Student's Guide to CMPUT 404
Web Applications and Architecture

A student's guide to the complex nature of the internet: its technologies, protocols, and implementations

Introduction to modern web architecture, from user-facing applications to machine-facing web-services. Topics include: the evolution of the Internet, relevant technologies and protocols, the architecture of modern web-based information systems, web data exchange and serialization, and service-oriented middleware.

By slaying his assignment minions, defeating his midterm champion(s) and ending the final themselves. The heroic legacy of your achievement will be remembered on your academic transcript.

Disclaimer: This guide is not responsible for the unintentional consequences of GETing the plane of Pandemonium, DELETEing the plane of Elysium, POSTing evil creatures in Mount Celestia, and PUTing Tiamat in the Beastlands.

Contents

Chapter 1: Internet and Web

Web

The web is a "tool" that we use to request, search, navigate and share information. We can use this web to access and operate software. This *tool* is really a collection of interconnected computers that is spread throughout the entire world and even in space.

You can use Firefox to inspect a website and see the source code that makes up the website. If you want to look at traffic you can use Wireshark to see the individual ethernet frame and protocol.

The web is *not* the internet. The internet is a network of computers like desktops, laptops, phones, TV, etc. While the World Wide Web is a network of pages like Wiki pages, new stories, blog post, etc. The web is a connection of software or virtual pages, while the internet is a connection of hardware. This also means that the web and internet have different domains of connections. The internet is connected using physical means like ethernet, wifi, cable, satellites, etc. While the WWW uses protocols and web formats to connect between different websites or pages. Examples include hyperlinks, HTML, images, redirects, etc. These are more software orientated connections.

Primary Components

You have web servers that serve web content and web clients like web browsers like Chrome or Firefox that connects to web servers. The connection is done via HTTP (HyperText Transport Protocol) and HTML (HyperText Markup Language). You can also use Hypertext which are text that has links in it to take you somewhere else.

Definition 1.0.1 (Hypertext and Hypermedia)**.** *By now the word "hypertext" has become generally accepted for branching and responding text, but the corresponding word "hypermedia", meaning complexes of branching and responding graphics, movies and sound - as well as text – is much less used. Instead they use the strange term "interactive multimedia": this is four syllables longer, and does not express the idea of extending hypertext.*

Nelson, Literary Machines, 1992

The internet is made up with multiple layers of protocols and mediums that allows communications between computers.

Link Layer

In computer networking, the link layer is the lowest layer in the Internet protocol suite, the networking architecture of the Internet. The link layer is the group of methods and communications protocols confined to the link that a host is physically connected to. The link is the physical and logical network component used to interconnect hosts or nodes in the network and a link protocol is a suite of methods and standards that operate only between adjacent network nodes of a network segment.

From: https://en.wikipedia.org/wiki/Link_layer

Ethernet

On a wired connection all information will need to be cut up into an *Ethernet* frame.

Ethernet Header Sizes

Schema	Size in bytes
Preamble	7
Start Frame Delimiter	1
Destination	6
Source	6
Length	2
Payload	46-1500
CRC	4

The preamble will always be 0xAA because 0xAA is encoded as 10101010 in 8-bit binary. This will set the sample rate for the hardware to listen, spool up and grab the information. The Start Frame Delimiter or SFD denotes the start of the Ethernet frame, it is encoded as 0xAB or 10101011, the double 1 at the end will tell the hardware to process the next 64-1518 bytes. The **Source** and **Destination** sections are MAC addresses that are mapped to hardware and tells who is the recipient and sender. **Length** is the size of the payload in byte units. **Payload** contains the actual data itself. The standard assumes a MTU or Maximum Transfer Unit of 1500 bytes, and the **CRC** is used as a checksum and error detection.

Note

This is a pretty old standard and in today's world we might use something faster or bigger in terms of payload size.

The standard is designed so that everyone knows the minimum packet size that can be sent. Note that there are potential wastes in a transmission. Also note that sizes that aren't fragmented or split will lead to minimal latency from the lack of overhead caused by splitting packets.

The idea is to keep the message sizes smaller than 1.5kb to ensure that you stay inside within packets, but not too small so that you send too many headers. Most people have MTUs at 1500 or less.

Internet Layer

The internet layer is a group of internetworking methods, protocols, and specifications in the Internet protocol suite that are used to transport network packets from the originating host across network boundaries; if necessary, to the destination host specified by an IP address. The internet layer derives its name from its function facilitating internetworking, which is the concept of connecting multiple networks with each other through gateways.

From: https://en.wikipedia.org/wiki/Internet_layer

IPV4

This is a routing standard that allows the Ethernet frame/data to be routed. This is because Ethernet is not routable. IPV4 is the backboard of the internet but we are running out of IPV4 addresses. It was designed to communicate over large distances and to many computers, in fact it was a compromise to address computers instead of using MAC addresses. Note IPV4 is stateless as well. The IPV4 header is wrapped into the Ethernet Frame.

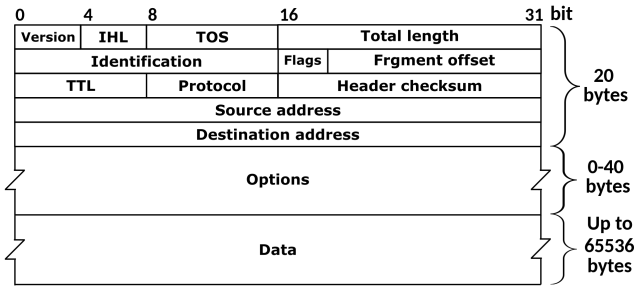

Figure 1.1: IPV4 header
https://en.wikipedia.org/wiki/IPv4
#/media/File:IPv4_Packet-en.svg

IPV6

IPV6 is like IPV4 but with more addresses. The number of addresses you can represent using IPV6 is 2^{128} while IPV4 can only represent around 4.2 billion values 2^{32}. TCP is encapsulated within the IPV6 layer. An address can be really big, Examples

- 2001:0bd8:0000:0000:0000:0000:0000:00001
- 3132:0:0:0:0:0:0:1
- 2001:bd8::1

For ports the IPV6 addresses are encompassed within square brackets. Example: http://[2001:29b::1]:443/

Note

The IPV6 header is much simpler compared to the IPV4 header as the designers of the IPV6 header wanted to keep the header simple and have a different protocol handle everything else. For integrity protection that task is handled by the link layer e.g., Ethernet, and the transport layer, mainly TCP or UDP.

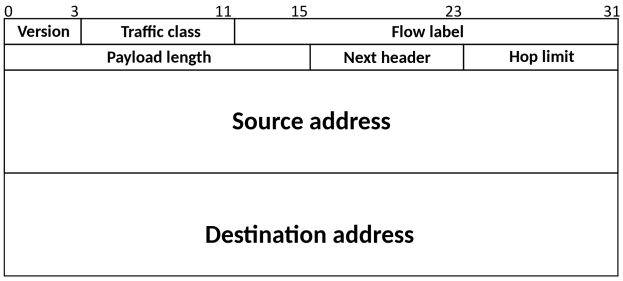

Figure 1.2: IPV6 Header
https://en.wikipedia.org/wiki/IPv6
#/media/File:IPv6_header-en.svg

Transport Layer

In computer networking, the transport layer is a conceptual division of methods in the layered architecture of protocols in the network stack in the Internet protocol suite and the OSI model. The protocols of this layer provide host-to-host communication services for applications. It provides services such as connection-oriented communication, reliability, flow control, and multiplexing.

From: https://en.wikipedia.org/wiki/Transport_layer

UDP

Stands for **User Datagram Protocol**, the user means that user-space applications can use it. Like IPV4 it's stateless, and only contains source and destination port numbers as well as a checksum to provide some integrity. This type of connection is lossy and not ordered, meaning deterministic behavior is not guaranteed. There is also no connections like TCP. This is useful for only sending a small amount of data across a network where latency is more important than integrity. UDP is encompassed within IP, the data comes after it. It will be the IP data size minus the UDP header size. Checksum is not required but includes data, UDP header and IP header. DNS or NTP uses UDP because they are simple query-responses that can fit within a UDP message packet, of $2^{16} = 65536$ bytes.

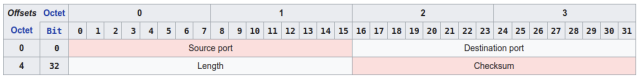

Figure 1.3: UDP Header
https://en.wikipedia.org/wiki/User_Datagram_Protocol

TCP

Stands for Transmission Control Protocol. Unlike UDP there is a connection between a socket that goes

through different states. Some of these states are part of the 3-packet handshake that an opening connection makes before sending data. This 3-packet handshake involves

1. The source socket will send a SYN to the destination socket who is listening

2. That destination socket will reply with SYN+ACK to tell the incoming connection "I've heard you SYN"

3. Finally the source socket will reply the destination socket's SYN+ACK with ACK to say "I've heard your SYN+ACK let's communicate"

There are two ways a TCP connection can close:

Active Close

This is initiated by the client and will send a CLOSE/FIN signal to the server, the server will respond with ACK in which the client will respond with FIN/ACK. The client could also close the connection and send the FIN+ACK/ACK at the same time to the server and ignore the response from the server.

Passive Close

In passive close the server will disconnect the connection with a FIN/ACK, and have the client respond with CLOSE/FIN with the server responding with ACK after CLOSE/FIN.

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)																															
⋮	⋮																																
60	480																																

Figure 1.4: TCP Header
https://en.wikipedia.org/wiki/Transmission_Control_Protocol

Application Layer

An application layer is an abstraction layer that specifies the shared communications protocols and interface methods used by hosts in a communications network. An application layer abstraction is specified in both the Internet Protocol Suite (TCP/IP) and the OSI model. Although both models use the same term for their respective highest-level layer, the detailed definitions and purposes are different
From: https://en.wikipedia.org/wiki/Application_layer

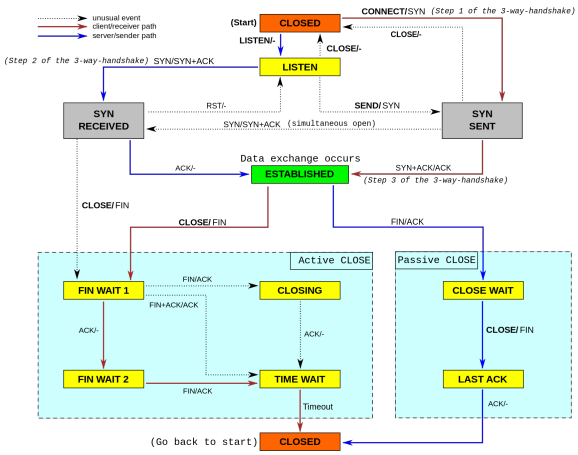

Figure 1.5: TCP State Diagram
https://en.wikipedia.org/wiki/Transmission_Control_Protocol#/media/File:Tcp_state_diagram_fixed_new.svg

DNS

Domain Name Service is part of the application layer which goes on top of UDP or TCP. This protocol allows us the bind a name to another name, IP, or set of IPs.

Type	What it represents
A	A record that point to an IPV4 address
AAAA	A record that points to an IPV6 address
CNAME	A record that points to another name

You can use programs like `host`, `dig`, or `nslookup` to check names. Because a `CNAME` record points to another name and that name could be another `CNAME` record this DNS behaviour can be recursive until it reaches an IP address or a set of addresses.

Firewalls

Usually prevents hosts from communicating on certain ports, or hosting services like port ssh, ftp, etc. HTTP and firewalls means that web clients are unlikely to be web servers as well. That communication must be initiated by clients rather then web services.

Internet Protocol Stack

Exchanging data over the internet uses multiple different standard and protocols that are built upon each other.

Layer	Examples
Application	DNS, HTTP(S), SSL, SSH, etc
Transport	TCP, UDP, etc
Internet	IPV4/6, ICMP, etc
Link	Ethernet, Wi-Fi, DSL, ARP, etc

Like layers of an onion each layer is wrapped inside each other. The Application layer is wrapped inside the Transport layer which is wrapped inside an Internet layer which is finally wrapped inside a Link layer. Which layers are used depends on the application, context, and physical medium used to communicate on the internet.

Note

For HTTPS the TLS layers goes in between the TCP and HTTP layer. There is an additional handshake that is done so that both the client and server can encrypt and decrypt the data. This add overhead as well as not encrypt anything below the transport layer. Meaning a sniffer can read the transport, internet and link layers. Basically they can see who you are talking to.

Review

Ethernet.

- The lowest protocol on the network stack, everything is encompassed within it. It's one example of the Link Layer.
- It's old but we still use it.
- Payload size is between 46 and 1500. Total max size including the header minus the preamble and delimiter is 1518.

IPV4.

- The second lowest protocol on the network stack, encompassed within the Link Layer, it handles the routing and addressing between one computer and another. It's one example of the Internet Layer.
- We are running out of them due to the number of interconnected devices
- Header is pretty complicated with options and stuff.
- Max payload size is $2^{16} = 65536$ bytes including all the headers and payloads above it or anything being encapsulated by it. But not including the IPV4 header itself.

IPV6.

- Like IPV4 it's the second lowest protocol on the network stack, encompassed within the Link Layer, it handles the routing and addressing between one computer and another. It's one example of the Internet Layer.
- We can more of these 2^{128} a number with 39 digit or a big fucking number.
- Header is a lot simpler as the task like checksuming is handled off to other layers like Link or Transport.
- Max payload size is $2^{16} = 65536$ bytes including all the headers and payloads above it or anything being encapsulated by it. But including the IPV6 header itself.

UDP.

- Simple, connectionless protocol for the Transport Layer or the third layer in the network stack.
- Used in DNS, and NTP queries for something simple or small.
- Message packets are not guaranteed to be in order.
- Max payload size is $2^{16} = 65536$ bytes including everything the UDP packet/message is encapsulating. But not including the 8 bytes for the header.

TCP.

- A more complicated, connection (or state) based protocol that is part of the Transport Layer like UDP.
- Has a three way handshake to establish communication, using SYN, SYN+ACK, ACK.
- Active Close is client driven
- Passive Close is server driven

DNS.

- Domain Name Service, part of the Application Layer or the last (top) layer in the network stack.
- Used to resolve domain names to IP addresses
- Made up wih different records, A for IPV4, AAAA for IPV6, CNAME which points to another name

Chapter 2: HTTP

HyperText Transport Protocol, this protocol beat FTP and Gopher because it was more flexible, extensible, and not burden with licensing agreements. It sends content not files which is useful if you are just searching for things and don't want to download an entire file or a small subset of that file. Can respond to requests based on reserved VERB words like GET, POST, DELETE, PUT, etc The flexibility comes from the fact custom headers could be included allowing for extension and adding new features. The protocol allowed for a more request or command oriented pattern for communication. HTTP relied on the pairing of web clients and web servers as well as using URIs to describe resources, allowing more than 1 resources to be hosted on 1 server. The standard for HTTP is outlined by RFC2616.

Generally HTTP uses TCP, and on port 80. For HTTPS to allow encrypted HTTP traffic using TLS it uses port 443 (generally). HTTP can work on both IPV4 and 6 addresses. HTTP requests are made to addresses called **URIs**.

HTTP Command

Every HTTP command is made to a URI

Command	What it does
GET	Retrieve information from that URI
POST	Run search, log-in, append data, change data
HEAD	GET without a message body (for caching)
PUT	Store the entity at that URI
DELETE	Delete the resource at that URI
PATCH	Modify the entity at that URI
OPTIONS	What options a resource can accommodate
TRACE	Debugging or echo request
CONNECT	Tunnelling proxy over HTTP

PATCH is not defined in RFC2616 those are in a different RFC standard.

URI

A URI is an Universal Resource Identifier, it identities i.e., points to a resource. Most URIs are URLs or Uniform Resource Locator but not all URLs are URIs. URLs tells you how to get to a resource. Some URIs are URNs or Uniform Resource Name, they tell you the unique name or number given to a resource by some body like IETF

URL

As stated before URLs or Uniform Resource Locator tells you how to get to a resource. It is comprised with two parts:

- **Scheme** e.x., - http, https, mailto, file, data, ftp, gopher, irc, spotify
- **Everything else** - Anything not part of the scheme.

URL syntax

All URLs follow a fixed syntax

```
URI =
scheme:[//authority]path[?query][#fragment]
```

The authority follows another syntax listed below

```
authority = [userinfo@]host[:port]
```

Syntax	Example
scheme	http, https
authority	Can be a username and password as part of userinfo, Host followed by a port after :
path	The location of the resource
?	A query string to query the resource
#	Fragments that will provide directions to a secondary resource such as a section heading within the resource

Absolute and Relative URLs

Both the authority and path can be absolute or relative, if the authority is missing then the authority is implied based on the current URL. If the path has any `..` or is relative then it locates the resource based on the "current working director" of the current URL. Examples:

- Absolute authority, absolute path - http://[::1]:8000/images/web-server.svg
- Absolute authority, relative path - http://[::1]:8000/images/../index.html
- Implied authority, absolute path - /images/web-server.svg
- Implied authority, relative path - image/web-server.svg

Queries

Queries are an optional portion that can be added to a URL. You can have one or more arguments in a key-value pair format like so `key=value&key2=value2`. Other separator exist like `;` instead of `&` or just having a string and no key-value pairs. The syntax is quite loose and it's highly dependent on the webserver.

Fragments

Fragments are an optional portion that can be added to a URL. They allow jumps to some spot in the content like a time in a video using `#t=` in a YouTube link, or a part of a page like a section's name.

Encodings

Universal URIs have to be able to handle anything, including characters like spaces, accents, punctuations, emoji, etc. For HTTP assume our URLs are Unicode UTF-8 encoded. For characters that aren't in `-._ 0-9a-zA-Z` we use `%` encoding. Uses RFC 3986, space is encoded as `%20`. For domina names we use "punycode" encoding where emojis and other special characters are converted to visible ASCII.

HTTP 2

Released in 2015, it is based on the SPDY protocol by Google and it was designed to reduce latency. It only supported over TLS so only on HTTPS connections and is backwards compatible with HTTP version 1's methods, including their status codes, headers, and URIs. Instead of encoding everything as ASCII, the entire protocol communicates in binary to reduce bandwidth. Features that make it faster, and more responsive (lower latency) is:

- **Pipelining** - Allows multiples requests to be handled at the same time instead of waiting for a response before continuing.
- **Push** - Allow the servers to look into the future and sends clients content before they even request it
- **Multiplexing** - Allows different types of requests to be send back to the client at the same time via interleaving.

This avoids using multiple TCP connection to a single server thus reducing the number of high-latency TCP and TLS handshakes. Both request and response is happening at the same time and data is interleaved. Requests and responses can be prioritized.

HTTP Request

To send a message to a **server** *from* the **client** one would need to send a HTTP Request message.

Definition 2.0.1 (HTTP Request). *This is the grammar for an HTTP request.*

```
Request = Request-Line
          *(( general-header
           | request-header
           | entity-header )"\r\n")
          "\r\n"
          [ message-body ]
```

Where:

```
Request-Line = Method URI HTTP-Version"\r\n"
```

The request line always come first where the `Method` is the type of request, the `URI` is the "URL" and the `HTTP-Version` is what version of HTTP you are running. For format for `HTTP-Version` is `HTTP/X.Y` where `X` is the major and `Y` is the minor version numbers. For this project it's entire in HTTP1.1 so `HTTP-Version` should be `HTTP/1.1`.

There must be an empty line with a `CRLF` or `\r\n` right after the header to separate the header and the body of the message.

HTTP Response

To receive a message from the server or responding to a request from the server-side one would need to send a HTTP Response message.

Definition 2.0.2 (HTTP Response). *This is the grammar for an HTTP response.*

```
Response = Status-Line
           *(( general-header
            | response-header
            | entity-header )"\r\n")
           "\r\n"
           [ message-body ]
```

Where:

```
Status-Line = HTTP-Version Status-Code
Reason-Phrase"\r\n"
```

Like in HTTP Request the status line comes first where the `Status-Code` is the response code and the `Reason-Phrase` is the phrase associated with the `Status-Code`. The format is for the HTTP-Version is the same as in HTTP Request.

HTTP Headers

There are a lot of different headers that can make up a HTTP message but for all headers they follow a specific syntax

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

It's encoded like a key value pair where `field-name` is the header's name and the `field-value` is the value for that header.

GET

HTTP GET is a simple request to be sent that resource.

- It might be dynamic (resource is generated when the request is received by the software on the server), this could be done by a JavaScript or a Python CGI.
- It might be static (just a file sitting on the server)
- It might be a mix of static and dynamic content.

We can send query parameters along with a HTTP GET in the URL using the `?` character. GET request **SHOULD NOT** cause the server to change data-we should use a different HTTP method for that. GET request are only used for reading data/information.

Note

You **SHOULD NOT** send a body with a GET Request because it might lead to a rejection. The behaviour is undefined so to keep everything working smoothly for all platforms you shouldn't send one.

Request Headers

With HTTP 1.1 you **must** send a `Host` header with the authority you are connecting with as the value. You **should** also send these headers as well

- Connection: Keeps the connection open or not after the current transaction finishes.
- Date: The date of the request.
- Accept: Advertises which content type, expressed as MIME types the client is able to understand.

POST

HTTP POST is a request to update, create, or generally interact with a URL.

- Can do things like queries like HTTP GET but not limited by length.
- Use to submit HTML forms.
- POST is expected to add or mutate data.

Moreover POST:

- URI identities a service/handler/script/process
- Arguments are stored in the HTTP request body
- The request body is interpreted by some software and processed
- "Send this here for processing" basically

POST can be used for:

- Login/logout
- Reply
- Post on a forum/blog
- Upload multiple files (Somewhere)
- Make an order
- Fill out a survey/poll

Forms

POST methods often send their parameters in the body of the POST requests. The `Content-Type` of the body is usually some type of form

- `applicaiton/x-www-form-urlencoded`
- `multipart/form-data` - Generally used when uploading files

Each form element has a `name`. The `name` is the identifier used for the variable that is going to store the content. Example: `<input name="occupation">` becomes `occupation=` The content of the input element becomes the value.

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

field1=value1&field2=value2
```

Figure 2.1: Example `x-www-form-urlencoded` https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

multipart/form-data uses mime (multipurpose internet mail extensions) to send form data. Mostly used to upload files as binary, but it can be used for any form. Send the content-size first and then ask the server i that's okay. The server must response with a `100 Continue` code if it can handle that size of data. Then the client will send the body. This interaction is a bit slower because it adds a round trip latency for confirmation with the header before sending the body. But the confirmation is done to ensure that the client doesn't send a file that is too big for the server to handle or of a type it can't handle, etc.

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: multipart/form-data;boundary="boundary"

--boundary
Content-Disposition: form-data; name="field1"

value1
--boundary
Content-Disposition: form-data; name="field2"; filename="example.txt"

value2
--boundary--
```

Figure 2.2: Example `mutlipart/form-data` https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

The boundary lets the server tell when one part of the form ends and another begins. The random number should be chosen so that it won't show up in the content. The last boundary line **must** end with `--` to tell the server that all of the form has been sent.

Request Headers

With HTTP 1.1 you **must** send these headers

- Host: Specifies the host and port to which the request is being sent to.
- Content-Type: The type of the form or what type of the body.
- Content-Length: The size of the body.

You **should** also send these headers as well

- Connection: Keeps the connection open or not after the current transaction finishes.
- Date: The date of the request.
- Accept: Advertises which content type, expressed as MIME types the client is able to understand.

PUT

HTTP PUT is like HTTP POST except the URI does not handle the request, it is the request. It creates a new resource or replaces a representation of the target resource with the request payload.

> The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request – the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource.
>
> Fielding, et al. ,RFC2616: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.6

The difference between `PUT` and `POST` is that `PUT` is idempotent; calling it once or several times successively has the same effect i.e., that is no side effect, where as successive identical `POST` requests may have additional effects, akin to placing an order several times.

Moreover PUT:

- URI identifies an entity (file, db entry . . .).
- Arguments are stored in the URI query string or HTTP headers.
- request body contains the entire entity, it essentially overwrites or creates a new entity with that content.

PUT can be used for:

- Create a new entity at the URI
- Replace an existing entity at the URI
- Add/replace an entry to a DB
- Entity can be retrieved later with GET and the same URI

DELETE

Like HTTP POST except the URI does not handle the request, it is the request, a request to delete the entity at that URI.

> The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method MAY be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully. However, the server SHOULD NOT indicate success unless, at the time the response is given, it intends to delete the resource or move it to an inaccessible location. A successful response SHOULD be 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has not yet been enacted, or 204 (No Content) if the action has been enacted but the response does not include an entity.
>
> Fielding, et al. ,RFC2616: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.7

Moreover DELETE:

- URI identities an entity (file, db entry. . .).
- Arguments are stored in the URI query string or HTTP headers
- Request body is usually empty
- Response body is usually empty
- Basically delete this entity, if I GET it again it should response with a 404 error if I didn't PUT or POST at that same URI.

DELETE can be used for:

- Delete the entity at the URI
- Delete a file on the server's file system
- Remove an entry in a DB
- Entity cannot be retrieved later with GET and the same URI

WebDAV

WebDAV is an extension of HTTP that allows clients to perform remote Web content authoring operations. Basically FTP but for the web

- Let's you create and upload to a URI using HTTP PUT
- Download from a URI using HTTP GET
- Delete an entity at a URI and the URI using HTTP DELETE
- Make directories and folders using a new HTTP command MKCOL for MaKeCOLlection

User Agent

A user agent is the client or browser that is accessing the web services or server. It is a HTTP header that is part of the HTTP request that is sent to the server. `User-Agent` is the header's name.

> **Note**
> Not all HTTPs are browser, cURL is not a web browser but it is a web client. Therefore it will provide a user agent to the server.

The user agent will tell a lot about the client, for web browsers they might show

- Browser i.e., Chrome, Firefox, etc
- Version
- Operating system
- Machine architecture
- Rendering engine
- etc

Mobile version will also have their own separate user-agent, even the Nintendo Switch will have their own user agent.

HTTP Status Codes

There are five different sets of status codes that can be sent as a response.

Code Range	Status Type
`1XX`	Information codes
`2XX`	Success codes
`3XX`	Redirection codes
`4XX`	Client Error codes
`5XX`	Server Error code

1XX: Information codes

These codes are use when they are negotiating a transaction. There are many but in this we are going to look at 2.

100 Continue

Used in multipart and uploads, specifically tells the client to send the request body/data. The server has the choice to accept the request or not and with the 100 code it has decided to accept it

101 Switching Protocols

This is a rare status code where it provides `Upgrade` as a header. This specify the new protocol to change to after the blank line at the end of the header. An example situation could be switching from HTTP/1.1 unencrypted to HTTP/2.0 but doesn't really happen as HTTP/2.0 only works on encrypted connections for browsers.

2XX: Success codes

These are success codes, meaning that the transactions between server and client was a success.

200 OK

The request was a success depending on the client's request.

- GET: sends an entity for the requested URI
- HEAD: sends headers for the entity for the requested URI
- POST: sends an entity describing the result of the POST
- TRACE: sends back the entity it received

201 Created

Request succeeded and a new entity was created and exists (e.g. PUT)

202 Accepted

Like `200 OK` but the server's not done with it yet. For example, you asked the server to perform a calculation and its working on it.

203 Non-Authoritative Information

This is a rare code used with proxies.

204 No Content

Request succeeded but the server is only sending headers and no entity or response body.

205 Reset Content

Rare status code, like `204 No Content` but the browser should clear the form/page.

206 Partial Content

Used to resume downloads. The client request with a `Range` header to continue a big download that was interrupted. The server responds with a `206 Partial Content` code and a header named `Content-Range` is returned indicating what part it's sending.

3XX: Redirection Codes

Redirects is a very powerful for HTTP. It is a cheap form of abstraction, provides work-around for browser/protocol issues, redirects HTTP to HTTPS, redirect request from old URIs to new URIs. It provides load balancing, separate dynamic and static content onto two different servers, allows more than one server behind the scenes. It allows URLs to be shorted and allows websites to be reorganized but still allow old URIs to keep working (fighting link-rot).

300 Multiple Choices

This is a rare response that provides a list of choices and the user or the browser chooses one. An example could be selecting a language for a page.

301 Moved Permanently

Go to the URI mentioned in the `Location` header, and don't ask again. The new location should be cached and the URI in the location bar automatically changes.

302 Found

This is a temporary redirect and the client should GET the URI mentioned int he `Location` header and display that response instead. This is useful for load-balancing.

> **Note**
> The URI in the location bar stays the same, the user doesn't know that it is in a different location.

303 See Other

Exists to solve the form-POSTing problem where POSTing the same post will cause duplicate post in the forum. GET the URI in the `Location` header and don't save the redirect in your cache, you can keep making POSTs to the URI that gave you a 303 code. The URI in the location bar changes.

304 Not Modified

The browser can make a conditional GET request for URIs that it has cached, asking the server to send the entity only if it's changed since the time it was cached or a specific version. There are two headers for this response:

- `If-None-Match` - Followed by a list of etags (like git tags) will get a 200 OK only if it has a new version not in the list

- `If-Modified-Since` - Followed by a date and time, will only get a 200 OK only if it has a newer version since that time.

The response will have no body or entity.

305 Use Proxy

Rare response, try the proxy server specified by the `Location` header.

307 Temporary Redirect

Go to the URI mentioned in the `Location` header. Because it is temporary the client can keep making requests to the URI you originally requested in case the sever needs to redirect you somewhere else next time. Cache the redirection using standard caching headers and rules. The URI in the location bar is updated.

4XX: Client Error codes

> It's your fault!!!!

Basically anything that is caused by even if you didn't know that that server couldn't do or support that.

400 Bad Request

The server can't read your garbage. Don't send it again. Basically your request is malformed.

401 Unauthorized

You have to send authentication information to see this URI. Header and entity(body) explains to the browser and user how to log in. Mostly useful for HTTP `Authorization:` header authentication.

402 Payment Required

Pay before moving on. Supposedly reserved but some services use it anyway like MobileMe, Google APIs use it. Youtube will use it to force you to solve a CAPTCHA.

403 Forbidden

The web server will never respond to this request, no matter who you log in as. Maybe it could answer your request buy an administrator disabled that ability.

404 Not Found

Self Explanatory. You've got the wrong resource or path, can't find the thing.

405 Method not allowed

Whatever HTTP method you used doesn't work on this URI.

406 Not Acceptable

The server cannot respond in way that matches your request's `Accept` header line. You ask for a type that the server cannot handle, and it's your fault for not knowing.

407 Proxy Authenticating Required

We're not going to proxy your request till you authenticate.

408 Request time out

You took too long to send your request, we're not going to service you. Try again but faster next time. This is to prevent a Slowloris attack where the attacker will take it sweet ass time trying to take resources away from other users.

409 Conflict

The request is in conflict. Often used with `PUT` requests. Basically there is a concurrency issue at the URI.

410 Gone

Yeah it was here, but it ain't coming back. Don't even try again. Meaning this request should be cached.

411 Length Required

I can't service a request like `POST` without a `Content-Length:` header.

412 Precondition Failed

Header information wouldn't be what you wanted it to be so I won't process the request. An example could be modified too recently, so don't allow PUT to succeed to modify it again.

413 Request Entity Too Large

Sending an entity (POST, PUT, . . .) that's bigger than the server can handle. The client should know better.

414 Request-URI Too Long

The web server cannot process the URI because it's too long and will overfill the buffer. Apache web servers is limited to 4000 to 8192 bytes depending on the version. nginx web server places a limit on total length of HTTP headers + requested URI.

415 Unsupported Media Type

Uploading (POST, PUT, . . .) using a format the server doesn't understand. Like posting pictures formatted in JPEG2000 or some other esoteric format.

416 Request Range Not Satisfiable

You sent a `Range:` header to get just part of a file but the part you asked for doesn't make sense. An example could be asking to resume a download that was interrupted for a 1 Mib file at 1.3 Mib

417 Expectation Failed

The server cannot meet the `Expect:` header. An example could be that the client send a `Expect: 100 Continue` while POSTing multipart/form-data, but the server can't do that.

418 I'm a teapot

Indicates that the server refuses to brew coffee because it is a teapot. Was an April Fools' joke that was built into the standard.

422 Unprocessable Entity

Indicates that the server understood the Content-Type and the syntax of the entity (request body) is correct but that it was unable to process it.

426 Upgrade Required

Indicates that the server requires use of HTTP 2 or later.

428 Precondition Required

Indicates that the client needs to send a request with an `If-` header.

429 Too Many Requests

Indicates that the client has sent too many request in a short period of time

431 Request Header Fields Too Large

Indicates that the client has sent request headers that are too long.

451 Unavailable For Legal Reasons

Indicates that the server could service the request if it wasn't illegal.

Note

The 451 code was used because it's a reference to the book Fahrenheit 451. People also call this code 451 censorship.

5XX: Server Error codes

> It's the server's fault

Basically anything that deals with the server crashing, or a script on the server crashing, etc.

500 Internal Server Error

Server side software encountered some kind of error

501 Not Implemented

The server can't fulfill that request (such as a n HTTP PUT) because it doesn't even know what HTTP PUT is.

502 Bad Gateway

The server talks to anther HTTP server to fulfill this request and that other server isn't working.

503 Service Unavailable

The service is temporarily down. Something's broken and we'll bring it back up eventually.

504 Gateway Timeout

The server talks to another process to fulfill this request and that other process isn't responding fast enough. Very common when a webapp is overloaded. Similar to 502, except in this case the packets between the reverse proxy and the origin webserver are just vanishing.

505 HTTP Version Not Supported

Your request used the wrong HTTP version. A version the server no longer supports. For example Twitter doesn't let you do HTTP/1.0 requests anymore.

511 Network Authentication Required

Used by *captive portals* to tell the web browser that it should show the user a login page for the network. Your request never made it to the web server until after you've authenticated on the network.

Status code summary

So what codes should you use? Some suggest that business logic errors (software error reporting) should be done with the client without codes. Some suggest that you should use HTTP status code. In general the decision should be decided if the application is user facing and how you should handle it for your audience.

- Do you need to talk to the User Agent or to the user?
- Example: many websites let you log in with cookie-based authentication. There's no standards-compliant way to use a response code, e.g. 401 Unauthorized to tell the user to log in this way. So it may be best to use a 302 or 307 redirect to send the user to the login page.

HTTP Headers

There are a lot of headers within the HTTP specification. They all provide additional context and information to the request and response messages to and from the server. These headers have an effect on:

- Authentication
- Caching
- Encoding
- Partial downloading
- Content type
- More...

You can look them up on RFC 2616 Section 14.

HTTP Request Headers

These headers are only allowed in a request message to the server.

Accept

Specifies the kind of media the client can handle

Syntax	Meaning
`<MIME_type>/<MIME_subtyp`	A single precise MINE Type like `text/html`
`<MIME_type>/*`	A MIME type, but without a subtype
`*/*`	Any MIME type
`;q=`	Weight to add on a MIME type

Accept-Charset

Specifies character encodings the client can handle.

Accept-Encoding

Specifies compression formats the client can handle. Examples

- gzip
- compress
- deflate

Access-Control-Request-Headers: Content-Type

Lets the browser ask the server if JS is allowed to make requests with those headers

Access-Control-Request-Method: POST

Lets the browser ask the server if JS is allowed to make e.g. POST requests

Authorization

Syntax. `Authorization: <auth-scheme> <authorisation-parameters>` Where: `<auth-scheme>` is the authorization scheme that defines how the credentials are encoded.

The user agent is sending a username and password or other kind of credentials to the server (rare, usually cookies are used instead)

Cache-Control

Asks the server/proxy not to send data thats been sitting in its cache too long. The directives could be in seconds like `max-age=60`.

Content-Disposition

Syntax. `Content-Disposition: attachment; filename="filename.jpg"`.

Tells the server what the name/filename of the form data being uploaded when POSTing multipart/form-data.

Cookie

The user agent is sending cookies (stored key-value pairs) relevant to the server. The cookies were previously sent to the user agent to store by the server or JS.

DNT

Do Not Track, largely ignored because webservers wants to track you regardless. The user prefers not to be tracked over receiving personalized content.

Expect: 100-continue

The user agent is expects the server to respond with `100 Continue`.

Forwarded

Syntax. `Forwarded: by=<identifier>; for=<identifier>; host=<host>; proto=<http|https>`. Where:

<identifier>	**An identifier disclosing the information that is altered or lost when using a proxy.**
by=<identifier>	The interface where the request came in to the proxy server.
for=<identifier>	The client that initiated the request and subsequent proxies in a chain of proxies.
host=<host>	The Host request header field as received by the proxy.
proto=<http|https>	Indicates which protocol was used to make the request (typically "http" or "https").

Used by (reverse) proxies to tell the server who made the original request, over what protocol, and what the original Host header was.

From

Syntax. `From: <email>`.

Email address of the person making the requests. Example: bot owner, so people can contact them about their bot if it misbehaves.

Host

Syntax. `Host: <host>:<port>`.

The hostname (and sometimes port) of the website the user agent is trying to connect to. When a single server or proxy is handling requests for many different websites, it needs to know which site the request was made to. Otherwise it only can differentiate by IP address, but server/proxy usually has only one public IP address. Required in HTTP/1.1 and later for all requests. If Host: is missing the server may respond with 400 Bad Request.

If-Match

Asks the server to send the content only if the ETag(s) matches the specified string.

If-None-Match

Asks the server to send the content only if the ETag(s) **doesn't** match the specified string.

If-Modified-Since

Asks the server to send the content only if it's changed recently.

If-Unmodified-Since

Asks the server to accept the request only if it hasn't changed recently.

If-Range
Only resume download if I'm still going to download the same version, otherwise start over.

Origin
Tells the server where the JS code making this request came from.

Proxy-Authorization
Log into forward proxy server.

Range
Resume download at the specified point.

Referer
Tells the server what URI you were viewing that caused you to make the current request Shows what page had the link you clicked to get to the current page Destroys privacy Lets server admins know, for example, that people are finding their page on Google, or Twitter, etc.

TE
Like Accept-Encoding but used with a proxy.

Upgrade-Insecure-Requests
Asks the server to send a redirect to the HTTPS version of the page.

User-Agent
Tells the server what version of browser/client is making the request.

HTTP Server Headers

Access-Control-Allow-Credentials: true
Whether to allow JS code running in the browser to make requests with cookies.

Access-Control-Allow-Headers
Whether to allow JS code running in the browser to make requests with extra headers specified in the header.

```
Access-Control-Allow-Headers:
[<header-name>[, <header-name>]*]
```

Access-Control-Allow-Methods
What methods the server will allow JS code running in the browser to make.

```
Access-Control-Allow-Methods: <method>,
<method>
```

Access-Control-Allow-origin
```
Access-Control-Allow-Origin: *
Access-Control-Allow-Origin: <origin>
Access-Control-Allow-Origin: null
```

Only allow JS that came from a certain host to make requests to this server

Access-Control-Expose-Headers
```
Access-Control-Expose-Headers:
[<header-name>[, <header-name>]*]
```

Whether to allow JS code running in the browser to see headers

Access-Control-Max-Age
```
Access-Control-Max-Age: <delta-seconds>
```

How long the browser can remember the other access-control-headers

Age
```
Age: <delta-seconds>
```

(reverse) proxy cache that's e.g. 24 seconds old

Allow
```
Allow: <http-methods>
```

Tells the user agent what HTTP methods the server supports

Connection: close
Tells the server to close the connection after it's done sending the content

Content-Disposition
Tells the browser to prompt the user to save the content with a default filename, instead of displaying the content.

Similar to the Request version of the header.

Content-Encoding
Tells the browser to decompress the content before using it and what format its compressed in.

Similar to the Request version of the header.

Content-Language
Tells the browser what natural language the content its sending is in

Content-Length
Tells the client how many bytes of content to expects.

Content-Location
Tells the client where it can find the content that matches the Accept, Accept-* headers it sent. Tells the client where it can find the content that it created using, e.g. POST.

Content-Range
Tells teh client what bytes of the requested content the server is sending, out of the total length in bytes.

```
Content-Range: <unit>
<range-start>-<range-end>/<size>
Content-Range: <unit>
<range-start>-<range-end>/* Content-Range:
<unit> */<size>
```

Content-Security-Policy
Used to combat Cross Site Scripting attacks. Example: Restrict the places that the User Agent should fetch and run JS from.

Content-Security-Policy-Report-Only
Used to debug `Content-Security-Policy`

Content-Type
Tells the user agent what kind of media the server's sending

ETag
Version number/name of the content. If it changes it means the content has changed. Example: a SHA1 hash of the content.

Expect-CT
Tells the web browser to double-check the TLS certificate with public certificate log service

```
Expect-CT: report-uri="<uri>", enforce, max-age=<age>
```

Date
The date and time the server sent the content.

Expires
Tells the user agent or caching proxy to cache a response only until the specified date and time.

Last-Modified
Tells the user agent or caching proxy when the content last actually changed.

Keep-Alive
`Keep-Alive: [timeout=<sec>] [max=<int>]`

Tells the client not to send more than `<int>` requests on this connection or to let it idle for more than `<sec>` seconds.

Location
Tells the client where to redirect to for 300 series status codes.

Proxy-Authenticate, Proxy-Authorization
Ask client to log into proxy server.

Public-Key-Pins
Tells the client not to only accept one of the two specified certificates from the webserver for the next certain amount of time. Dangerous: can lock people out of being able to view your site for a long time Disabled in Chrome. This is also a depreciated header.

Referrer-Policy: no-referrer-when-downgrade
Webserver tells the client not to send as much referrer information to the next webserver.

Retry-After
Webserver tells the client to wait a certain amount of time before retrying after a 503 Service Unavailable, 429 Too Many Requests or how long to delay before following a 3xx redirect

Server
Tells the client the server software and version, like User Agent but for servers.

Set-Cookie
Webserver tells the client to store a cookie (key-value pair) in its cookie jar. The cookie will be stored and sent back to the webserver every time the client makes a request, using the Cookie: header. But only if the security restrictions are met: only over HTTPS, not with requests made by JS, not with requests initiated by code that came from a different website, delete the cookie after a day...

Transfer-Encoding
Like Content-Encoding but used by a proxy. If `chunked` is specified data will be sent in chunks. Replaced by HTTP/2 for that encoding only.

Strict-Transport-Security
Don't contact this server or any subdomains without TLS for a specified period of time.

Trailer: Expires
Tells the client there will be additional headers after the content is sent. HTTP/1.1 requires Transfer-Encoding: Chunked for this to work. HTTP/2 doesn't work in all browsers.

Vary: Accept-Language
Tells a cache that this page is one of multiple versions of the page based on the HTTP request headers listed.

Via
Records the proxy chain a request or response went through.

```
Via: [ <protocol-name> "/" ] <protocol-version> <host> [ ":" <port> ] Via: [ <protocol-name> "/" ] <protocol-version> <pseudonym>
```

WWW-Authenticate
Sent with 401 Unauthorized to tell the user agent/user how to authenticate Browser will pop up a dialog box with "What is the password" on it

Warning
There are a whole bunch of warning codes

Code	Reason	Explanation
110	Response is Stale	Warns the client that the content was in the cache for too long.
111	Revalidation Failed	Warns the client that the content was in the cache but the original server is down.
112	Disconnected Operation	Warns the client that the content was in the cache but the network is down.
113	Heuristic Expiration	Warns the client that the content was in the cache over 24 hours.
199	Miscellaneous Warning	Warns the client about something.
214	Transformation Applied	Warns the client that the content was changed in some way by the proxy.
299	Persistent Warning	Warns the client about something the proxy doesn't expect to change anytime soon

X-Content-Type-Options: nosniff

Tells the client not to try to figure out the Content Type (MIME Type) of the content on its own, e.g. based on contents.

X-DNS-Prefetch-Control: off

Tells the browser not to resolve hostnames in links before the user clicks on them.

X-Frame-Options: deny

Tells the browser not to show the page in a <frame>, <iframe>, <embed> or <object>.

Custom HTTP Headers

Used to have use X- headers, but X- headers are now deprecated. You can add whatever headers you want as long as they don't conflict with standard headers.

Authentication

Multiple strategies for authentication using HTTP exist:

- Cookies
  - Session-based (stateful)
  - Token-based (stateless)
  - Easy to mess up and create a security problem
- Authorization:
  - Basic (old, insecure)
  - JWTs (JSON Web Token), RFC 7519 <- this is the most common

Session Cookies

1. User browses to site
   - Doesn't have cookie yet so browser doesn't send any Cookie: headers
2. Server responds with cookie
   - Generates a large random number, ex: 86e7eca9-d484-48ad-98ed-e060541b7870
   - Records session number in server's database
   - Responds with Set-Cookie header, ex: Set-Cookie: s=86e7eca9-d484-48ad-98ed-e060541b7870; Secure; HttpOnly; SameSite=strict
   - Responds with redirect or page allowing the user to log in

HTTP proxies

Forward Proxies

- Rare
- Forward HTTP requests for you
- You set it up
- Example: Watch US Netflix from Canada

Reverse Proxies

- Common
- Forward HTTP requests for the server
- Set up by the people managing the server
- Example: Cloudflare CDN

Reverse proxies are used for

- Caching
- CDN: Keep content nearer the user
- Load balancing
- TLS
- Fail-over

Generally there are multiple layers of proxy servers that communicate between the user agent and the application server. The application server's database could also be on a separate system adding another layer of latency.

Review

URI. Universal Resource Identifier, the location of a resource on a "webserver"

HTTP Methods. Here are the four basic ones you should know:

- GET - Get (potentially) dynamic information from a URI.
- POST - Run search, apply forms, append, and change data URI.
- PUT - Store the entity at that URI, full replacement (overwrite) or insertion at the URI.
- DELETE - Delete the resource at that URI

URL.

- Has a syntax and schema.
- Can be absolute or relative for both service and path.

- Can query using (`?key1=value1&key2=value2&...`) in the URL but is limited, long queries should be done in a POST.
- Queries could just be a string as well
- Fragments are accessed using `#` like accessing a section on a page

Request and Response.

- HTTP 1.1 requests requires the Request Line i.e., `METHOD path HTTP/1.1` and the `Host:` header.
- HTTP 1.1 responses requires the Response Line i.e., `HTTP/1.1 status_code`.
- All headers are ended with an empty `CRLF` or
  `r`
  `n`.
- HTTP header names follow capitalized snake case with the space is `-` and the first letter in each word is capitalized.

Status Codes.

- 1xx are informational
- 2xx are success codes
- 3xx are redirection codes
- 4xx are client side errors
- 5xx are server side errors
- There are whole bunch just look them up

Authentication.

- Cookies to store auth information, easy to mess up and creates a security problem. They can be session-based with a state or token-based that is stateless.
- Authorization tokens are more secure and commonly used.

Proxies.

- Forward proxies are client side that you have to setup and forward HTTP requests for you
- Reverse proxies are server side that does a lot of things, load balancing, caching, CDN which keeps the content closer to the user, etc.

Chapter 3: HTML

The HyperText Markup Language, or HTML is the standard markup language for documents designed to be displayed in a web browser.

The current version of has two standards

- WHATWG's HTML which is a living standard that is constantly being updated.
- W3C's HTML 5 Standard which is closer to a releases versioning standard. i.e., 5.1, 5.2, etc.

Generally people prefer the WHATWG's Living standard but because not all web browsers will constantly adhere to the the standard you should look at the capability table to see if the standard is implemented by the user agent/browser.

DOCTYPE

You need to tell the world that this is a modern HTML file. You must put a DOCTYPE at the top and enclose your content in the HTML tag.

```
<!DOCTYPE html>
<html>
...
</html>
```

XHTML

You can write HTML in XML format if you follow XML syntax: XHTML. Valid HTML is not valid XML, and valid XML is not valid HTML.

- For a while there was a drive to combine XML and HTML into "polyglot HTML" where you could write both at the same time, but this was abandoned in 2015.
- Browsers still support `<voidtag/>` syntax in HTML but it breaks other HTML parsers

XML is very unforgiving, a simple mistake could cause a parser to refuse to parse the document. Breaks `Document.write()`, the `<template>` tag, most entities...

Adds support for namespaces, HTML 5 XML has no DTD URL. Uses `Content-Type: application/xhtml+xml`.

Head/Header

The head tag is where we put information about the webpage. You will usually include a title here.

Meta tags contain meta information for browsers and other tools to help interpret.

```
<head>
  <title>Page Title</title>
  <meta charset="UTF-8">
  <meta name="description" content="Example
  ↪  Page">
  <meta name="author" content="Abram Hindle">

  <!-- proprietary extensions -->
  <meta name="apple-mobile-web-app-capable"
  ↪  content="yes" >
  <meta
  ↪  name="apple-mobile-web-app-status-bar-style"
  ↪  content="black-translucent" >
  <!-- mobile viewport information -->
  <meta name="viewport"
  ↪  content="width=device-width,
  ↪  initial-scale=1.0, maximum-scale=1.0,
  ↪  user-scalable=no">
  <!-- stylesheets / css -->
  <link rel="stylesheet"
  ↪  href="css/reveal.min.css">
  <link rel="stylesheet"
  ↪  href="css/theme/default.css"
  ↪  id="theme">
  <link rel="stylesheet"
  ↪  href="lib/css/zenburn.css">

</head>
```

Body

Is where all the content goes.

```
<body>
<p>Visible content goes here.</p>
<p>Consider that you should not directly
↪  specify layout information
  here but that you should mark up blocks
  ↪  with a class allowing you to
  apply layouts later. You can abstract
  ↪  layout</p>
</body>
```

Tags

All tags should be closed to allow for ease of reading and parsing, except void tags.

Start Tags are enclosed in < and >

End Tags are enclosed in </ and >

```
<tag>
  <enclosedtag>
  </enclosedtag>
</tag>
```

Void tags are enclosed in < and > like start tags, but don't have a matching end tag.

```
<tag>
  <voidtag>
  <br>
</tag>
```

Unless you are writing XHTML, in which case you must follow XML syntax, void tags typically look like `<voidtag>`. but sometimes are written like `<voidtag/>`.

This is against the HTML spec.

Tags can be any capitalization in HTML, but typically they are written in lower case. Stick to lower-case when writing HTML.

Paragraph Tag: <P>

```
<p>
  Now bears us onward one of the hard
  ↪  margins,
  And so the brooklet's mist o'ershadows it,
  From fire it saves the water and the dikes.
</p>
<p>
  Even as the Flemings, 'twixt Cadsand and
  ↪  Bruges,
  Fearing the flood that tow'rds them hurls
  ↪  itself,
  Their bulwarks build to put the sea to
  ↪  flight;
</p>
```

We can use CSS to style these text blocks. Whitespace in the <p> block doesn't matter. So formatting will most likely be off.

> Now bears us onward one of the hard margins, And so the brooklet's mist o'ershadows it, From fire it saves the water and the dikes.
>
> Even as the Flemings, 'twixt Cadsand and Bruges, Fearing the flood that tow'rds them hurls itself, Their bulwarks build to put the sea to flight;

Breakline Tag

We can make line-breaks explicit with `<br>`.

Image Tag <img>

We want to show images, we cannot embed them in HTML easily so we use hyperlinks (URIs) to reference them and include them.

Remember to include alt tags so they are machine and human-readable.

```
<img src="images/plasma.png" alt="Plasma
↪  Pattern">
<!-- scaling -->
<img src="images/plasma.png" alt="Plasma
↪  Pattern" width="30">
<!-- scaling without respecting aspect -->
<img src="images/plasma.png" alt="Plasma
↪  Pattern" width="30" height="60">
```

The `img` tags has certain attributes that are important. The `alt` tag should be used to describe the image using text. This is sometimes a requirement for accessibility requirements for people who have a visual impairment or for Search engine optimizations. Where most SEOs uses text, not images, although Google might have the capabilities for processing images.

The `width` and `height` attributes are used to set the size of the image. If one of the size attribute is specified then the aspect ratio is preserved otherwise it will follow both the height and width attributes specified by the developer.

Anchor Tag <a>

The anchor tag is used to make hypertext or text with hyperlinks.

Anchor tags let us link to other documents or locations. The `href` attribute links to a URL. The URL can be relative to the location of the current page. Anchor tags used to place anchors on pages, but in HTML 5 they just navigate to ID'd sections now.

```
<a href="http://slashdot.org">Slashdot.org:
↪  News for Nerds Stuff that
Matters</a><br>
<a href="http://cnn.com">T</a><a
↪  href="http://msn.com">a</a>
<a href="http://softwareprocess.es/">g</a><a
↪  href="http://ualberta.ca">s</a>
don't have to be very long.<br>
<a href="../">The a directory down!</a>
↪  Relative Link
```

You can also have images within the anchor tags to allow images to become buttons to another link. Elements within the anchor tags are often highlighted in blue to show that it's a hyperlink and a the URL of the link will show on the bottom right corner of the web browser.

Text Layout Philosophy

Let the user agent decide how to display the text. You don't need to control everything. But provide the appropriate semantics first. Then apply layout information to those semantics.

If you want something to show up in a certain make a CSS class and style a span or a div or a paragraph that way.

If you want a fancy layout use flexbox or grid layouts.

and tags

`<div>` and `<span>` tags have no particular meaning. They are used for separating content within the body and allow different style attributes to be applied. Using div and span as is will not change the content of style of the content.

- `<div>` will cause a line break before it is displayed.
- `<span>` will be inlined.

Example

```
<div>
<span style="background-color:#AAAAAA;
↪  font-weight:bold">
  Now bears us onward one of the hard
  ↪  margins,</span><br>
  And so the brooklet's mist o'ershadows
  ↪  it,<br>
  From fire it saves the water and the
  ↪  dikes.<br>
</div>
<div style="font-size:150%">
Even <span style="color:red">as the
↪  Flemings</span>, 'twixt Cadsand and
↪  Bruges,<br>
  Fearing the flood that tow'rds them hurls
  ↪  itself,<br>
  Their bulwarks build to put the sea to
  ↪  flight;<br>
</div>
```

Style

Tags can have attributes defined like so:

```
<tag attribute1="value1" attribute2="val2">
```

These attributes could affect the style of the content. The style attributes are Cascading Style Sheet properties.

Cascading Style Sheet (CSS)

A language to style an HTML document.

- Cascade - The children inherits properties of the "parent"
- Style - Properties refer to style properties such as layout, position, color, font, background, etc.
- Sheets - Apply to a page, change a page.

CSS can be applied on a per tag level, but can also be applied globally.

```
<!-- Include a file of CSS  -->
<link href="path/to/cssfile.css"
↪  rel="stylesheet">

<!-- Inline CSS -->
<style type="text/css">
.angry {
font-weight:900;
zoom: 3;
}
.angry:nth-child(odd)
{
color:green;
transform:rotate(7deg);
-webkit-transform:rotate(7deg); /* Safari and
↪  Chrome */
float: left;
}
.angry:nth-child(even)
{
color:red;
transform:rotate(-12deg);
-webkit-transform:rotate(-12deg); /* Safari
↪  and Chrome */
float: right;
}
</style>
<p style="color:orange">Apply style
↪  directly</p>
<div>
<div class="angry">HULK</div> <div
↪  class="angry">SMASH!</div>
</div>
```

Properties

Here are some of the properties to know

- color – color of the text or object
  - color:green
  - color:#abc (short form)
- font-family – the font
  - font-family:"Times New Roman"
  - font-family:"Verdana"
- font-size – the font size
-
  - font-size:10px;
  - font-size:10pt;
  - font-size:large;
  - font-size:200
- font-style – normal, italic, oblique
  - font-style:normal;
- background-color
- background-image

Selectors (classes)

A CSS selector selects the HTML element(s) you want to style.

Classes

CSS labels that starts with a period . denotes a class. We can use HTML attributes to label tags with classes that allow the tags to inherit CSS style information. The attribute name is `class`. E.x., `class="bg1"` for class bg1 or `.bg1`. You can also combine classes with HTML tag selectors; this example makes p tags with class="highlight" orange instead of yellow. You can override other CSS as well.

```
p.highlight { background-color:orange; }

<p class="highlight">ALERT in orange!</p>
```

ID

ID selectors are like class selectors except they aim at the one tag with the id="idtag" as an attribute

```
#yellowtag { background-color:yellow; }
```

Element

Element selectors let you style entire HTML elements (tags). Important because you might want to theme all divs or imgs or links.

```
p { background-color:yellow; }
```

This will style all paragraphs with a yellow background.

Context

Selectors that behave depending on the context Can be chained with other selectors

- :hover - when the mouse is over
- :active - active link
- :first-letter - operate on the first letter
- :nth-child(2) - second child

> **Note**
> The index for nth-child starts at 1 not 0 unlike in most programming languages.

Positioning

The `position` attribute is used to position a element within a page. It use left, right, top, bottom attributes to control the position.

- `fixed` - stays on one spot in the browser
- `relative` - position relative to where it normally goes
- `absolute` - absolute positioning relative to the first parent that was positioned (often the page itself).
- `z-index` - can be used to order overlapping elements. The higher the number the close it is to the screen

Alignment

One big problem with CSS is how to center something! Margins can be used or alignment attributes can be used.

- `text-align` - specifies the horizontal alignment of text in an element
- `flex` - sets the flexible length on flexible items. Helps for big centering jobs, things with many elements, etc
- `align-content` - aligns the flexible container's items when the items do not use all available space on the main-axis (horizontally).
- `justify-content` - defines how the browser distributes space between and around content items along the main-axis of a flex container, and the inline axis of a grid container.

HTML for User Interfaces

The HTML Form elements let us accept input from browsers in a structured way and form HTTP GETs and POSTs.

<form> tag

The form tag encloses a group of HTML input/UI widgets which can then be submitted as once.

- method - We can specify the HTTP method (POST/GET)
- action - We can specify the URI to GET or POST to.

<input> tag

The input tag can take in textual input, passwords, or act as submit button.

type

The input could of many different types including but not limited to

- button
- checkbox
- color
- date
- datetime
- datetime-local
- email
- file
- hidden
- image
- month
- number
- password
- radio - Allow only one selection from a group of buttons
- range
- reset
- search
- submit
- tel
- text
- time
- url
- week

name

The name of the data sent to the URI

value

The default value Hidden types allow you to embed values to send along to help the request.

Example

```
<form action="http://webdocs.cs.ualberta.ca/~hindle1/1.py" method="get">
What is your name? <input name="name"/></br>
What is your quest? <input name="quest"/></br>
What is your favorite colour? <input name="colour"/></br>
<input type="submit"/>
</form>
```

<SELECT> tag

Select tag is for a dropdown box of options.

```
<form action="http://webdocs.cs.ualberta.ca/~hindle1/1.py" method="get">
What kind of cake shall we have?</br>
<select name="caketype">
  <option value="angel">Angel Food Cake</option>
  <option value="devil" selected>Devil's Food Cake</option>
  <option value="cowpatty">Marie Antionette's Cake</option>
</select>
<input type="submit"/>
</form>
```

File Upload

You can upload files with the input tag but we need to switch the POST encoding.

```
<form action="http://webdocs.cs.ualberta.ca/~hindle1/1.py"
      enctype="multipart/form-data"
      method="get">
  Choose a file to upload!
  <input name="uploadFile" type="file" />
  <input type="submit"/>
</form>
```

CGI Diversion

Did you know you can write CGI in just about any language?

All it takes is printing HTTP Response headers, reading environment variables and reading from STDIN!

Here's a minimal PERL example.

```
use Data::Dumper;
print "Content-type: text/plain\r\n\r\n";
print Dumper(\%ENV);
my $line = 0;
while(<>) {
    print $line++ . " " . $_ . $/;
}
```

So you can use multiple languages.

I really really really recommend using the CGI library that comes with your language if you do. Mostly for parsing of GET and POST arguments.

```
import cgi
import cgitb
cgtib.enable()
# print your header
```

CGI Problems

There are many issues with CGI

- Slow

- 1 process per invocation
- Inefficient communication of requests
- Lack of object-orientation representation of a request
- Difficult to share state

Review

HTML.

- The language used for documents to be displayed on web browsers
- People prefer the WHATWG standard but not all web browsers supports it because it's a living standard
- Must have `<!DOCTYPE html>` at the beginning of the document.
- You can write HTML in XML format but the compatibility between the two ain't great.

Header.

- The header is where you put information about the webpage
- `<meta charset="UTF-8">` goes here.
- All `meta` tags goes in the header.
- Title goes here as well as the CSS style sheet under the `<link rel="stylesheet" href="path_to_css.css">`.

Body.

- The body contains the actual content of the page
- All tags must have an opening `<tag>` and a closing `</tag>`
- `<voidtag>` and `<br>` don't have a matching closing tag.
- `<p>` paragraph tags doesn't care about white space. Use `<br>` to breaklines explicitly.
- `<img>` Image tags shows images and you should set the `alt` property within that tag for text-to-speech on images and SEOs.
- `<a>` anchor tags are used to make hypertext or text with hyperlinks. Basically what makes Wikipedia great.

Text Layout.

- `<div>` will create a newline before it's content is displayed
- `<span>` will separate a div for column vertically.

CSS.

- The children will inherit properties of the parent node
- A css file can apply the style globally or on a per tag level.
- Style properties can be applied to certain selectors like
  - Tags like `div`
  - Classes that start with a `.` for example `.pokemon` will be applied to any tags with the class `<... class="pokemon">`
  - Tags that start with `#` for example `#yellowtag` will be applied to any tags with the id `<... id="yellowtag">`
  - Based on context with the `:` for example `:hover` apply when the mouse is over the element.
- You can set position using fixed, relative, absolute, and z-index for overlapping
- Alignment using text-align, flex, align-content, justify-content.

Forms.

- You can create user submittable forms using the `<form>` tag
- The input for the form is defined using the `<input>` tag and you have to define the type, the name of the input to be sent to the URI, and a default value.
- The `<select>` tag can be used to show a dropdown box of options.

CGI Diversion.

- You can write CGI in any language
- Basically you can create a program that will generate html content dynamically at runtime.
- Has some problems.

Chapter 4: JavaScript

Where did it come from? Created in one day by Brendan Eich for Netscape in 1995 and is inspired by Java, C, Self, Perl... It's Multi-paradigm

- Imperative
- Functional
- Object-oriented
- Prototype-driven
- Event-driven
- Embeddable

JavaScript is best known for it's weird type casting behavior, this leads to a lot of debugging issues.

The reason why we keep using it is because it runs everywhere:

- Browsers
- Servers
- PDFs

It's also fast on modern browsers, they can compile javascript to machine code.

Running JS on a webpage

You can run JavaScript within the `<script>` tags inside a html file.

```
<script>
var someJS = "JS in a <script> tag!";
</script>
<!-- you can embed oneliners within HTML! -->
<button onClick="alert('Stuck in JS factory,
↪  send help!');">
  Test me!</button>
```

You can also put JavaScript at a separate URL with the `<script>` tag as well.

```
<script src="myscript.js"></script>
<!--You have to put the closing </script> tag
because script isn't a void tag!-->
```

This allows the html content to be cached while the content of `myscript.js` can change without affecting the caching behavior of the source html file.

Functions

- Functions can return values
- Functions can have parameters
- Functions can access all available scope

```
// Function one with no parameters
function one() {
    return 1;
}
// A function with 1 param
function f(x) {
    return 2*x;
}
// How to call the function
f(2) == 4.0;
// Show on the console each click event
// Unnamed function
button.onclick = function(e) {
    console.log(e);
};
```

Comments

```
// This is a line comment
/*
This is a block comment
*/
```

Closure

A closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
// I'm some javascript
/* also a comment */
function outer(text) {
    function inner(x) {
        alert(text);
    }
    return inner;
}
f = outer("Hi mom I'm on the projector");
f(1);
```

The code segment before will give an alert to the user with `Hi mom I'm on the projector`

Another example:

```
function p(a) {
    function q() {
        console.log("a: " + a);
        console.log("arguments: " +
        ↪  arguments[0]);
    }
    return q;
}
r = p(1);
s = p(2); // now p is finished running, what
↪  happens to a?
r(3); // print 1
s(4); // print 2
```

Even though `r` will print 1 and `s` will print 2 the arguments that `r` and `s` will print is 3 and 4 respectively.

This is because `function p` will return `function q` to the caller that is `r` and `s`. When give parameters to `r` and `s` those arguments will go to `function q`. However, `function q` doesn't take in any parameters. What gives? In JavaScript giving additional parameters to a functions is stored in the `arguments` variable as a list. Where the first additional argument is stored in `arguments[0]`, the next additional argument is stored in `arguments[1]` and so on.

Scope

By default creating a variable makes globals. To create a local variable use

- `'use strict'`
- `let`
- `const` for constants
- `var`

- `let` and `const` scopes variables/constants to the enclosing block
- `var` scopes variables to the enclosing function

```
"use strict";
function f() {
    let a = 1;
    for (let i = 1; i<10; i++) {
        var c = i; // var defines it inside
        ↪  the function
        let b = i; // let only defines it
        ↪  inside the for loop
    }
    console.log(c); // 9
    console.log(b); // Reference Error
}
f();
console.log(a); // Reference Error
```

Note

`let` should be used over `var` because regardless of where a `var` variable is created it's is hoisted to the top of it's scope. While `let` behaves similar to a regularly defined variable.

Type Coercion

Python does a lot of implicit type conversion

```
"1" + 1 //-> "11"
```

There is a bias towards strings and floats for conversions.

this

Like `self` in Python, except: You don't have to list in the arguments. It's not automatically bound to an object.

In most cases, the value of `this` is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called.

In non-strict mode the value of `this` is always a reference to an object. In `'use strict'` mode it can be any value.

```
class Quiz {
  write() {
    console.log(this);
  }
}
quiz = new Quiz();
w = quiz.write
w()
```

In JavaScript the w method isn't bound to the quiz object unless we call it explicitly like `quiz.write`, `w.call(quiz)` or bind it with `w.bind(quiz)`

ES5 introduced the `bind()` method to set the value of a function's this regardless of how it's called, and ES2015 introduced arrow functions which don't provide their own this binding (it retains the this value of the enclosing lexical context).

Anonymous Functions

JavaScript allows anonymous functions by returning a function definition.

```
var outer = function(text) {
    return function(x) {
        console.log(this);
        alert(text);
    }
}
var f = outer("Hi mom I'm on the projector");
f.call({}, 1);
```

Arrow Function =>

Like anonymous functions but they always keep the `this` from when they were created.

```
var outer = function(text) {
    return (x) => {
        console.log(this);
        alert(text);
    }
}
var f = outer("Hi mom I'm on the projector");
f.call({}, 1); // {} is ignored because it's
↪  an arrow function
```

Names

- Start with a letter followed by underscores, letters or numbers.
- Can't be a reserved word like `break` or `case` or `for` or `function` or `if` or `in` etc.
- Convention is to use `camelCase` like Java not `snake_case` like Python.

```
var aString = "Strings";
var break = "not allowed!";
var BREAK = "This is allowed!";
var BrEAK = "Try not to abuse case
↪  sensitivity";
```

Numbers

Everything is a double, write integers, decimals, or decimals with an exponent.

```
var aNumber = 10;
var aNumber = 11.11;
var aNumber = 1e-100;
var aNumber = 1E+100;
var nan = NaN;
var inf = Infinity;
var negativeInfinity = -Infinity;
```

Casting Numbers

There are many different ways to cast numbers

Casting to an "integer"

```
Math.floor(0.7)
Math.ceil(0.7)
Math.round(0.7)
Math.trunc(0.7)
```

Casting to a float

```
parseFloat("127")
Number("0x7F")
+"0x7F" // Unary plus is the same as Number
```

JavaScript also accepts string hexadecimal for conversions to numbers

Rounding Errors

Since everything's a double, you get rounding errors.

```
a = 0.1
b = 0.2
a + b == 0.3 // Should be true but false due
↪  to a rounding error
```

Strings

- Unicode by default
- Use ' ' or " "
- Any characters except control characters and " or '

```
var aNumber = 10;
var aString = "";
var anotherString ="Hi how are you";
var escapesString = "\r\n\t\f\b\/\\\\'\"";
var snowMan = "\u2603";
snowMan.length === 1;
aString.length === 0;
```

Casting Strings

There are a ways to convert things to a string

```
"" + 1;
1 + "";
String(1);
(1).toString();
String(null); // "null"
null.toString(); // Error
(127).toString(16); // "7F"
```

Booleans

- `true` or `false`
- Unfortunately conditional expressions have lot of truthy and falsey values
- False values:

```
false
null
undefined
''
0
NaN
```

- Everything else is true plus `true` itself.

Equality

- `==` the *abstract* equality operator
- `===` the *strict* equality operator i.e., type must match.

```
3=="3" // true
3==="3" // false
1==true // true
1===true // false
undefined == null // true
undefined === null // false
NaN==NaN // false
NaN===NaN // false
isNaN(NaN) // true
```

Generally you should use `===`.

Arrays

Arrays in JavaScript are object-oriented and fill of methods.

Unlike MATLAB arrays are 0-indexed.

```
var empty = [];
var arrayInitialized = [1,2,3,4,'5'];//mixed!
var arr = new Array(10);
arr[0] === undefined;
arr[0] = 'Assigned';
'Assigned' === arr[0];
arrayInitialized[4] === '5';
arrayInitialized.length === 5;
arrayInitialized.splice(3,1); // delete 4
↪  from the array (slow)
```

Iterating over an array

You can use `for ... of` to iterate over iterable objects, including; strings, arrays, and array-like objects.

```
let a = [1, 2, 3, 4, 5];
for (let i of a) {
  console.log(i);
}
```

If you use `for ... in` on an iterable object, like strings, arrays, and array-like objects then you will get the *indices* of those iterated elements rather than the elements themselves.

Objects

Everything is an object except for these primitives

- Booleans
- numbers
- strings

Although those primitives still have methods, **objects** have properties. These properties are named by a string and property values can be anything including `undefined`. Objects don't have a class and objects are pass by reference.

Example

```
var empty = {};
var abram = {
   "name":"Abram Hindle",
   "job":"Throwing Down JS",
   "favorite tea":"puerh"
};
var dog = {
   paws: 4 // note I didn't quote paws
};
dog.paws === 4;
abram["favorite tea"] === "puerh";
abram.name === "Abram Hindle";
abram["favorite tea"] = "oolong";
```

More examples

```
undefined.property; // Throws a type error
undefined && undefined.property // returns
↪  undefined
var empty = {};
empty.property === undefined;
var abram = {
   "name":"Abram Hindle",
   "job":"Throwing Down JS",
   "favorite tea":"puerh"
};
keys(abram); // produces
↪  ["name","job","favorite tea"]
//prototype!
var abramChild = Object.create(abram)
keys(abramChild); // produces []
abramChild.name === "Abram Hindle"; //
↪  inherits keys from abram
```

Prototypes

All javascript objects inherit properties and methods from a prototypes. Prototype provides inheritance in JavaScript where objects can have a prototype object, which acts as a template object that it inherits methods and properties from.

```
var abram = {
   "name":"Abram Hindle",
   "job":"Throwing Down JS",
   "favorite tea":"puerh",
   "sayName": function() {
      alert(this.name);
   }
};
abramChild = Object.create(abram);
abramChild.name = "Child";
function doit() {
    abram.sayName();
    abramChild.sayName();
}
```

In this case `abramChild` is a object created via the `abram` prototype. It inherited all the properties from abram as well as their default values, but can change the value of the properties after initialization. In this case change `abramChild.name` to `Child` instead of `Abram Hindle`.

Looping over Properties

The `for ... in` statement iterates over all enumerable properties of an object that are keyed by strings, including inherited enumerable properties from the prototype.

```
let author = {
   "name":"Unknown Slide Author",
   "job":"Making Slides",
   "sayName": function() {
      alert(this.name);
   }
};
let hazel = Object.create(author);
hazel.name = "Hazel Campbell";
for (let property in hazel) {
  alert(property + ": " + hazel[property]);
}
```

`Object.keys(object)`

If you want the properties or keys of the object and **exclude their inherited properties** then use the `Object.keys(object)` method to iterate over the object's own properties and not their inherited properties.

Classes

Classes are in essence, "special functions", and and just as you can define function expressions and

function declarations, the class syntax has two components: class expression and class declarations.

JavaScript has several ways of creating `classes`,

- ECMAScript 2015 classes
- Constructor functions

ECMAScript 2015 Classes

The proper way of creating classes is using the ECMAScript 2015 classes.

```
class Pokemon {
  constructor(name, level) {
    this.name = name;
    this.level = level;
  }
  levelUp() {
    this.level += 1;
  }
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

This is similar to C++ where you have a constructor method within the class definition that initializes a class instance. Class methods are defined in a similar way to the constructor.

Constructor Function

You might see this in older code

```
// classes start with a capital letter by
↪  convention
function Pokemon(name,level) {
  this.name = name;
  this.level = level;
  this.levelUp = function() {
    this.level += 1;
  }
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

Where instead of a `class` keyword and definition you have a function that will define and initialize its data members and define the class methods within the body of the function constructor.

```
// classes start with a capital letter by
↪  convention
function Pokemon(name,level) {
  this.name = name;
  this.level = level;
  this.levelUp = function() {
    this.level += 1;
  }
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

Both instances uses new to create a new class instance.

Adding a method to a prototype

If you want to add a new or overwrite a method to an already defined class you can assign the new function on the class's prototype.

```
// classes start with a capital letter by
↪  convention
function Pokemon(name,level) {
  this.name = name;
  this.level = level;
}
// have to include "prototype" here so "this"
↪  works in the method
Pokemon.prototype.levelUp = function() {
  this.level += 1;
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

You can also use the `Object.Create(prototype)` convention to create a new object from a "default" object. Though this is awkward and rarely used.

```
// classes start with a capital letter by
↪  convention
var Pokemon = {
  name: null,
  level: null,
  levelUp: function() {
    this.level += 1;
  }
}
pikachu = Object.create(Pokemon);
pikachu.name = "Pikachu";
pikachu.level = 1;
pikachu.levelUp();
```

DOM Manipulation

You can use JavaScript to manipulate what's on the page. The browser turns HTML into the DOM or Document Object Model.

- Document: the stuff on your page, the content
- Object: gets turned into objects accessible by JS
- Model: it's a tree with children nodes

DOM elements from HTML

```
<p>A paragraph</p>
<div>
  Hi!
  <a href="https://google.ca">Click me!</a>
</div>
```

The DOM elements for the html document is

- document (it's a tree with children nodes!)
- Root Element: HTML (document.children[0])
- Element: Head (document.children[0].children[0])
- Element: Body (document.children[0].children[1])
- Element: p (document.children[0].children[1].children[0])

- #text: A paragrah
- Element: div
- Text: Hi!
- Element: a ; attribute href
- Text: Click me!

Recursive

Because the DOM is a tree you can recursively traverse the in whatever ways you see fit.

```
function domRecurse(start, depth) {
  var out = "";
  for (child of start.childNodes) {
    indent = "    ".repeat(depth);
    out += indent + "* " + child.nodeName;
      if (child.nodeName === "#text") {
        out += " " +
        ↪  JSON.stringify(child.wholeText);
      }
    out += "\n";
    out += domRecurse(child, depth+1);
  }
  return out;
}
```

querySelector

You can also jump straight to an element using `querySelector`. It uses CSS-style selectors for the query. You don't need jQuery.

```
function domRecurseExample() {
  var start =
  ↪  document.querySelector("#domexample");
  alert(domRecurse(start, 0));
}
```

You can use other selectors than `querySelector` to find specific element(s).

```
// Get all DIVs
var divs =
↪  document.getElementsByTagName("div");
// gets all elements with class divider
var dividers =
↪  document.getElementsByClassName("divider");
// get the element with the ID main
var main = document.getElementById('main');
// get the element by name
var ups = document.getElementsByName('up');
```

jQuery

jQuery's functionality is now available from APIs built-in to browsers.

Old projects that need backwards-compatibility or already use jQuery:

- Using jQuery is totally cool!

New projects using ECMA 2016 and later:

- Better to just use the tools the browser gives you.

Changing the DOM

You can change the DOM by creating elements on the `document`

Filling a div

This will add

> Here's some text in that div! Let's make a link too!

The `Let's make a link too!` is an `a.href` to `https://google.ca/`.

```
<div id="fillme"></div>

function fillExample() {
  fillme = document.getElementById("fillme");
  fillme.textContent = "Here's some text in
  ↪  that div! ";
  a = document.createElement("a"); // create
  ↪  an a element
  a.textContent = "Let's make a link too!";
  a.href = "https://google.ca/";
  fillme.appendChild(a);
}
```

Adding a colour border

This will add a randomly colored border to each of the two `div` elements.

```
<div class="styleme">I'm text in a div!</div>
<div class="styleme">I'm text in another
↪  div!</div>

function styleExample() {
  var divs =
  ↪  document.getElementsByClassName("styleme");
  divs = Array.prototype.slice.call(divs); //
  ↪  convert HTMLCollection to Array
  divs.map( (div) => {
    console.log(div);
    div.style.border = "5px solid";
    div.style.borderColor = "rgb(" +
    ↪  (256*Math.random())
      + ", " + (256*Math.random())
      + ", " + (256*Math.random()) + ")";
  });
}
```

This will apply a random border color to both divs.

```
function styleExample2() {
  old =
  ↪  document.getElementById("styleme2sheet");
  if (old) {
    old.parentNode.removeChild(old);
  }
  var css = "div.styleme2 { border: 5px solid
  ↪  "
      + "rgb(" + (256*Math.random())
      + ", " + (256*Math.random())
```

```
        + ", " + (256*Math.random()) + "); }";
    style = document.createElement('style');
    style.type = "text/css";
    style.id = "styleme2sheet";
    style.textContent = css;
    document.head.appendChild(style);
}
```

Review

Running JS in HTML.

- Use the `<script>` tag to either inline JS code or even better
- Set the `src` property to the JavaScript file so you don't invalidate the page's cache.

Function.

- Functions can return values.
- Functions can have parameters.
- If parameters are not defined you can still pass arguments to the function by passing them regardless and use the `arguments` array in the function body to access them.
- Function can access all available scope.
- Everything is passed by reference.

Closure.

- Gives you access to an other function's scope from an inner function.
- Closures are created every time a function is created.
- You can call a function that returns another function (let's call it function 2) and calling that returned function (function 2) again will allow that second function (function 2) to access data from the first function. called.

Scope.

- Default creating a variable makes it global.
- For local variable, use `var`, `const` for constants, `let`, or `'use strict'`.
- `let` and `const` scopes variables to the enclosing block.
- `var` scopes variable to the enclosing function.
- `var` also declares and set variable to undefined at the start of the enclosing function.
- Use `let` over `var`

Type Coercion.

- For numbers there is a bias towards floats.
- For everything else there is a bias towards strings, and strings takes precedence.

This.

- Like Python's `self`
- Don't need to list it in the argument so no `def func(self, ...)`.
- Not automatically bound to an object, you have to explicitly define this to a variable or
- call the function using the `call()` or `bind()` method.

Anonymous Functions.

- Defined by just writing `function()` with no identifier.
- The `=>` operator also creates an anonymous function but defines `this` from when they were created.

Names.

- Start with a letter followed by `[a-zA-Z0-9_]`.
- Cannot be a reserved word like `break`, etc
- Use camelCase.

Numbers.

- Everything is a double
- Casting to int can be done using the `Math` module.
- Casting to float can be done using `parseFloat` or `Number` methods and it can take in both integers as well as hexadecimals.
- Numerical errors still apply like in C and C++.

Strings.

- Unicode by default
- You can use the `String` constructor or `(2).toString()` method on the primitive number type or just implicit casting.

Booleans.

- `true` or `false`
- There are falsey types which var keywords and values that default to false. For example an empty string is false.

Equality.

- == abstract equality operator, will try and implicit type cast before evaluating.
- === strict equality operator, will evaluate if the types match as well.
- Use the strict equality operator over the abstract equality operator.
- Also `NaN != NaN` for both operators, use `isNaN(NaN)`.

Arrays.

- Arrays are object oriented and have methods like
- length, splice
- index starts at 0.
- index operator is `[]`
- Range based iteration of objects is done using `for (... of ...)`
- Range based iteration of the indices is done using `for (... in ...)`

Objects.

- Everything is an object except for Numbers, Booleans and strings
- They are like JSON format with key-value pairs.
- The keys can be either strings or regular identifier with a colon `:`
- Passed by reference.
- Object's attributes can be accessed using the object indexing operator `object["string"]` or as a direct attribute so long as it doesn't have any spaces `object.name`.

- `Object.keys(object)` will produce the keys of the object.
- Iterating though the object using `for (... in object)` will iterate through all the keys including its prototype.

Classes.

- Constructor which is the newer standard way
- Constructor functions which is the older way where everything is defined in the function's body.
- You can add a method to a prototype by creating a new attribute to the `object.prototype` attribute.

DOM manipulation.

- Change the content of a page using JS
- You can get elements by using `document.querySelector()` or get specific elements using `document.getElementBy<selector>()` where the `<selector>` is what you want.
- You can create an element using `document.createElement()` and assign predefined attributes to it.

Chapter 5: AJAX

What is AJAX

AJAX stands for **Asynchronous JavaScript and XML**, although XML has been replaced by JSON.

- Client Side
- Allows Javascript to make HTTP requests and use the results without redirecting the browser.
- Enables heavy-clients and lightweight webservices
- Can be used to avoid presentation responsibility on the webservice.
- JSON is a common replacement for XML
- Twitter.com is heavy on Ajax

AJAX is great an all but has its issues.

- You have to manage History, Back button, Bookmarks in JS
- Security: browsers heavily restrict AJAX to prevent abuse, has Same-Origin Policy
- Even more HTTP requests, consuming more CPU and RAM resources.

Making Requests

Use JS to make an HTTP request and get the content The old school way of doing that was to use:

```
new XMLHttpRequest()
```

The new way is to use the `Fetch` API

```
promise = new Request("some-url");
```

When you use the `Fetch` API it returns a `Promise` object which represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Promise

When you make a `new Request()` it returns a Request object, that Request object can be asynchronously executed by calling `fetch(request)` on the request. This will create a "pending" Promise object. Once created that request will be put into a queue waiting to be asynchronous processed. Once processed or rejected it will then run a callback function given as a argument of the `then()` method of the promise object.

`then()` can take in two callback functions, the first function being called on a successful operation, and the second function being called on a failed operation. Both callback functions can be optional, and are replaced with an identity function if not specified. The returning object from a `then()` method is another promise. This allows users to chain promises together.

Example

```
function makeAPromise(data) {
  return new Promise((resolve, reject) => {
    resolve(data);
  });
}
function promiseExample() {
  // .then returns a NEW promise
  var promise1 = makeAPromise("X");
  var promise2 = promise1.then((data2) => {
    // inner function gets called after the
    ↪ promise is resolved
    alert("data2: " + data2);
    return "Y";
  });
  // .then calls its argument with the return
  ↪ value of the
  //     previous promises's callback
  ↪ function
  var promise3 = promise2.then((data3) => {
    alert("data3: " + data3);
  });
}
```

In the `promiseExample()` the first promise is to resolve the value "X", then makes another promise to alert the user that data2 is X, where it will return the value "Y". After which is another promise to promise3 where it will alert the user with data3 is Y.

Promise dot-chaining

You can also dot-chain or cascade promises into one another.

```
  var request = new
  ↪ Request("images/fetch.gif");
  var promise1 = fetch(request);
  var promise2 =
  ↪ promise1.then(function(response) {
    console.log("Got headers!");
    return response.blob(); // return a
    ↪ promise for raw binary data
  });
  promise2.then(function(blob) {
    console.log("Got data blob!");
    var objectURL =
    ↪ URL.createObjectURL(blob);
    img.src = objectURL;
  });
}
```

is the same as

```
var request = new
↪ Request("images/fetch.gif");
fetch(request).then(function(response) {
  console.log("Got headers!");
  return response.blob(); // return a promise
  ↪ for raw binary data
}).then(function(blob) {
```

```
  console.log("Got data blob!");
  var objectURL = URL.createObjectURL(blob);
  img.src = objectURL;
});
```

Fetching JSON

This is a generic JSON GET function

```
function fetchJSON(url) {
  var request = new Request(url);
  return fetch(request).then((response) => {
    if (response.status === 200) { // OK
      return response.json(); // return a
      ↪ Promise
    } else {
      alert("Something went wrong: " +
      ↪ response.status);
    }
  });
}

var getterID; // global
// Get some JSON every second
function startGetting() {
  getterID = window.setInterval(() => { //
  ↪ callback
    var now = new Date();
    var s = 1 + (now.getSeconds() % 4); //
    ↪ remainder
    var url = s + ".json" // 1.json 2.json
    ↪ 3.json...
    fetchJSON(url).then((json) => { //
    ↪ another callback
      console.log(json); // browser turned
      ↪ the JSON into an object
      text = json.message; // it has
      ↪ properties

      ↪ document.querySelector("#ajaxy").innerText
      ↪ = text;
    });
  }, 1000); // 1 second or 1000ms
}
function stopGetting() {
  window.clearInterval(getterID);
}
```

This code will fetch a JSON object as a promise and return the content of the JSON as another promise to be process/read. The third promise is not consumed or used. The content of the JSON file is put into blockquotes using DOM manipulation. A new content is put into the blockquotes after a second or so.

```
<button type="button"
↪ onclick="startGetting()">Start
↪ Getting</button>
<button type="button"
↪ onclick="stopGetting()">Stop
↪ Getting</button>
<blockquote id="ajaxy"></blockquote>
```

Timers

- `window.setInterval` lets you run a function every so many milliseconds.
- `window.setTimeout` lets you run a function once after so many milliseconds.

Note

These timers are not accurate and are not designed to be accurate.

JSON

JSON

- JavaScript Object Notation
- Strict subset of JavaScript
- `JSON.parse` parses JSON text into an Object
- `JSON.stringify` turns an Object into JSON text

Example

```
function stringifyExample() {
  var obj = { "food":"hotdog",

              ↪ "condiments":["ketchup","mustard","che
             "sausage":"weiner"
  };
  document.querySelector("#hotdog").value =
    JSON.stringify(obj, null, " "); // pretty
    ↪ print
}
function parseExample() {
  text =
  ↪ document.querySelector("#hotdog").value;
  var newObj = JSON.parse(text);

  ↪ document.querySelector("#sausage").innerText
  ↪ = newObj.sausage;
}
```

Design with AJAX

The design goal of AJAX is to minimize AJAX requests and traffic by bundling together request, etc. Don't hook into every event, use timeouts. This was use to ease page state transition.

What are your events?

- Per user input?
- Per user commit?
- Time based?
- Per Server action?
- Polling?
- Data?
- Content oriented?
- Messages?
- Multimedia?
- Read-based (reddit)

AJAX Observer Pattern

Observer Pattern

Observer pattern is where an observable keeps a collection of observers (listeners) and notifies those observers if anything changed by sending an update message.

This works great with AJAX if the observable is held client side in a browser and the observer is client side in the browser! Go ahead!

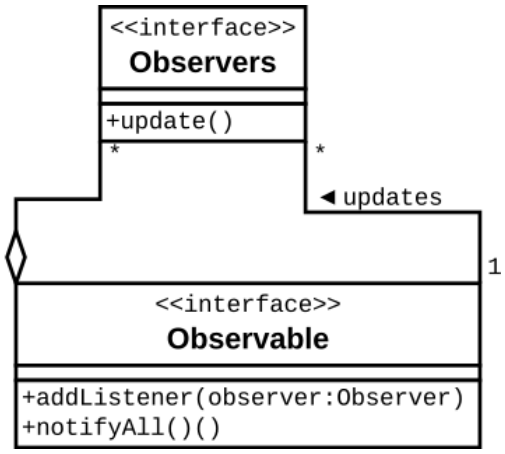

Figure 5.1: Observer Pattern UML

- Still works well with observable in browser and the observers server-side, the client simply notifies via the server's observers whenever an update occurs (but it should also communicate some lightweight state).
- Due to the lack of a client-side HTTP server it is problematic to do the observer pattern with client side observers.

Observing the Server

HTTP is stateless, so a client needs to communicate somehow all of the objects it is observing. Perhaps a serverside Observer proxy that stores observables for a client. Clients need to poll the server to check if there are updates. For the observer pattern to work this polling should allow the server to send update commands. Due to bandwidth concerns and latency concerns, an update from the server should probably include relevant data.

Fighting against:

- Latency
- Bandwidth
- Lack of communication channels
- Lack of ability to push messages to a client
- Polling
- Timer smashing

Solution?

- Polling: The most common
- Push API: Not supported on all browsers
- Comet "long polling": Difficult server-side support
- Websockets: need to make a websocket server.

Polling the Server

- Don't send too many requests
- Batch (bundle together) requests to the server
- Minimize the number of timers and the frequency of timers E.g. if drawing, a user doesn't need more than 30FPS!
- Don't make requests until the previous request finished...
- Don't make requests you don't have to

Async and Await

The `async` and `await` are JavaScript keywords that makes asynchronous code easier to write and read afterwards.

- `async` functions returns a promise
- `await` in the async function blocks code execution (stop and wait for the promise to resolve).

Example

```
var getterID; // global
async function get2() {
  var now = new Date();
  var s = 1 + (now.getSeconds() % 4); //
  ↪ remainder
  var url = s + ".json" // 1.json 2.json
  ↪ 3.json...
  var json = await fetchJSON(url);
  console.log(json); // browser turned the
  ↪ JSON into an object
  var text = json.message; // it has
  ↪ properties
  document.querySelector("#ajaxy2").innerText
  ↪ = text;
}
function startGetting2() {
  getterID2 = window.setInterval(get2, 1000);
  ↪ // 1 second or 1000ms
}
function stopGetting2() {
  window.clearInterval(getterID2);
}

<button type="button"
↪ onclick="startGetting2()">Start
↪ Getting</button>
<button type="button"
↪ onclick="stopGetting2()">Stop
↪ Getting</button>
<blockquote id="ajaxy2"></blockquote>
```

Disadvantages

- Execution stops at await, instead of continuing in parallel, although if the browser supports threads

then the waiting execution will be threaded.
- with `.then(...)` in a loop, the loop completes instantly and each callback function can run in parallel as soon as its ready
- With `await` in a loop, the loop will keep stopping each time in gets to the await.

Review

AJAX.

- Asynchronous JavaScript and XML now JSON
- Allows JS to make HTTP request and use the result without redirecting
- A lot of websites uses AJAX
- The goal of using AJAX is to minimize AJAX requests and traffic by either bundling requests together, etc

XMLHttpRequest.

- The the old school of making async requests.
- You have to setup an `onreadystatechange()` function as an event handler and
- have to process a lot of things by yourself

Request.

- Is the new, modern way of doing things.
- It will handle all the state changes and event handlers and all you need to do is implement the promise's handler or callback function.
- `new Request("url")` will create a request object that is the interface for the Fetch API.
- Calling `fetch(request)` on the request object will create a pending promise object that will be executed asynchronously.
- If the promise successfully executes, it will run the callback function given by the promise's `.then(func)` method which will then return another promise.
- You can chain or cascade promises together by successively calling `then()` after each promise.
- You can also create a normal promise by calling `new Promise(func)` where `func` is a function with two callback function parameters.
- You can put promises on timers to execute them periodically or one after a certain period of time. The timers themselves are not accurate.

JSON.

- JavaScript Object Notation
- Strict subset of JS
- `JSON.parse(text)` will turn JSON `text` into a JS object
- `JSON.stringify(object)` will turn the JS `object` into JSON text.

AJAX observer pattern.

- Where an observable keeps a collection of observers (listeners) and notifies those observers if anything changed by sending an update message.
- Works great in AJAX if both observable and observers are client side.
- Works well if the observable is client side and the observers is the server.
- Works **not so well** if the observer is the client and the observable is the server due to the lack of a client-side HTTP server.
- For observing the server you can poll, push if the browser supports it, comet or use a websocket. But you have to deal with latency, bandwidth, etc.

Async and Await.

- `async` and `await` are JavaScript keywords that makes async code easier to write and read afterwards
- `async` on a function returns a promise
- `await` within an `async` function will block until the promises resolve.
- Issues is that it stops code execution with the await unless your browser supports threading. Big issue if the await is in a loop.

Chapter 6: Websockets

Websockets

Webscokets is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. It can work on HTTP which is a good thing because JavaScript in the browser can't do TCP or UDP.

HTTP

- Traditionally, you make a request and get a response
- Problem: Not nearly as flexible as UDP and TCP
- Workarounds:
  - Polling
  - Long-polling (Comet)
  - Other tricks

We want to send messages either way whenever, we want them to arrive in order, the primary solution is to **Websockets**. **WebRTC** and **EventSource** are also valid solutions but less common.

Technology Breakdown

- Websockets!
  - Messages
  - Full duplex
- WebRTC
  - Streaming Media
  - Messages (same API as Websockets)
- EventSource
  - Sever-generated events
  - Just HTTP

You can upgrade existing HTTP connections to a websockets by using the `Upgrade` header. This allows websockets to operate on the same port as HTTP(S) and allows them to be compatible with proxies and reverse proxies.

Why Websockets

It allows for full-duplex message-based communication

- Server can push whenever, client can push whenever
- Both sides can send data at the same time
- Connection stays open
- Don't need to poll
- Avoid big HTTP headers for each message
- Reuse existing technologies as much as possible

Websocket Handshake

First the client makes a request

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: soap, wamp
Sec-WebSocket-Version: 13
```

And the server responses with

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
↪  s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: wamp
```

Use GET to specify which Websocket, 1 webserver can service multiple websocket services. *wss://server.example.com/mysocket*. `Connection: Upgrade` is used to upgrade the connection to use websockets not `keep-alive` nor `close`. `Upgrade: websocket` specifiy the protocol we're upgrading to. An optional header can be sent, `Sec-WebSocket-Protocol: wamp`, this specify the sub-protocol on what kind of websockets the user agent want to use.

Key and Accept

It uses a kind of Brown M&M test. The client sends `Sec-WebSocket-Key` with a random string. The server responses with a `Sec-WebSocket-Accept` header. This header is generated by taking the client key and appends, `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`. Then takes the SHA1 sum of the appended key. Then finally takes the base64-encoding of the SHA1 sum to generate the accept. After this the client knows for sure the server's ready for websocket.

Websocket Protocol

The client and server exchange "frames".

- The control frames (out of band): with the hex code `0x8` to close the connection, `0x9` for a ping and `0xA` for a pong.
- The data frame: These are the messages, updates, events, RPCs, whatever you're exchanging with the server.

Messages

Are sent in data frames, the messages could be binary or text, and the size if known ahead of time. This allows for efficient buffering. The messages could be fragmented into multiple data frames.

In Figure 6.1 shows the websocket format.

The first byte contains

- `FIN` bit
- 3 reserved bits
- 4-bit opcode

The second byte contains:

- A mask bit that enables XOR "encryption" mask

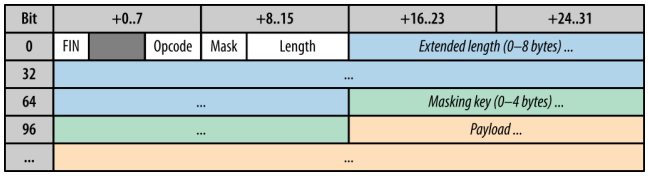

Figure 6.1: Websocket Format

- 7-bits for payload length.
- If the payload length = 126 then a 16-bit payload length follows
- If the payload length = 127 then a 64-bit payload length follows
- If masks is set then the masking key follows

Finally the data.

The smallest fragment is a one byte message, no mask equals 4 bytes. The largest possible header is 14 bytes. Each message just send fragments until FIN. Ordering is handled by TCP.

Opcodes

There are several opcodes for websockets

- `0x1` UTF-8 text (don't break characters)
- `0x2` binary
- `0x8` close
- `0x9` ping
- `0xa` pong

Example

`0x01 0x03 0x48 0x65 0x6c` => "Hel". The `0x01` tells the recipient, that the message is UTF-8. The `0x03` is the payload length of 3. `0x80 0x02 0x6c 0x6f` => "lo". The `0x80` sets the FIN bit and nothing else. The `0x02` is the payload length of 2.

Why the mask

The primary reason for the mask is to prevent accidentally being parsed as HTTP. The websocket protocol is supposed to work with existing infrastructure and maintainers were worried about cache poisoning by sending fake looking GET requests over websockets.

Masking encodes and garbles a frame with a mask so that you can't send a GET request in the plain. The mask is not there for privacy. This allows browsers to protect against malicious pages doing bad things they shouldn't. Of course we can't do anything about custom clients, which can send whatever they want over TCP.

Webscoket URIs

You can use the scheme `ws` to use websockets over a uri. For example, `ws://server.com:9090/path`. `wss:` is websocket secure, it inherits TLS from the HTTPS connection used initially. The format is the same as HTTP URI but only for the GET method.

Performance

The performance benefit for using websockets are as follows:

- Better 2-way communication
- Missing out on client side caching
- Reinvent the wheel by reusing TC/UDP
- Beat the firewall
- Doesn't fully replace XHR/fetch AJAX

Errors

During errors like a bad UTF-8 encoding this will lead to a close connection. There is no real prescript other than to close the connection. Closing is done by control frame, TLS, and TCP close.

Websockets in the Browser

JS code in the browser won't have access to fragments, masking, etc. Those are handled by the WebSocket API. The Simple browser API have the following actions

- Open
- Send and receive messages
- Close

The browser sanitizes everything, is in control to prevent malicious web pages from exploiting your browser to do things like poison proxies.

Websockets in JS

Here is an example usage for the websocket API in JavaScript.

```
var ws = new
↪  WebSocket("ws://www.example.com/socketserver");
var ws = new
↪  WebSocket("ws://www.example.com/socketserver",
↪  ["proto1", "proto2"]);

ws.send("A string");

var buffer = new ArrayBuffer(16);
var int32View = new Int32Array(buffer);
websocketInstance.send( int32View ); // send
↪  binary

ws.close();
```

You can send a plain string text or an array of binary information, your choice.

Chapter 7: Course Work and Evaluation

Monster

Kobold Lab

Small website psychic, neutral evil

Armor Class 5
Hit Points 20
Speed 1-2 weeks

STR	DEX	CON	INT	WIS	CHA
7 (–2)	15 (+2)	9 (–1)	8 (–1)	7 (–2)	8 (–1)

Senses darkvision 60 ft., passive Perception 8
Languages Python
Challenge 1/8 (25 XP)

Coverage

Everything that is being taught in the lab during the existence of the minion. Including the readings.

Composition

The total weight (or armor class) of the lab component is made up of 9 labs.

Each lab will have several questions that you must answer and upload your code onto your GitHub account and submit the link onto the lab's eclass submission.

Lab 1: Virtualenv & cURL. Introduction to CMPUT 404 labs. Setup virtualenv and understand basic usage of curl.

Lab 2: TCP Proxy. Create a tcp client, proxy server, echo server in Python. Understand how sockets work in relation to web requests. Use multiprocessing for forking new processes.

Lab 3: Common Gateway Interface. Explore the Common Gateway Interface. Refer to the CGI documentation.

Lab 4: Django. Big lab! Build a simple Django website. Understand the fundamentals of Django's MVC architecture using the built in models and views.

Lab 5: Pelican & Basic HTML/CSS. Create a static page using Pelican, or write plain html. Deploy using GitHub pages. Learn to style the basic theme.

Lab 6: Heroku. Deploy the Django application created in Lab 4 to Heroku. Understand the reasoning behind Platform as a Service (PaaS) businesses like Heroku. You may follow the official documentation.

Lab 7: Flask. Create a basic RESTful web application backend using Flask. Consume the API endpoints using cURL and httpie.

Lab 8: Websockets. Learn how to utilize WebSockets and Phaser.io. Create a basic Phaser game with WebSocket connectivity for real-time server to client communication. Use Node.js for our application server. Use TypeScript with Parcel for bundling browser client code.

Lab 9: Authentication. Learn the basics of authentication for web applications. Explore the provided Django Rest Framework applications utilizing HTTP Basic, HTTP Token, and HTTP Session authentication. Understand the high-level intention behind OAuth/OAuth2 and the security implications behind these different authentication schemes.

Archmage Assignment

Medium website psychic, neutral evil

Armor Class 35
Hit Points 35
Speed Varies

STR	DEX	CON	INT	WIS	CHA
10 (+0)	14 (+2)	12 (+1)	20 (+5)	15 (+2)	16 (+3)

Senses —
Languages Python, HTML, CSS, JavaScript
Challenge 12 (8,400 XP)

Coverage

Everything that is being taught in the lecture during the existence of the archmage. Including the readings.

Composition

The total weight (or armor class) of the assignment component is made up of 5 assignments.

Each assignment will have several user stories and requirements that the developer has to implement and upload the code onto your GitHub account and submit the link onto the assingment's eclass submission. Each assignment is worth 7% of the course.

Assignment 1: Webserver. Your task is to build a partially HTTP 1.1 compliant webserver. Your webserver will serve static content from the www directory in the same directory that you start the webserver in.

You are meant to understand the very basics of HTTP by having a hands-on ground up understanding of what it takes to have an HTTP connection.

Assignment 2: Web Client. Your task is to build a partially HTTP 1.1 compliant HTTP Client that can GET and POST to a webserver.

You are meant to understand the very basics of HTTP by having a hands-on ground up understanding of what it takes to have an HTTP connection

Assignment 3: CSS Hell. You will get experience with existing HTML and CSS and you will make some of your own as well.

Assignment 4: AJAX. Your task is to fill in the blanks and connect your browser to your webservice via AJAX. We won't be using XML here unless you want to. I recommend JSON.

Your task is to get this shared drawing program to work using AJAX.

This program allows numerous clients to connect and draw on the same surface and share the drawing they have made with anyone who is currently on.

The goal here is for you to understand the very basics of AJAX, learn a little about canvas, learn a little bit about the overhead of AJAX, learn about JSON and other technologies. Also you'll get to learn about the flask micro-framework.

Assignment 5: Webscokets. Your task is to fill in the blanks and connect your browser to your webservice via Websockets. We won't be using XML here. I recommend JSON.

Your task is to get this shared drawing program to work using Websockets.

This program allows numerous clients to connect and draw on the same surface and share the drawing they have made with anyone who is currently on.

The goal here is for you to understand the very basics of Websockets, learn a little about canvas, learn a little bit about the overhead of Websockets, learn about JSON and other technologies. Also you'll get to learn about the flask micro-framework.

You should be able to see the difference between using websockets and AJAX in this assignment.

Dragon Midterm Champion

Huge website psychic, lawful evil

Armor Class 20
Hit Points 20
Speed 50 minutes

STR	DEX	CON	INT	WIS	CHA
23 (+6)	12 (+1)	21 (+5)	18 (+4)	15 (+2)	17 (+3)

Senses blindsight 60 ft., darkvision 120 ft., passive Perception 22
Languages Python, JavaScript, HTML, CSS
Challenge 15 (13,000 XP)

Coverage

Anything that was covered in the lecture up to the midterm can be tested on the midterm.

Composition

20 multiple choice questions. The midterm is submitted on eclass.

Grade Slayer Tiamat

Gargantuan digital psychic, Lathrain of grades, chaotic evil

Armor Class 33
Hit Points 33
Speed Weeks

STR	DEX	CON	INT	WIS	CHA
30 (+10)	10 (+0)	30 (+10)	26 (+8)	26 (+8)	28 (+9)

Senses darkvision 240 ft., truesight 120 ft.
Languages Python, JavaScript, HTML, CSS
Challenge 30 (155,000 XP)

Description

The web is fundamentally interconnected and peer to peer. There's no really great reason why we should all use facebook.com or google+ or myspace or something like that. If these social networks came up with an API you could probably link between them and use the social network you wanted. Furthermore you might gain some autonomy.

Thus in the spirit of diaspora https://diasporafoundation.org/ we want to build something like diaspora but far far simpler.

This blogging/social network platform will allow the importing of other sources of posts (github, twitter, etc.) as well allow the distributing sharing of posts and content.

An author sitting on one server can aggregate the posts of their friends on other servers.

We are going to go with an inbox model where by you share posts to your friends by sending them your posts. This is similar to activity pub: https://www.w3.org/TR/activitypub/ Activity Pub is great, but too complex for a class project.

We also won't be adding much in the way of encryption or security to this platform. We're keeping it simple and restful.

Choose at least 2 other groups to work with

Composition

Project Parts. Comprised of 3+1 parts

1. Part 0 - Sign up a repo, worth 1%
2. Part 1 - 1/2 way implementation, worth 7%
3. Connect with groups, worth 5%
4. Finalize, worth 20%

RESTful APIs. There are a lot of APIs you must implement for the project, these APIs could be used to implement some of the user stories and requirements.

Take-aways.

- 1 Working website hosted on Heroku or on the lab machines/VM
- 1 GitHub repository
- 1 Presentation
- 1 Video

Credits

General

- Created by: Jihoon Og, u/DnD_Notes
- Compiled on Thursday 18th November, 2021 at 06:34
- Typesetting engine: LaTeX
- Dungeon and Dragon (5e) LaTeX Template

License

Art

- Flumph for the cover art is from D&D Beyond
- Andy the D&D Ampersand is from Dungeon and Dragons
- Book logo is from Adobe Stock
- Cover art formatting and design done in Photoshop CC 2019

Disclaimer