive schema A and half will receive schema B.

We can also provide multiple handlers in a single app.get() call:

```
//A-B-C testing
app.get('/wishlist',
    function(req,res, next){
        if (Math.random() < 0.33) {
            return next();
        }
        console.log("Wishlist in schema A returned");
        res.json(wishlist_A);
    },
    function(req,res, next){
        if (Math.random() < 0.5) {
            return next();
        }
        console.log("Wishlist in schema B returned");
        res.json(wishlist_B);
    },
    function(req,res){
        console.log("Wishlist in schema C returned");
        res.json(wishlist_C);
    }
);
```

This code snippet contains three handlers - and each handler will be used for about one third of all clients requesting /wishlist. While this may not seem particularly useful at first, it allows you to create generic functions that can be used in any of your routes, by dropping them into the list of functions passed into app.get(). What is important to understand *when* to call next and *why* in this setting we have to use a return statement - without it, the function's code would be continued to be executed.

Routing paths and string patterns

When we specify a path (like /wishlist) in a route, the path is eventually converted into a **regular expression** (short: regex) by Express. Regular expressions are patterns to match character combinations in strings. They are very powerful and allow us to specify **matching patterns** instead of hard-coding all potential routes. For example, we may want to allow users to access wishlists via a number of similar looking routes (such as /wishlist, /whishlist, /wishlists). Instead of duplicating code three times for three routes, we can employ a regular expression to capture all of those similarly looking routes in one expression.

Express distinguishes three different types of route paths:

- strings;
- string patterns; and
- regular expressions.

So far, we have employed just strings to set route paths. **String patterns** are routes defined with strings and a subset of the standard regex meta-characters, namely: + ? * ( ) []. Regular expressions contain the full range of common regex functionalities (routes defined through regular expressions are enclosed in / / instead of ' '), allowing you to create arbitrarily complex patterns.

Express' **string pattern meta-characters** have the following interpretations:

Character	**Description**	**Regex**	**Matched expressions**
+	one or more occurrences	ab+cd	abcd, abbcd, …
?	zero or one occurrence	ab?cd	acd, abcd
*	zero or more occurrences of any char (wildcard)	ab*cd	abcd, ab1234cd, …
[…]	match anything inside for one character position	ab[cd]?e	abe, abce, abde
(…)	boundaries	ab(cd)?e	abe, abcde

These meta-characters can be combined as seen here:

```
app.get('/user(name)?s+', function(req,res){
      res.send(…)
});
```

The string pattern can be deciphered by parsing the pattern from left to right. It matches all routes that:

- start with user,
- are followed by name or '', and
- end with one or more occurrences of s.

And thus /user or /names do *not* match this pattern, but /users or /usernames do indeed match.

Routing parameters

Apart from regular expressions, routing parameters can be employed to enable **variable input** as part of the route. Consider the following code snippet :

```
const express = require("express");
const app = express();

var wishlistPriorities = {
    high: ["Wingspan","Settlers of Catan","Azul"],
    medium: ["Munchkin"],
    low: ["Uno", "Scrabble"]
};

app.get('/wishlist/:priority', function (req, res, next) {
    let list = wishlistPriorities[req.params.priority];
    if (!list) {
        return next();
    }
    res.send(list);
});

app.get("*", function(req, res){
    res.send("No wishlist to return");
});

app.listen(3002);
```

We have defined an object wishlistPriorities which contains high, medium and low priority wishes. We can hardcode routes, for example /wishlist/high to return only the high priority wishes, /wishlist/medium to return the medium priority wishes and /wishlist/low to return the low priority wishes. This is not a maintainable solution though (just think about objects with hundreds of properties). Instead, we create a single route that, dependent on a **routing parameter**, serves different wishlists. This is achieved in the code snippet shown here. The routing parameter priority (indicated with a starting colon :) will match any string that does **not** contain a slash. The routing parameter is available to us in the req.params object. Since the route parameter is called priority (what name we give this parameter is up to us), we access it as req.params.priority. The code snippet checks whether the route parameter matches a property of the wishlistPriorities object and if it does, the corresponding wishlist is returned in an HTTP response. If the parameter does not match any property of the wishlistPriorities object, we make a call to next and move on the next route handler.

Routing parameters can have various levels of nesting:

```
 var wishlistPriorities = {
    high: {
        partyGame: [],
        gameNightGame: ["Wingspan","Settlers of Catan","Azul"]
    },
    medium: {
        partyGame: ["Munchkin"],
        gameNightGame: []
    },
    low: {
        partyGame: ["Uno"],
        gameNightGame: ["Scrabble"]
    }
};
```

```
app.get('/wishlist/:priority/:gameType', function (req, res, next) {
    let list = wishlistPriorities[req.params.priority][req.params.gameType];
    if (!list) {
        return next(); // will eventually fall through to 404
    }
    res.send(list);
});
```

We do not only use the priorities for our wishlist, but also partition them according to whether the wanted games are party games or rather something for a long game night. The route handler now contains two routing parameters, :priority and :gameType. Both are accessible through the HTTP request object. If our corresponding Node.js script runs on our own machine, we can access high priority games for a game night via http://localhost:3002/wishlist/high/gameNightGame. We use the two parameters to access the contents of the wishlistPriorities object. If the two parameters do not match properties of wishlistPriorities, we call next() and otherwise, send the requested response.

Templating with EJS

For live statistics one approach to solve this problem is the use of **Ajax**: the HTML code is *blank* in the sense that it does not contain any user-specific data. The HTML and JavaScript (and other resources) are sent to the client and the client makes an Ajax request to retrieve the user-specific data from the server-side. An alternative to Ajax is **templating**. With **templating, we are able to directly send HTML with user-specific data to the client** and thus remove the extra request-response cycle that Ajax requires:

HTML template + data = Rendered HTML view

With templates, our goal is to write as little HTML by hand as possible. Instead:

- we create a **HTML template** void of any data,
- add data, and
- from template+data generate a rendered HTML view.

This approach keeps the code clean and separates the coding logic from the presentation markup. Templates fit naturally into the *Model-View-Controller* paradigm which is designed to keep logic, data and presentation separate.

EJS example

There are two types of scriptlet tags that **output values**:

- <%= ... %> output the value into the template in **HTML escaped** form. This means that the characters that are indicating the start/end of markup sequences (such as <script> and </script>) are converted in such a way that they are rendered as content instead of being interpreted as markup.
- <%- ... %> output the value into the template in **unescaped** form. This means that a value such as <script> remains as-is. This enables cross-site scripting attacks, which we will discuss in the security lecture.

- <% … %> for control-flow purposes (in javascript):

```
var ejs = require('ejs');

var template = "<%
if(movies.length>2){movies.forEach(function(m){console.log(m.title)})} %>";

var context = {'movies': [
  {title:'The Hobbit', release:2014},
  {title:'X-Men', release:2016},
  {title:'Superman V', release:2014}
]};

ejs.render(template, context);
```

Express and templates

1. We set up the *views directory* - the directory containing all templates. Templates are essentially HTML files with EJS scriptlet tags embedded and file ending .ejs:

```
 app.set('views', __dirname + '/views');
```

2. We define the template engine of our choosing:

```
app.set('view engine', 'ejs');
```

3. We create template files.

An functioning Express/EJS demo can be found here. Try it out yourself by installing and running it. Let’s consider the application’s app.js:

```
var express = require("express");
var url = require("url");
var http = require("http");
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port, function () {
  console.log("Ready on port " + port);
});

var wishlist = [];
wishlist.push({ name: 'Wingspan', type: 'game night'  });
wishlist.push({ name: 'Munchkin', type: 'party game' });
wishlist.push({ name: 'Scrabble', type: 'game night' });
wishlist.push({ name: 'Uno', type: 'party game' });

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');

app.get("/wishlist", function (req, res) {
  res.render('wishlist', { title: 'Game wishlist', input: wishlist });
});
```

We first set up the views directory, then the view engine and finally we use Express' res.render in order to render a view and send the rendered HTML to the client. Important to realize in this example is, that the first argument of res.render is a view stored in views/wishlist.ejs which the Express framework retrieves for us. The second argument is an object that holds the variables of the template, here title and input (an array). To confirm this, let's look at the template file itself, wishlist.ejs which contains the corresponding variable names :

```
<!DOCTYPE html>
<html>
<head>
      <title><%= title %></title>
</head>
<body>
      <h1>Wishlist</h1>
      <div>
            <% input.forEach(function(w) { %>
            <div>
                  <h3><%=w.name%></h3>
                  <p><%=w.type%></p>
            </div>
            <% }) %>
      </div>
</body>
</html>
```

This is mostly HTML, with a few scriptlet tags sprinkled in. While EJS has more capabilities than we present here, for the purposes of our board game project, this excursion into EJS is sufficient.

Node Self-Check

- Node.js has a file-based module system.
  - True.
- Node.js runs each module in a separate scope and only returns the final value of `module.exports`.
  - True.
- The `require` module runs asynchronously.
  - False.
- module.exports` and `exports` can be used in exactly the same way.
  - False.
- Middleware components can execute code.
  - True.
- Middleware components can change the request and response objects.

  - True.
- Middleware components can start the request-response cycle.
  - False.
- Middleware components can call any middleware function in the middleware stack
  - False.

CSS Self-check

- Through which mechanism can data be stored directly in CSS files?
  - The \`content\` attribute.
- A single DOM element cannot contain multiple classes
  - False.
- A \`class\` attribute can be added to any DOM element.
  - True.
- What is the main purpose of a "CSS reset"?
  - To reset the browser-specific default stylesheet to a consistent baseline.
- What is the difference between an inline and a block element?
  - Block-level elements are surrounded by line-breaks. They can contain block-level and inline elements. Inline elements can be placed within block-level or inline elements. They can contain other inline elements.
- What is the difference between \`nth-of-type\` and \`nth-child\`?
  - \`nth-child(X)\` is any element that is the Xth child element of its parent; \`nth-of-type(X)\` is any element that is the Xth sibling of its type.
- What is the purpose of CSS media queries?
  - CSS media queries enable the use of device/media-type dependent style sheets and rules. While the HTML document is written once, the CSS is written once per device type.
- What is the difference between pseudo-classes \`hover\` and \`active\`?
  - \`hover\` is a selector that becomes active when a mouseover on the respective element occurs. In contrast, \`active\` is a selector that is triggered when the element is activated.
- What does the selector combination e1+e2 mean?
  - This combination selects all \`e2\` elements that follow \`e1\` immediately.
- Explain the difference between \`position:absolute\` and \`position:relative\`.
  - The difference lies in the element flow. With \`position:relative\`, the element is adjusted on the fly, other elements are not affected. In contrast, with \`position:absolute\`, the element is taken out of the normal flow (no space is reserved for it).
- When should \`position:fixed\` be used?
  - A major use case are elements that should be visible at all times to the user, no matter the scrolling behavior.

Lecture 7. Cookies, sessions and third-party authentication

HTTP is stateless

every HTTP request contains all information necessary for the server to send a response in reply to a request. The server is not required to keep track of the requests received. This became obvious when we discussed authentication: the client, making an HTTP request to a server requiring authentication will send the username/password combination in every single request. This design decision simplifies the server architecture considerably.

Cookies Introduction

The modern web, however, is **not** stateless. Many web applications track users and their state, for example:

- bol.com keeps users' shopping cart filled even when they leave the site;
- Newspapers such as nytimes.com or nrc.nl track users' number of free articles read per month.
- Users remain logged into portals such as Facebook and Twitter across browser tabs.

Not the stateless web is the norm, but the stateful web. Cookies (and sessions) are one major way to achieve a stateful web. Cookies are **short amounts of text** that are most often **generated by the server, sent to the client** and **stored by the client** for some amount of time. These small amount of texts consist of a key and a value. It should be mentioned that MDN considers cookies an outdated technology - there are a host of other options available today as outlined by MDN. It remains a fact though that cookies are employed by all major websites. The reason is simple: no matter the browser, cookies are supported.

Client-side cookies (i.e., cookies generated by the client and stored by the client, but also sent to a server) also exist, but as we will see later, (1) they do not contribute to making the web stateful, (2) are not very intuitive to use, and (3) better client-side storage options exist.

According to the *HTTP State Management Mechanism* RFC6265, clients (most often browsers) should fulfill the following minimum requirements to store cookies:

- store 4096 bytes per cookie;
- store 50 cookies per domain;
- and at least 3000 cookies in total.

Cookies are old compared to other technologies of the web, they have been around since 1994. This also explains their small size: in those days, the Internet was a very slow piece of technology, to transmit 4KB of data, a dial-up modem took about a second. If a web application has 20 cookies to send, each one 4KB in size, the user will have waited 20 seconds for just the cookies to be send from server to client.

A server-side application creating cookies should use as few cookies as possible and also make those cookies as small as possible to avoid reaching these implementation limits. In the age of video streaming, a few kilobytes worth of cookies seems negligible, however, Internet access is not on the same level in all corners of the world and not every client is a modern browser.

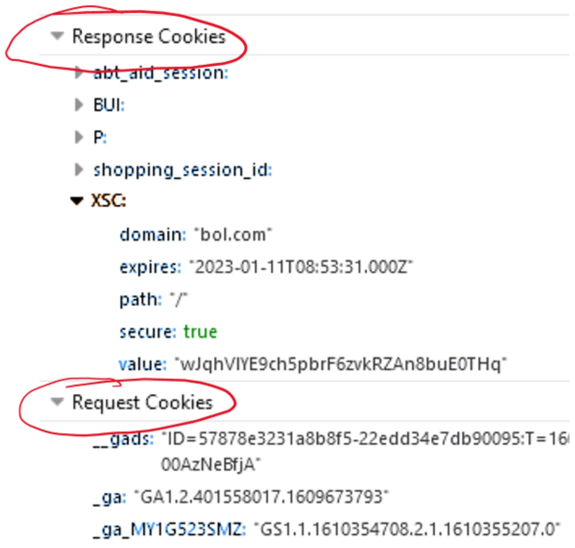
Once you start looking more closely at cookies servers send to clients, you are likely to find cookies with keys like __utma, __utmb or __utmz over and over again. These are Google Analytics cookies, one of the most popular toolkits that web developers use to *track* their web applications' access and usage patterns.

A question we have not yet considered is what actually can be stored in cookies. Cookies are versatile. They act as the **server's short term memory**; the server determines what to store in a cookie, typical examples being:

- the history of a user's page views,
- the setting of HTML form elements (which can also be done fully on the client-side as we will see later), or,
- the user's UI preferences which the server can use to personalize an application's appearance.

Viewing cookies in the browser

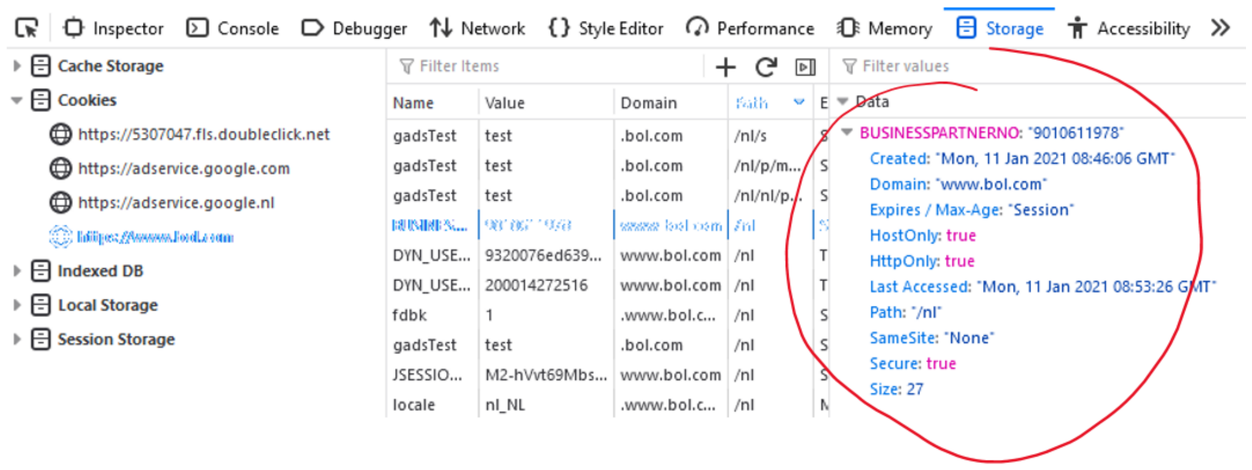
Cookies are **not hidden** from the user, they are stored *in the clear* and can be viewed. Users can also delete and disallow cookies.



```
▼ Response Cookies
  ▶ abt_aid_session:
  ▶ BUI:
  ▶ P:
  ▶ shopping_session_id:
  ▼ XSC:
      domain: "bol.com"
      expires: "2023-01-11T08:53:31.000Z"
      path: "/"
      secure: true
      value: "wJqhVIYE9ch5pbrF6zvkRZAn8buE0THq"
▼ Request Cookies
  __gads: "ID=57878e3231a8b8f5-22edd34e7db90095:T=16
          00AzNeBfjA"
  _ga: "GA1.2.401558017.1609673793"
  _ga_MY1G523SMZ: "GS1.1.1610354708.2.1.1610355207.0"
```

Highlighted are the two categories of cookies: response cookies and request cookies. As the name suggests, *response cookies* are cookies that are appearing in an HTTP response (cookies sent by the server) and *request cookies* are cookies appearing in an HTTP request (cookies sent from client to server). **Importantly, that the client does not send client-generated cookies to the server, the client only *returns* cookies to the server that the server sent to the client beforehand.**

When developing web applications, use Firefox's Storage Inspector tab, which allows us to efficiently debug and investigate cookie settings



Name	Value	Domain	Path	E
gadsTest	test	.bol.com	/nl/s	S
gadsTest	test	.bol.com	/nl/p/m...	S
gadsTest	test	.bol.com	/nl/nl/p...	S
[illegible]	[illegible]	[illegible]	/nl	[illegible]
DYN_USE...	9320076ed639...	www.bol.com	/nl	T
DYN_USE...	200014272516	www.bol.com	/nl	T
fdbk	1	.www.bol.c...	/nl	S
gadsTest	test	.bol.com	/nl	S
JSESSIO...	M2-hVvt69Mbs...	www.bol.com	/nl	S
locale	nl_NL	.www.bol.c...	/nl	N

```
▼ Data
  ▼ BUSINESSPARTNERNO: "9010611978"
      Created: "Mon, 11 Jan 2021 08:46:06 GMT"
      Domain: "www.bol.com"
      Expires / Max-Age: "Session"
      HostOnly: true
      HttpOnly: true
      Last Accessed: "Mon, 11 Jan 2021 08:53:26 GMT"
      Path: "/nl"
      SameSite: "None"
      Secure: true
      Size: 27
```

The screenshot shows the essence of a cookie: a cookie consists of a name/value pair and a number of optional cookie fields. The screenshot also shows that a modern browser has more than just cookie storage available (besides local storage these are beyond the scope of CSE1500).

How cookies are sent and received: in the HTTP header fields `Set-Cookie` and in `Cookie` respectively.

Cookie security

Cookies are just small pieces of text, in the form of key (or *name*) and value. They can be **altered by the user** and **send back** to the server in their **altered form**.

This opens up a line of attack: a server that is trusting all cookies it receives back from clients without further checks, is susceptible to abuse from malicious users. Imagine a web application that determines the role of a user (e.g., on Brightspace we have instructors, graders, students and visitors) based on some criteria, saves this information in a cookie and sends it to its clients. Each time a client makes a request to a server, the server reads out the returned cookie to determine for which role to send back a response. A malicious user can change that role - say, from student to grader - and will receive information that is not intended for her.

Overall, security and privacy are not strong points of cookies and as long as we are aware of this and do not try to transmit sensitive or compromising information within cookies, the potential risks are limited.

The private browsing mode (also known as incognito mode) of modern browsers still allows to receive/send cookies. The cookies though are only held in memory (not stored on disk, independent of the cookie type) and are completely separated from the cookie storage of the regular browsing mode. All cookies are discarded once the private/incognito browser session ends. *Note, that the incognito/private browsing mode does* ***not*** *make you anonymous to servers, instead it makes your data/browsing activities untraceable to other people with physical access to your device and browser.*

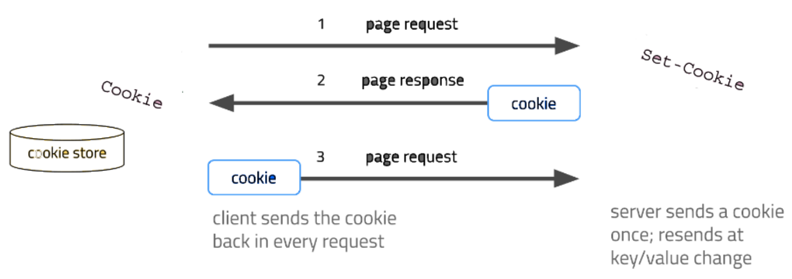
Cookies vs. sessions

Cookies and sessions are closely related. **Sessions make use of cookies**. The difference to the cookie-only setting is the following: a single cookie is generated per client by the server and that cookie only contains a **unique ID**, the rest of the data is **stored on the server** and associated with the client through that ID.

In general, sessions are preferable to cookies as they expose very little information (only a randomly generated ID).

Cookie flow

Consider the diagram below. On the left we have our browser (the client) part of which is a cookie store and on the right we have our server-side application.


1 page request
2 page response
cookie
3 page request
cookie
Cookie
Set-Cookie
cookie store
client sends the cookie back in every request
server sends a cookie once; resends at key/value change

At the first visit to a web application, the client sends an HTTP request not containing a cookie. The server sends an HTTP response to the client including a cookie. Cookies are **encoded in HTTP headers**. At each subsequent HTTP request made to the same server-side application, the browser returns all cookies that were sent from that application. Cookies are actually **bound to a site domain name**, they are only sent back on requests to this specific site - a security feature of the browser. Servers usually only send a cookie once, unless the key/value pair has changed.

Transient vs. persistent cookies

Cookies can either be transient or persistent.

Transient cookies are also called ***session*** *cookies* and only exist in the memory of the browser. They are deleted when the browser is closed. They are *not* deleted when just the browser tab or browser window is closed however! **If a cookie has no explicit expiration date, it automatically becomes a transient cookie.**

Persistent cookies on the other hand remain intact after the browser is closed, they are **stored on disk**. They do have a maximum age and are send back from client to server only as long as they are **valid**. They are valid when they have not yet exceeded their maximum age. (**Time cookies)**

Note: modern browsers offer to ***restore*** *the session after a browser crash. In this case, transient cookies are often restored as well as the browser session is continued. This though is implemented inconsistently across different browsers.*

Cookie fields

Cookies consist of seven components, of which only the first one is a required component:

1. The cookie-name=cookie-value field has to be set for a cookie to be valid;
2. The Expires (expiration date) and Max-Age (seconds until the cookie expires) fields determine whether a cookie is a transient or persistent cookie.
3. The Domain field determines the (sub)domain the cookie is associated with. It is restricted to the same domain as the server is running on.
4. The Path field determines for which paths the cookie is applicable using wildcarding. Setting the path to a slash (/) matches all pages, while /wishlist will match all pages under the /wishlist path and so on.
5. Secure flag: if this flag is set for a cookie it will only be sent via HTTPS, ensuring that the cookie is always encrypted when transmitting from client to server. This makes the cookie less likely to be exposed to cookie theft via eavesdropping. This is most useful for cookies that contain sensitive information, such as the session ID. A browser having stored a secure cookie will not add it to the HTTP request to a server if the request is sent via HTTP.
6. HttpOnly flag: cookies with this flag are not accessible to **non-HTTP entities**. By default, cookies can be read out and altered through JavaScript, which can lead to security leaks. Once this flag is set in a cookie it cannot be accessed through JavaScript and thus no malicious JavaScript code snippet can compromise the content of the cookie. As all cookies, these cookies remain readable to the user that operates the client.
7. Signed flag: signed cookies allow the server to check whether the cookie value has been tampered with by the client. Let’s assume a cookie value monster, the signed cookie value is then s%3Amonster.TdcGYBnkcvJsd0%2FNcE2L%2Bb8M55geOuAQt48mDZ6RpoU. The server signs the cookie by **appending** a base-64 encoded *Hash Message Authentication Code* (HMAC) to the value. Note that the value is still readable, signed cookies offer **no privacy**, they make cookies though robust against tampering. The server stores a *secret* (a non-guessable string) that is required to

compute the HMAC. For a cookie that is returned to the server, the server recomputes the HMAC of the value and only if the computed HMAC value matches the HMAC returned to the server, does the server consider the cookie value as untampered. Unless the server has an easily guessable secret string (such as the default secret string of the server framework), this ensures protection against tampering.

Cookie field Domain

Cookie fields are generally easy to understand. There is one field though which requires a more in-depth explanation and that is the Domain field.

Each cookie that is sent from a server to a client has a so-called **origin**, that is the **request domain** of the cookie. For example if a client makes a GET request to http://www.my_site.nl/wishlist the request domain of the cookie the server sends in the HTTP response is www.my_site.nl. Even if the port or the scheme (http vs https) differ, the received cookie is still applicable. That is, our cookie with the request domain www.my_site.nl will also be returned from the client to the server if the next request the client makes is to https://www.my_site.nl:3005.

If the Domain field is not set, the cookie is only applicable to its request domain. This means that the cookie with request domain www.my_site.nl is not applicable if the next request from the client is made to http://my_site.nl - note the lack of the www. prefix.

If the Domain field is set however, the cookie is applicable to the **domain listed in the field and all its sub-domains**. Importantly, the Domain field has to cover the request domain as well.

```
GET http://www.my_site.nl/wishlist
Set-Cookie: name=value; Path=/; Domain=my_site.nl
```

is applicable to

```
www.my_site.nl
wishlist.my_site.nl
serverA.admin.wishlist.my_site.nl
```

Return to sender …

If you carefully look at the cookies that are sent in HTTP request/response messages you may notice that HTTP requests only contain name/value pairs (i.e., the data sent from the browser to the server consists only of name/value pairs) while the HTTP responses in addition to name/value pairs contain a variety of cookie fields. **The client** (in our case the browser) **relies on the information in the cookie fields to determine how to store the cookies and when to send them back to the server as part of an HTTP request.**

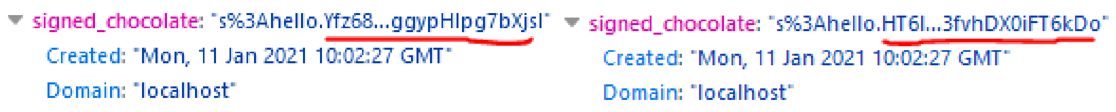
A Node.js application

Dedicated **middleware** makes the usage of cookies with Express easy.

Since cookies can be modified by a malicious user we need to be able to verify that the returned cookie was created by our application server. That is what the `Signed` flag is for.

To make cookies secure, a **cookie secret** is necessary. The cookie secret is a string that is known to the server and used to compute a hash before they are sent to the client. The secret is ideally a random string. Such as:

```
module.exports = {
    cookieSecret: "L3dOX2nGjdNnzTZ97r0lawA4dU"
};
```



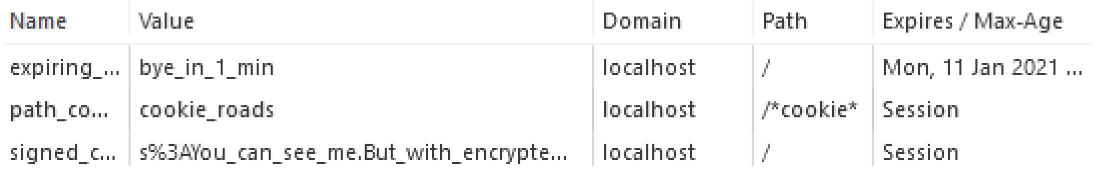
```
▼ signed_chocolate: "s%3Ahello.Yfz68...ggypHlpg7bXjsl"
    Created: "Mon, 11 Jan 2021 10:02:27 GMT"
    Domain: "localhost"

▼ signed_chocolate: "s%3Ahello.HT6l...3fvhDX0iFT6kDo"
    Created: "Mon, 11 Jan 2021 10:02:27 GMT"
    Domain: "localhost"
```

You can see the different parsed value from a different seed.

It is a common practice to externalize third-party credentials, such as the cookie secret, database passwords, and API tokens. Not only does this ease maintenance (by making it easy to locate and update credentials), it also allows you to omit the credentials file from your version control system. This is especially critical for open source repositories hosted on platforms such as GitHub. In demo-code/node-cookies-ex, the credentials are stored in `credentials.js` (which **for demo purposes** is actually under version control): it is a module that exports an object, which contains the `cookieSecret` property, but could also contain database logins and passwords, third-party authentication tokens and so on.

```
var credentials = require("./credentials"); //signature seed
var cookies = require("cookie-parser"); //middleware
app.use(cookies(credentials.cookieSecret));

app.get("/sendMeCookies", function (req, res) {
    res.cookie("path_cookie", "cookie_roads", {path: "/*cookie*"});//default path
 is the current one
    res.cookie("expiring_cookie", "bye_in_1_min",{expires: new Date(Date.now() +
60000)}); //deafult expire is this session
    res.cookie("signed_cookie", "You_can_see_me.But_with_encrypted_signature", {
signed: true, }); //default signed is false
    res.send("Cookies sent to client"); //end the request
});
```

Cookies the client sends back to the server appear in the HTTP request object and can be accessed through req.cookies. Here, a distinction is made between signed and unsigned cookies: you can only be sure that the signed cookies have not been tampered with.



Name	Value	Domain	Path	Expires / Max-Age
expiring_...	bye_in_1_min	localhost	/	Mon, 11 Jan 2021 ...
path_co...	cookie_roads	localhost	/*cookie*	Session
signed_c...	s%3AYou_can_see_me.But_with_encrypte...	localhost	/	Session

Accessing and deleting cookies in Express

To access a cookie value, append the cookie key to req.cookies or req.signedCookies:

```
app.get("/listAllCookies", function (req, res) {
    console.log("++ unsigned ++");
    console.log(req.cookies.path_cookie,req.cookies["expiring_cookie"]); //ac
cess a specific cookie
    console.log("++ signed ++");
    console.log(req.signedCookies); //lists all cookies
    res.clearCookie("signed_cookie"); //we can manually expire them and set t
he path to / (so we can delete it for all paths)
    res.clearCookie("path_cookie");
    res.clearCookie("expiring_cookie");
    res.send("");
});
```

In order to delete a cookie we call the function clearCookie in the HTTP **response** object:

If we dig into the Express code, in particular response.js, we find clearCookie to be defined as follows:

```
res.clearCookie = function clearCookie(name, options) {
  var opts = merge({ expires: new Date(1), path: '/' }, options);

  return this.cookie(name, '', opts);
};
```

This means, that, in order to clear a cookie, the server sends the cookie to the client with an expiration date **in the past**. This informs the browser that this cookie has become invalid and the browser deletes the cookie from its cookie storage. In order to delete a cookie successfully, not only the name has to match but also the cookie domain and path.

Third-party cookies

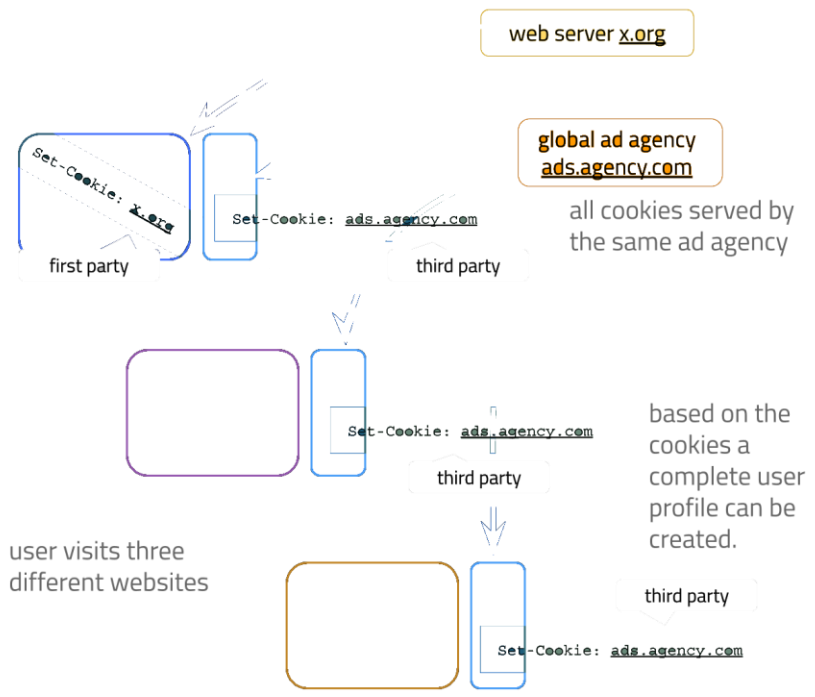
We distinguish two types of cookies, based on the browser's address bar:

- **first-party** cookies, and,
- **third-party** cookies.

While cookies have many beneficial uses, they are also often associated with user tracking. Tracking occurs through the concept of third-party cookies.

First-party cookies are cookies that belong to the same domain that is shown in the browser's address bar (or that belong to the sub domain of the domain in the address bar). Third-party cookies are cookies that belong to domains *different* from the one shown in the browser's address bar.

Web portals can feature content from third-party domains (such as banner ads), which opens up the potential for tracking users' browsing history.


web server x.org
global ad agency
ads.agency.com
all cookies served by the same ad agency
Set-Cookie: x.org
first party
Set-Cookie: ads.agency.com
third party
Set-Cookie: ads.agency.com
third party
based on the cookies a complete user profile can be created.
user visits three different websites
third party
Set-Cookie: ads.agency.com

Here, we suppose a user visits x.org. The server replies to the HTTP request, using Set-Cookie to send a cookie to the client. This is a first-party cookie. x.org also contains an advert from ads.agency.com, which the browser loads as well. In the corresponding HTTP response, the server ads.agency.com also sends a cookie to the client, this time belonging to the advert's domain (ads.agency.com). This is a third-party cookie. This by itself is not a problem. However, the global ad agency is used by many different websites, and thus, when the user visits other websites, those may also contain adverts from ads.agency.com. Eventually, all cookies from the domain ads.agency.com will be sent back to the advertiser when loading any of their ads or when visiting their website. The ad agency can then use these cookies to build up a browsing history of the user across all the websites that show their ads.

Thus, technologically **third-party cookies are not different from first-party cookies**.

Evercookie

Storing small pieces of information **somewhere** in the browser can actually be accomplished in many different ways if one is familiar with the technologies: cookies can be stored in local storage, session storage, IndexedDB and so on

Evercookie is a JavaScript API that does exactly that. It produces extremely persistent cookies that are not stored in the browser's standard cookie store, but elsewhere. Evercookie uses several types of storage mechanisms that are available in the browser and if a user tries to delete any of the cookies, it will recreate them using each mechanism available. Note: *this is a tool which should* ***not*** *be used for any type of web application used in production, it is however a very good educational tool to learn about different components of the browser.*

Browser fingerprinting

Besides all sorts of client-side storage, it is also possible to collect a relatively stable and unique *fingerprint* of a particular browser with a few of the browser's web API calls.

In this manner, users can be tracked as long as they use the same browser on the same device. One defense against browser fingerprinting is for instance the randomization of specific browser functions. This is an active area of security research. This problem is also significant enough, that the W3C has an entire working group dedicated to mitigating browser fingerprinting in web specifications.

Client-side cookies

These are cookies which are created by the client itself but operate just as if they were sent by the server. All saved cookies will appear in all the request headers

To set a client-side cookie, usually JavaScript is employed. A standard use case is a web form, which the user partially filled in but did not submit yet. Often it is advantageous to keep track of the information already filled in and to refill the form with that data when the user revisits the form. In this case, keeping track of the form data can be done with client-side cookies (i.e. cookies that never leave the client).

```
//set TWO(!) cookies
document.cookie = "name1=value1";
document.cookie = "name2=value2; expires=Fri, 24-Jan-2025 12:45:00 GMT";

//delete a cookie by RESETTING the expiration date
document.cookie = "name2=value2; expires=Fri, 24-Jan-1970 12:45:00 GMT";
```

To set a cookie we assign a name/value to document.cookie. document.cookie is a string containing a semicolon-separated list of all cookies. Each time we make a call to it, we can only assign a single cookie. Thus, LINE 2 does not replace the existing cookie, instead we have added a second cookie to document.cookie. The cookie added in LINE 3 showcases how to set the different cookie fields. Deleting a cookie requires us to set the expiration date to a **date in the past**; assigning an empty string to document.cookie will **not** have any effect.

The fact that cookies are appended one after the other in document.cookie also means that we cannot access a cookie by its name. Instead, the string returned by document.cookie has to be parsed, by first splitting the cookies into separate strings based on the semicolon and then determining field name and field value by splitting on =.

Local Storage

Local storage persists data on disk and has an intuitive programming interface:

```
localStorage.setItem('1600637','Why not use something else?'); //add a data
item (key/value must be strings)
let note = localStorage.getItem('1600637'); //retrieve an item with a
specific key
localStorage.removeItem('1600637'); //remove an item
```

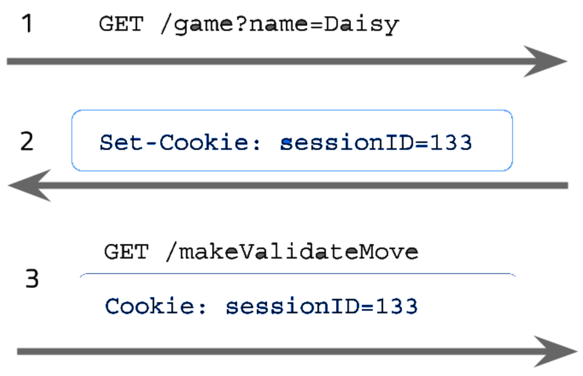
Since localStorage.setItem() only supports strings as key and value, JavaScript objects need to be transformed into strings first via JSON.stringify. The W3C's web storage recommendation mentions a maximum storage size of 5MB per origin but also notes that this is a *mostly arbitrary limit*.

Sessions

Let's now turn to sessions. Sessions make use of cookies. Sessions improve upon cookies in two ways:

- they enable tracking of user information without too much reliance on the unreliable cookie architecture;
- they allow server-side applications to store much larger amounts of data.

However, we still have the problem that without cookies, the server cannot tell HTTP requests from different clients apart. Sessions are a hybrid between cookies and server-side saved data.



A client visits a web application for the first time, sending an HTTP GET request to start a game. The server checks the HTTP request and does not find any cookies; and thus the server randomly generates a session ID and returns it in a cookie to the client. This is the piece of information that will identify the client in future requests to the server. The server uses the session ID to look up information about the client, usually stored in a database. This enables servers to store as much information as necessary, without hitting a limit on the number of cookies or the maximum size of a cookie.

For this process to be robust, the session IDs need to be generated at random. If we simply increment a session counter for each new client that makes a request we will end up with a very insecure application. Malicious users can snoop around by randomly changing the session ID in their cookie. Of course, this can partially be mitigated by using signed cookies, but it is much safer to not let clients guess a valid session ID at all.

Sessions are easy to set up, through the use of another middleware component: `express-session`. The most common use case of sessions is authentication, i.e. the task of verifying a user's identity.

Simple example of views counter:

```
var express = require("express");
var http = require("http");

/* Cookie and session setup */
var credentials = require("./credentials");
var cookies = require("cookie-parser");
var sessions = require("express-session");
/* Setup complete */
```

```
var app = express();

app.use(cookies(credentials.cookieSecret));
var sessionConfiguration = {
      // Code is slightly adjusted to avoid deprecation warnings when running
the code.
      secret: credentials.cookieSecret,
      resave: false,
      saveUninitialized: true,
};
app.use(sessions(sessionConfiguration));
http.createServer(app).listen(3000);

app.get("/countMe", function (req, res) {

  /* session object available on req object only */
      var session = req.session;

      if (session.views) { /* the session exists! */
            session.views++;
            res.send("You have been here " + session.views + " times (last
visit: " + session.lastVisit + ")");
            session.lastVisit = new Date().toLocaleDateString();
      }
      else { /* the session does not exist */
            session.views = 1;
            session.lastVisit = new Date().toLocaleDateString();
            res.send("This is your first visit!");
      }
});
```

Here, we store the session information in memory, which of course means that when the server fails, the data will be lost. In most web applications, we would store this information eventually in a database. To set up the usage of sessions in Express, we need two middleware components: `cookie-parser` and `express-session`. Since sessions use cookies, we also need to ensure that our middleware pipeline is set up in the correct order: the `cookie-parser` should be added to the pipeline before `express-session`, otherwise this piece of code will lead to an error.

We define one route, called /countMe, that determines for a client making an HTTP request, how many requests the client has already made. Once the session middleware is enabled, session variables can be accessed on the session object which itself is a property of the request object - req.session. This is the first course of action: accessing the client's session object. If that session object has a property views, we know that the client has been here before. We increment the views count and send an HTTP response to the client, informing it about how often the client has been here and when the last visit was. Then we set the property lastVisit to the current date and are done. If session.views does not exist, we create the views and lastVisit properties and set them accordingly, returning a *This is your first visit* as content in the HTTP response. Finally, it is worth noting that all session-related actions are performed on the request object.

Third-party authentication

The final topic of this lecture is third-party authentication. This is a topic easily complex enough to cover a whole lecture. We introduce the principles of third-party authentication, but do not dive into great detail of the authentication protocol.

Even if you are not aware of the name, you will have used third-party authentication already. In many web applications that require a login, we are given the choice of either creating a username/password or by signing up through a third party such as Facebook, Google or Twitter.

Third-party authentication has become prevalent across the web, because **authentication**, i.e. the task of verifying a user's identity, is hard to do right.

If an application implements its own authentication scheme, it has to ensure that the information (username, password, email) are stored safely and securely and not accessible to any unwanted party. Users tend to reuse logins and password and even if a web application does not contain sensitive information, if the username/passwords are stolen, users might have used the same username/password combination for important and sensitive web applications such as bank portals, insurance portals, etc.

To avoid these issues, application developers *out-source* authentication to large companies that have the resources and engineering power to guarantee safe and secure storage of credentials. If an application makes use of third-party authentication, it **never has access to any sensitive user credentials**.

There are two drawbacks though:

- we have to **trust** that the web platform providing authentication is truthful;
- some of our users may not want to use their social web logins to authenticate to an application.

The protocol that governs most third-party authentication services today is the **OAuth 2.0 Authorization Framework**, standardized in RFC6749. Its purpose is the following:

The OAuth 2.0 authorization framework enables a third-party application

to obtain limited access to an HTTP service, either on behalf of a resource

owner by orchestrating an approval interaction between the resource owner

and the HTTP service, or by allowing the third-party application to obtain

access on its own behalf.

OAuth 2.0 roles

The OAuth 2.0 protocol knows several roles:

Role	Description
Resource owner	Entity that grants access to a protected resource
Resource server	Server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

Client	An application making protected resource requests on behalf of the resource owner and with its authorization.
Authorization server	Server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

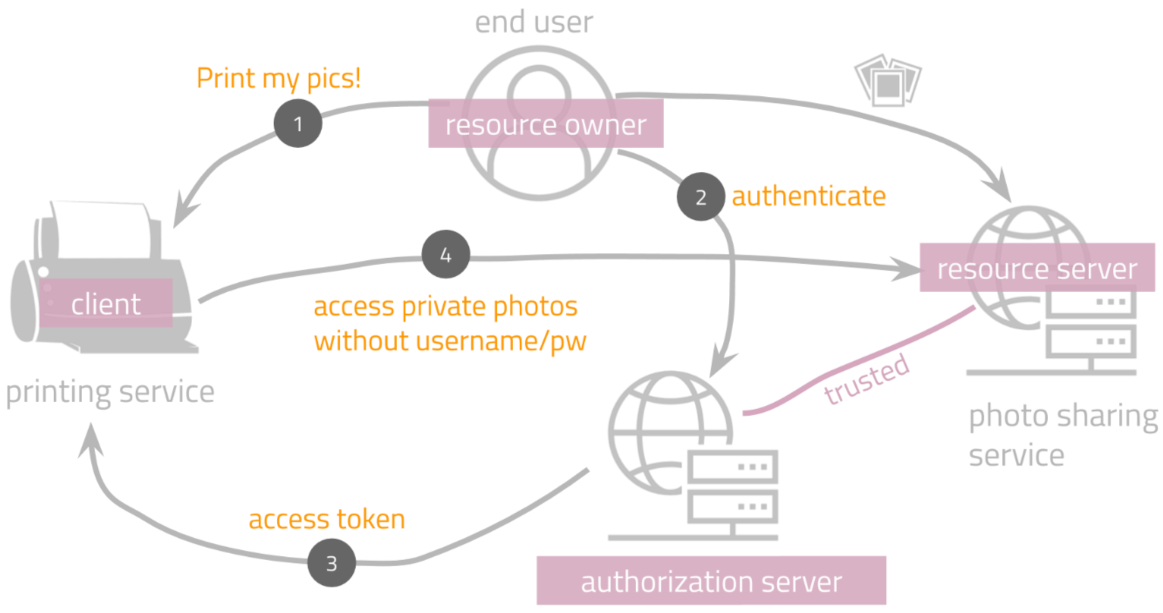
The **access token** referred to in the resource server role is a string denoting a *specific scope, lifetime and other access attributes*.

Roles exemplified

Let's consider this specific example: an end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo-sharing service (resource server), without sharing her username and password with the printing service.

The user authenticates directly with a server trusted by the photo-sharing service(authorization server), which issues the printing service delegation-specific credentials (access token).

The mapping between the entities and OAuth 2.0 roles is as follows:



Express

To incorporate third-party authentication in an Express application, incorporate an authentication middleware component that is OAuth 2.0 compatible. A popular choice of middleware is passport, which incorporates a range of authentication protocols across a number of third-party authenticators.

Self-check

- Third-party cookies and first-party cookies are stored in different cookie storage facilities within the browser.
    - False.
- Third-party cookies originate from a different domain than first-party cookies.
    - True.

- Third-party cookies have a lower priority than first-party cookies and are returned to the server only after any first-party cookies.
    - False.
- A server-side application uses sessions to track users. What is the common approach to determine the end of a session?
    - If x seconds have passed without a request from the client, the server ends the session.
- What is the main purpose of the "consumer secret" (or "client secret") in third-party authentication?
    - It ensures that only authorized applications query the authentication server for an access token.
- A signed cookie enables the server to issue an encrypted value to the client, that cannot be decrypted by the client.
    - False.
- A signed cookie enables the server to verify that the issued cookie is returned by the client unchanged, without having to store the original issued cookie on the server.
    - True.
- The value of a signed cookie is encrypted with HMAC to avoid man-in-the-middle attacks. The client can decrypt the value with a previously negotiated key.
    - False.
- If a cookie's `HTTPOnly` flag is set, the user cannot view or change the cookie in the browser.
    - False.

```
Set-Cookie: bg=white; Expires=Fri, 01-Aug-2016 21:47:38 GMT; Path=/;
Domain=tudelft.nl
Set-Cookie: pref=1; Path=/; Domain=tudelft.nl
Set-Cookie: dom=23; Expires=Thu, 01-Jan-2023 00:00:01 GMT; Path=/;
Domain=tudelft.nl; HttpOnly
Set-Cookie: view=mobile; Path=/; Domain=tudelft.nl; secure
```

- The browser B currently has no stored cookies. The server sends the four cookies above to B (assume this is happening today). B crashes 10 minutes later and the user restarts B. How many cookies can the user access after the restart with client-side JavaScript?
    - 0

Lecture 8. Web security

Web applications are an attractive target for *attackers* (also known as *malicious users*) for several reasons:

- Web applications are open to attack from **different angles** as they rely on various software systems to run: an attacker can go after the **web server** hosting the web application, the **web browser** displaying the application and the **web application** itself. The **user**, of course, is also a point of attack.
- Successfully attacking a web application with thousands or millions of users offers a lot of potential gain.
- "Hacking" today does not require expert knowledge, as easy-to-use automated tools are available that test servers and applications for known vulnerabilities (e.g. w3af).

When developing a web application, it is important to ask yourself **how can it be attacked?** and secure yourself against those attacks. While web applications are relatively easy to develop thanks to the tooling available today, they are difficult to secure as that step requires substantial technological understanding on the part of the developer. Large-scale web portals such as Facebook have partially outsourced the finding of security issues to so-called *white hat hackers* - people interested in security issues that earn money from testing companies' defenses and pointing them towards specific security issues.

Defacement

Website defacement is an attack against a website that changes the visual appearance of a site. It can be an act of hacktivism (socio-politically motivated), revenge, or simply internet trolling.

Data disclosure

Data disclosure is a threat that is in the news, when a large company fails to protect sensitive or confidential information from users who should not have access to it.

Data loss

This threat is the most devastating for organizations that do not have proper backups in place: attackers are deleting data from servers they infiltrate.

Denial of service

Denial of service (DoS) is a disruption attack that makes web applications unavailable for legitimate users. A signature of a DoS attack is the abnormal traffic increase.

A variant of a DoS attack is a *Distributed Denial of Service* (DDoS) attack where specifically **multiple** systems flood a targeted system. Typically, an attacker recruits multiple vulnerable machines (or bots) to join a *Botnet* for DDoS attacks.

Unauthorized access

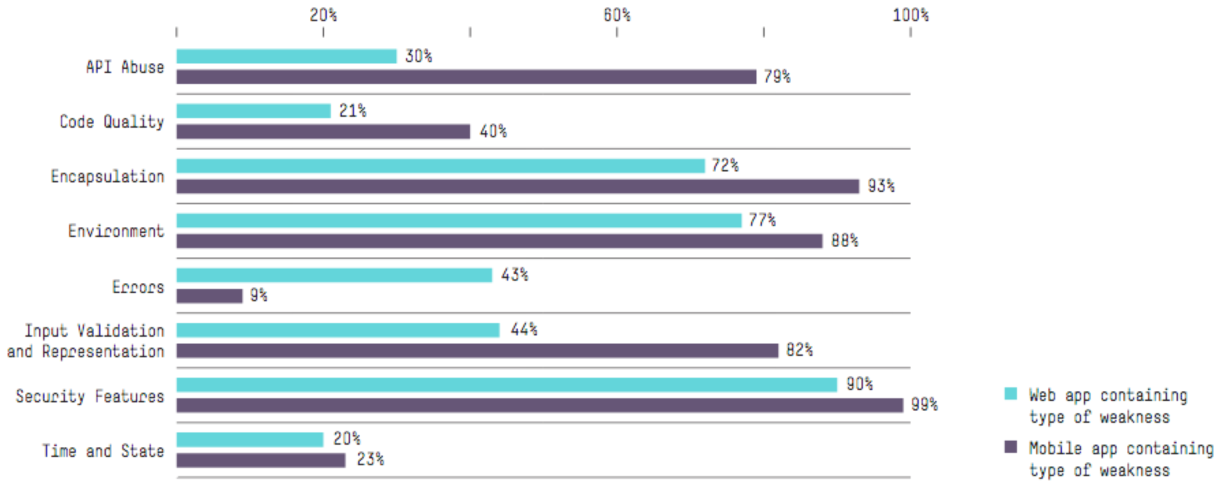
The most difficult component of a system to secure is its users. **Phishing** and **social engineering** can lead unsuspecting users to give access to some part of the secured system to attackers. Once in, attackers try to infiltrate other internal systems.

Unauthorized access can also be achieved by probing web applications for functionalities that should not be accessible to the average user. An example is Instagram's backend admin panel which was accessible on the web while it should have only been accessible from the internal Instagram network.

Typosquatting attacks

The npm registry has its share of *typosquatting attacks* (e.g. here, here and here): malicious users upload packages to the npm registry with names that are typos of popular repository names (e.g. electorn instead of electron). If an unsuspecting developer is not careful and types npm install electorn instead of the wanted npm install electron, and starts up a Node.js script with this package included, undesired code is executed. In the case of electorn, npm's security team has removed the malicious code from the registry but maintained a placeholder.

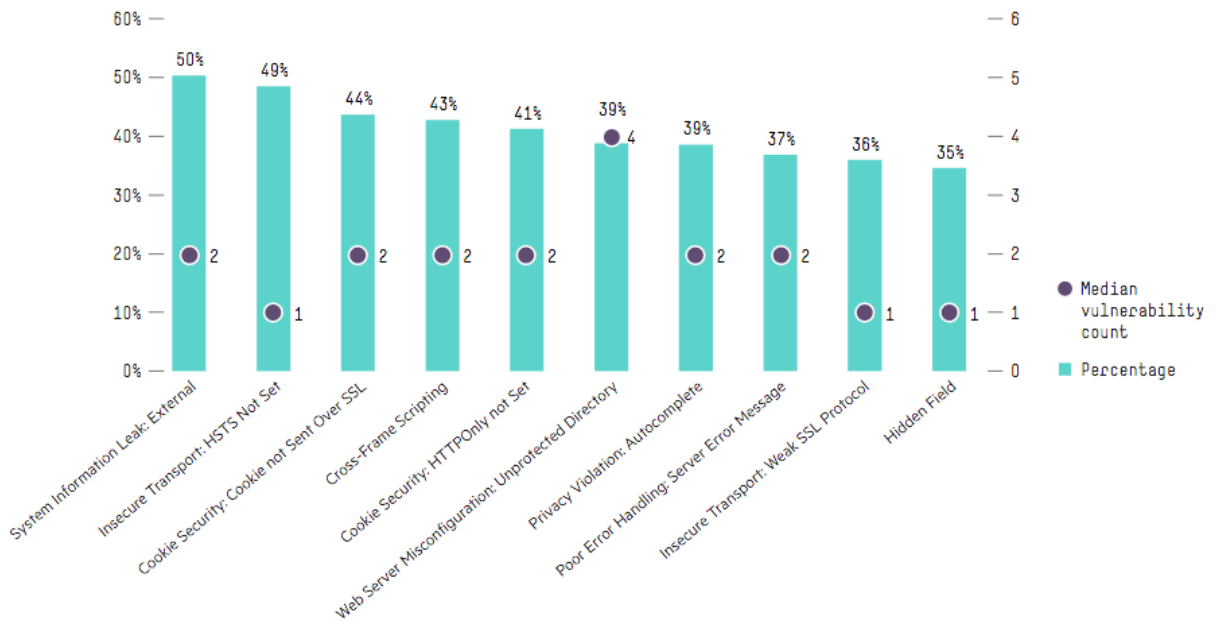
Most frequent vulnerabilities



Most important software security issues. Figure taken from page 56, CSRHPE.

In general, mobile applications are more vulnerable than web applications; the worst issues were found in the *security features* category, which includes authentication, access control, confidentiality and cryptography issues. 99% of mobile applications had at least one issue here. The *environment* category is also problematic with 77% of web applications and 88% of mobile applications having an issue here - this refers to server misconfigurations, improper file settings, sample files and outdated software versions. The third category to mention is *input validation and representation* which covers issues such as cross-site scripting and SQL injections, that are present in most mobile applications and 44% of web applications. The latter is actually surprising, as a lot of best practices of how to secure web applications exist - clearly though, these recommendations are often ignored.

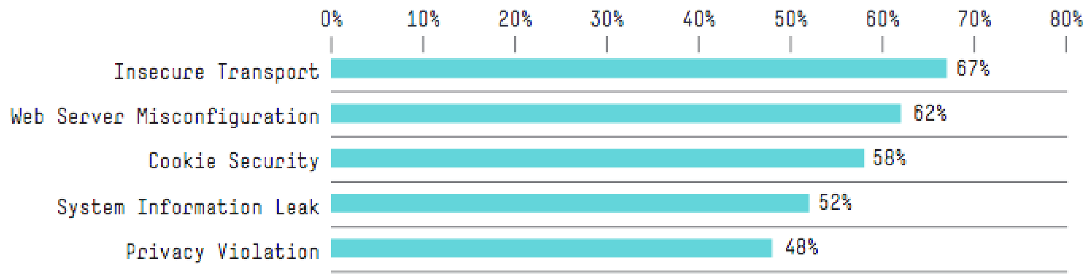
If we zoom in on the non-mobile applications, the ten most commonly occurring vulnerabilities are the following, reported as the *percentage of applications* and *median vulnerability count*:



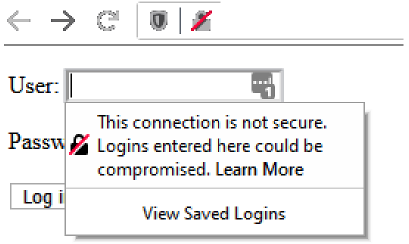
Top 10 vulnerabilities. Figure taken from page 57, CSRHPE.

Some of these vulnerabilities you should already recognize and be able to place in context, specifically *Cookie Security: cookie not sent over SSL* and *Cookie Security: HTTPOnly not set*. The vulnerability *Privacy violation: autocomplete* should intuitively make sense: auto-completion is a feature provided by modern browsers; browsers store information submitted by the user through <input> fields. The browser can then offer autocompletion for subsequent forms with similar field names. If sensitive information is stored in this manner, a malicious actor can provide a form to a user that is then auto-filled with sensitive values and transmitted back to the attacker. For this reason, it is often worthwhile to switch off autocompletion for sensitive input fields.

Lastly, let's discuss the *Hidden field* vulnerability. It provides developers with a simple manner of including data that should not be seen/modified by users when a <form> is submitted. For example, a web portal may offer the same form on every single web page and the hidden field stores a numerical identifier of the specific page (or route) the form was submitted from. However, as with any data sent to the browser, with a bit of knowledge about the dev tools available in modern browsers, the user can easily change the hidden field values, which creates a vulnerability if the server does not validate the correctness of the returned value. Taking a slightly higher-level view, the top five violated security categories across all scanned applications are the following, reported as *percentage of applications violating a category*:



Insecure Transport



This connection is not secure. Logins entered here could be compromised. Learn More

This refers to the fact that applications rely on insecure communication channels or weakly secured channels to transfer sensitive data. Nowadays, at least for login/password fields, the modern browsers provide a warning to the user indicating the non-secure nature of the connection. It is worth noting that in recent years browsers have implemented support for the `Strict-Transport-Security` header, which allows web applications to inform the browser that it should **only** be accessed via HTTPS.

OWASP Top 10 vulnerabilities

OWASP is an acronym and stands for Open Web Application Security Project and is an organization whose mission is to improve software security. Creating a vulnerable application to showcase the worst security issues is one way to train software engineers in web security. The OWASP Top 10 vulnerabilities are those vulnerabilities that - by consensus among security experts - are the most critical security risks to web applications. Below these, in no particular order.

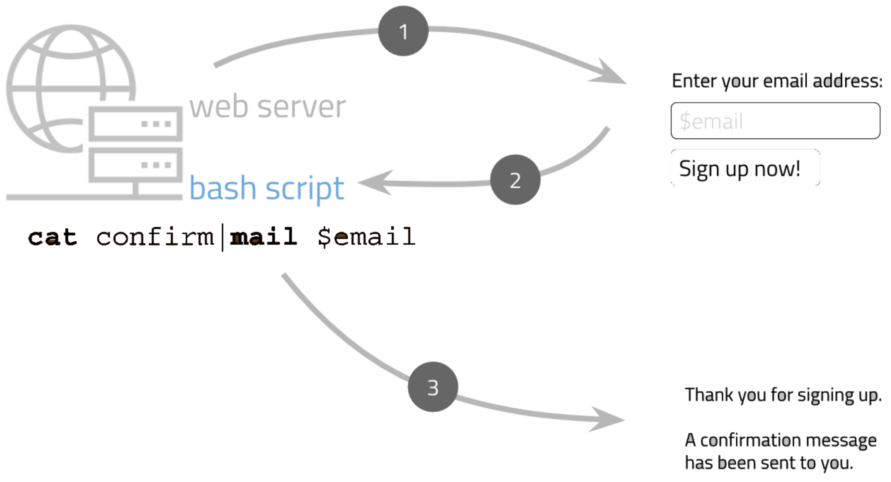
Injection

Injection attacks exploit the fact that input is interpreted by the server without any checks. A malicious user can create input that leads to unintended command executions on the server.

Input for injection attacks can be created via:

- parameter manipulation of HTML forms (e.g. input fields are filled with JavaScript code);
- URL parameter manipulation;
- HTTP header manipulation;
- hidden form field manipulation;
- cookie manipulation.

Injection attacks on the server can take multiple forms, we first consider **OS command injection.**


1
2
3
web server
bash script
Enter your email address:
$email
Sign up now!
Thank you for signing up.
A confirmation message has been sent to you.

```
cat confirm|mail $email
```

Here, we have a web portal that allows a user to sign up to a newsletter. The form looks simple enough: one <input type="text"> element and a <button> to submit the form. On the server-side, a bash script takes a fixed confirmation string (stored in file confirm) and sends an email to the email address as stated in the user's input (*Thank you for signing up for our mailing list.*). This setup of course assumes, that the user actually used an email address as input. Let's look at benign and malicious user input:

- The benign input john@test.nl leads to the following OS command: cat confirm|mail john@test.nl. This command line is indeed sufficient to send an email, as Linux has a command line mail tool.
- An example of malicious input is the following: john@test.nl; cat /etc/password | mail john@test.nl. If the input is not checked, the server-side command line will look as follows: cat confirm | mail john@test.nl; cat /etc/password | mail john@test.nl. Now, two emails are sent: the confirmation email and a mail sending the server's file /etc/password to john@test.nl. This is clearly *unintended code execution*.

The main issue here is the lack of **input validation**. It should not be assumed that any input is the desired/wanted input, this has to be validated.

Another instance of an injection attack: Imagine a calculator web application that allows a client to provide a formula, which is sent to the server, evaluated on the server inside a Node.js script, and the final result is sent back to the client. JavaScript offers an eval() function that takes a string representing JavaScript code and runs it, e.g. the string 100*4+2 can be evaluated with eval('100*4+2'), resulting in 402. However, a malicious user can also try to input while(1) or process.exit(); the former leads the Node.js event loop to be stuck forever in the while loop, while the latter instructs Node.js to terminate the running process. eval() in fact is so dangerous that it should never be used.

SQL injection attacks. They are a regular occurrence when input is not validated

Consider this example code snippet below :

```
let n = /* code to retrieve user provided name */
let p = /* code to retrieve user provided password */

/* a database table 'users' holds our user data */
let sqlQuery = "select * from users where name = '"+u+"' and password = '" +
p + "'";
/* execute query */
```

A benign user input such as john as username and my_pass as password will lead to the SQL query select * from users where name='john' and password='my_pass'. Once we know (or guess) how the SQL query is constructed, we can construct malicious input that allow us to retrieve the row for user john without knowing the correct password. For example:

- The username john'-- with any password will lead to the following SQL query select * from users where name='john'-- and password=''. Here, the fact that we can add comments within SQL statements is exploited to remove the requirement for the correct password.

- The username john with password anything' or '1'='1 will lead to the SQL query select * from users where name='john' and password='anything' or '1'='1'.

How to avoid it

Injection attacks can be avoided by **validating** user input (e.g., is this input really an email address?) and **sanitizing** it (e.g., by stripping out potential JavaScript/SQL code elements). **These steps should occur on the server-side, as a malicious user can always circumvent client-side validation/sanitation steps.**

A popular Node package that validates and sanitizes user input is validator. For example, to check whether a user input constitutes a valid email address, the following two lines of code are sufficient:

```
var validator = require('validator');
var isEmail = validator.isEmail('while(1)'); //false
```

To avoid SQL injection attacks, sqlstring is a Node.js package that escapes user provided input. Even better, instead of writing SQL queries on the fly (so-called dynamic queries as shown above), use **prepared statements** as described here for PostgreSQL.

Broken authentication and session management

Quoting OWASP: *"Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently."*.

We here focus in particular on technical weaknesses, though phishing and other social techniques are often employed to guess passwords and answers to security questions (which in turn allow password resets).

Recall that in order to establish *sessions*, cookies are used. A cookie stores a randomly generated user ID on the client, the remaining user information is stored on the server.

An attacker can exploit broken authentication and session management functions to impersonate a user. In the latter case, the attacker only needs to acquire knowledge of a user's session cookie ID. This information can be revealed to an attacker in several ways:

- Via the usage of **URL rewriting** to store session IDs. Imagine the following scenario: a bookshop supports URL rewriting and includes the session ID in the URL, e.g., http://mybookshop.nl/sale/sid=332frew3FF?basket=B342;B17. An authenticated user of the shop (who has stored her credit card information in it) wants to let others know about her buying two books. She e-mails the link without realizing that she is giving away her session ID. Anyone with the link can use her session (at least for a short period of time) and is thus able to use her credit card to buy products.

- When **storing a session ID in a cookie without informing the user**. A user may use a public computer to access a web application that requires authentication. Instead of logging out, the user simply closes the browser tab. Closing the browser tab does NOT delete a session cookie though. An attacker uses the same browser and application a few minutes later - the original user will still be authenticated.

- When sending **session ID via HTTP** instead of HTTPS. In this case, an attacker can listen to the network traffic and read out the session ID. The attacker can then access the application without requiring the user's login/password.
- When relying on **static session IDs** instead of regularly changing ones. If session IDs are not regularly changed, they are more easily guessable.
- When **session IDs are predictable.** Once an attacker gains knowledge of how to generate valid session IDs, the attacker can wait for a user with valuable information to pass by.

Guessing passwords:

The very popular GitHub repo SecLists contains among others collections of common usernames and common passwords to aid *penetration testers* (i.e. testers evaluating the security of applications) in their work. If an application does not control the maximum number of input attempts in a certain time window (Juice Shop indeed does not), an automated script could be employed to run through all common username/password combinations until a valid username/password combination is found.

How to avoid it

- Good authentication and session management is difficult - avoid, if possible, an implementation from scratch.
- Ensure that the session ID is never sent over the network **unencrypted**. We have already learnt in the previous lecture that the Secure flag can be set for a cookie to ensure that.
- Session IDs should not be visible in URLs.
- Generate a new session ID on login and **avoid reuse**.
- Session IDs should have a timeout and be invalidated on the server after the user ends the session.
- Conduct a sanity check on a request's HTTP header fields (referer, user agent, etc.). Valid HTTP requests typically come from a particular set of pages within the web application. From which page a request was made is visible in the referer request header field.
- Ensure that users' login data is stored securely.
- Do not allow unlimited login attempts to avoid automated scripts trialing common username/password combinations.
- Disallow users to use known weak passwords.
- Add two-factor (or more generally multi-factor) authentication if possible: in this manner, even a guessed username/password combination is not sufficient if the signin happens from a device that has never been used before as an additional verification step is required (e.g. a particular token sent to your phone).

XSS

XSS stands for **cross-site scripting**.

"XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content." (OWASP)

The browser executes JavaScript code all the time; this code is **not** checked by anti-virus software. The browser's sandbox is the main line of defense.

XSS attacks come in two flavors: stored XSS and reflected XSS.

Stored XSS: the injected script is **permanently stored on the target server** (e.g. in a database or text file). The victim retrieves the malicious script from the server, when she requests the stored information. This attack is also known as **persistent or Type-I** XSS.

A common example of stored XSS are forum posts: if a malicious user is able to add a comment to a page that is not validated by the server, the comment can contain JavaScript code. The next user (victim) that views the forum posts receives the forum data from the server, which now includes the malicious code. This code is then executed by the victim's browser.

```
http://myforum.nl/add_comment?c=Let+me+…

http://myforum.nl/add_comment?c=<script>…
```

In a **reflected XSS** attack (also known as **non-persistent or Type-II** attack), the injected script is not stored on the server; instead, it is *reflected* off the target server. A victim may for instance receive an email with a tainted link that contains malicious URL parameters.

In the example the tainted URL contains JavaScript code as query. An unsuspecting user (the victim) may receive this URL in an email and trust it, because she trusts http://myforum.nl. The malicious code is reflected off the server and ends up in the victim's browser, which executes it.

```
http://myforum.nl/search?q=Let+me+…

http://myforum.nl/search?q=<script>…
```

How to avoid it

As before, **validation** of user input is vital. Importantly, input validation has to occur on the client **and** server-side. A malicious user can bypass client-side validation by using a tool like curl to create HTTP requests instead of using the browser or by manipulating the requests inside the browser. Thus, client-side validation only helps actual users of the application, it offers no defense against an XSS attack.

A server that generates output based on user data should **escape** it (e.g., escaping <script> leads to &lt;script&gt;), so that the browser does not execute it. DOMPurify is a good tool for sanitizing HTML code and therefore you can use it to protect your web application from XSS attacks.

Here is a code example of DOMPurify:

```
let dirty = '<script> alert("I am dangerous"); </script> Hi';
let clean = DOMPurify.sanitize(dirty);
```

After execution, the variable clean will contain only **"Hi"**, DOMPurify will remove the <script> tag to prevent an XSS attack.

We have shown here only a small sample of input that is usually employed to evaluate an application's XSS defenses. An elaborate list of inputs that need to be included in any such testing can be found at OWASP.

Improper input validation

When user input is not checked or incorrectly validated, the web application may start behaving in unexpected ways. An attacker can craft an input that alters the application's control flow, cause it to crash, or even execute arbitrary user-provided code.

Improper input validation is often the root cause of other vulnerabilities, e.g., injection and XSS attacks.

How to avoid it

As stated already a few times, input validation on the client **and** server-side is vital to avoid these vulnerabilities.

Security misconfiguration

Web application engineering requires extensive knowledge of system administration and the entire web development stack. Security vulnerabilities can arise in all aspects of the application: the web server, the database, the application framework, the server's operating system, etc. Common issues are the following:

- Default accounts and passwords remain set.
- Resources may be publicly accessible that should not be.
- The root user can log in via SSH and thus allowing remote access to privileged accounts.
- Security patches are not applied on time.
- Error handling is not done properly causing the application to crash.

How to avoid it

Install the latest stable version of Node.js and Express. Install security updates. Rely on npm audit (and then npm audit fix) to assess and fix your dependencies' security issues.

RetireJS is another tool to detect outdated JavaScript libraries. It does not fix anything but lists vulnerabilities with greater detail than npm audit.

A popular package to secure Express-based applications is Helmet. It acts as middleware in Express applications and sets HTTP headers according to best security practices.

Rely on automated scanner tools to check web servers for the most common types of security misconfigurations.

Sensitive data exposure

If a web application does not use HTTPS for all authenticated routes (recall that HTTPS is needed to protect the session cookie), a malicious user can monitor the network traffic and steal the user's session cookie.

If a web application relies on outdated encryption strategies to secure sensitive data, it is just a matter of time until the encryption is broken.

In addition, if sensitive documents can be accessed without authorization a malicious user can run scans of likely end points that are not secured.

How to avoid it

- All sensitive data should be encrypted across the network and when stored.
- Only store necessary sensitive data and discard it as soon as possible (e.g., credit card numbers, backups).
- Use strong encryption algorithms (a constantly changing target).
- Disable autocompletion of HTML forms collecting sensitive data.
- Disable caching for pages containing sensitive data.

Broken access controls

A malicious user, who is authorized to access a web application (e.g., a student accessing Brightspace), changes the URL (or URL parameters) to a more privileged function (e.g., from student to grader). If access is granted, **insufficient function level access control** is the culprit.

Web applications often make use of ***Direct Object References*** when generating a HTTP response.

Consider a user who accesses her wishlist using the following URL http://mywishlist.nl/wishlist?id=234. Nothing stops the user from also trying, e.g., http://mywishlist.nl/wishlist?id=2425353

If the id values are **insecure direct object references**, the user can view other users' wishlists in this manner. If we assume that the wishlists are of the same type of users (i.e. "customer" instead of "admin") you are accessing other accounts having the same privilege level, we are here dealing with a ***horizontal privilege escalation***.

How to avoid it

- Avoid the use of direct object references (indirect is better).
- Avoid exposing object IDs, keys and filenames to users.
- All routes should include an authorization subroutine. (i.e. /* middleware checks for credentials)

Cross-Site Request Forgery (CSRF)

In the words of OWASP: *"An attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds."*

Here is an example scenario: imagine a web application that allows users to transfer funds from their account to another account via a URL such as http://mygame.nl/transferFunds?amount=100&to=342432

the URL contains the amount and which account to send the funds to. **The victim is already authenticated**. An attacker constructs a request to transfer funds to her own account and embeds it in an image request stored on a site under her control:

```
<img    src="http://mygame.nl/transferFunds?amount=1000&to=666"    width="0"
height="0" />
```

If the victim accesses the website that is under the attacker's control (e.g., because the attacker sent the victim an enticing email to access the URL), the browser downloads the HTML, parses it and starts rendering. It will automatically download the image without checking whether the src is actually an image. The transfer of funds will then take place if the web application the user is authenticated to does not defend against a CSRF attack.

How to avoid it

The main reason why this attack is successful in our example is due to the server *trusting* that the request was made with this intention by the victim - the request has the correct session ID and the server responds to the request.

How can the server validate that this request was intentionally sent by the user? The most common approach today is via a so-called ***CSRF token***, a randomly generated string, generated by the server, that is unpredictable and cannot be guessed. The server inserts this CSRF token in the response, typically in a hidden form field (**not** in a cookie). The server also keeps track of the combination of session ID and CSRF token. When the user then fills in the form and submits it to the server, the CSRF token is returned to the server and the server checks whether it matches the one it has on record. If it is a match, the server executes the requested action (e.g., changing a user's profile data) and rejects it otherwise. An attacker is not able to guess this CSRF token - as long as we do not rely on cookies to store CSRF tokens, as cookies are appended to the HTTP request automatically by the browser. Although the attacker can try to guess a valid CSRF token, it should be impossible if the server generates CSRF tokens according to best practices.

As CSRF tokens are an established line of defense, Express middleware exists (a popular option is csurf) that takes care of the generation and validation of CSRF tokens.

Another option for a web application to verify that a user intended a particular request are the use of **reauthentication** (the user is asked to authenticate again, e.g., if the request takes place at an unusual time, or at an unusual location) and **CAPTCHA mechanisms**. (it was a human, not a script).

Insecure components

Vulnerabilities of software libraries and frameworks are continuously being discovered and patched. An application that is not patched when a vulnerability becomes known is a candidate for exploitation. It is important to be aware of the dependencies of one's Node.js application and install security patches quickly when they become available.

Not only the application itself needs to be kept up-to-date, the server's operating system also needs to be continuously patched, as well as any other software used to support the web server (e.g., Redis, MongoDB, Nginx).

How to avoid it

- Always use latest versions of libraries.
- Keep a look out for potential vulnerabilities in open source software, and patch them as soon as possible.
- Use available tools such as npm audit, especially when relying on many third-party packages. It is important to realize what security issues those packages have.

Unvalidated redirects

Let's cite OWASP one last time: *"An attacker links to an unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site."*

Here is an example scenario: imagine a web application that includes a route called redirect. That route takes a URL as parameter and redirects to that URL. Once an attacker finds a route that enables such redirects, the attacker creates a malicious URL that redirects users to her own site for phishing, example:

```
http://www.mygame.nl/redirect?url=www.malicious-url.com
```

How to avoid it

This attack can be avoided by disallowing redirects and forwards in a web application. When redirects have to be included, users should not be allowed to redirect via URL parameters. The user-provided redirects need to be validated.

Summary

Overall, as we have just seen, web applications offer many angles of attack. Securing a web application requires extensive knowledge in different areas. When securing a web application, start with defending against the most frequent vulnerabilities. Use well-known auditing tools to fix issues. Keep your application up to date as much as possible. Look into tools such as Dependabot that create automatic pull requests to keep your application's dependencies up to date. Keep in mind to sanitize and validate.

Self-check

- You write a web application that contains a session management component. Clients who want to use your application have to authenticate and subsequently a session ID is used to guarantee them access to your application. Once they log out of your application, two steps should be implemented: (1) the session ID is to be deleted from the client and (2) the session ID is to be set to expired on the server. Which attack is enabled if the implementation of step (2) is forgotten?
    - CSRF.
- You have developed a Web application with multiple user accounts. Which threat is your Web application susceptible to if you can manipulate the URL of an account page to access other account pages?
    - Insecure Direct Object References.
- You develop a Web application that contains a session management component. You decide to use as session ID the concatenation of the current POSIX time (i.e. the number of seconds elapsed since 00:00:00 UTC January 1, 1970) and a counter starting at 100 that is incremented by 1 each

time a new user requests a session at the same POSIX timestamp. Which main vulnerability arises in this scenario?

  - Session hijacking.
- What approaches are most likely to secure an application against cross-site request forgery?
  - Reauthentication and CAPTCHA mechanisms.
- A session-based system authenticates a user to a Web application to provide access to one or more restricted resources. To increase security for the end user, an authentication token should
  - .. be stored in a non-persistent cookie.
- What does an attacker require to succeed in a reflected XSS attack?
- A vulnerable web service that takes parts of the URLs it receives and generates output without validating the received data.
- One common defense against CSRF attacks is the use of CSRF tokens. What is the idea behind this defense?
- Create a secret identifier (the CSRF token) on the server, share it with the client in a hidden form field and require the client to return it when making a request. The server only handles the request if the provided token is the same as the one it sent to the client earlier.
- Which type of attack occurs when an attacker convinces a user to send a request to a server with malicious input and the server echoes the input back to client?
  - Non-persistent XSS attack.
- Sanitizing user input protects from which of the following attacks?
  - Cross-site scripting.
- Which attack type does this scenario describe: An attacker can browse through other users' Facebook timelines by URL manipulation.
  - Broken access controls.