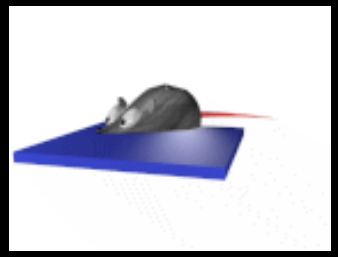
CSE 312 Midterm Review!!!

Web Applications :3



THE INTERNET

- The web is built on top of the internet, which itself is built up of different protocols.
- Internet Protocol (IP) is how we communicate between machines across the physical cables connecting them.
  - Handles routing bytes along routers via IP addresses
- Transmission Control Protocol (TCP) is built on top of IP and establishes ports to route bytes received from IP to specific processes
  - TCP also ensures that the bytes are received in the proper order and that none of them are dropped.

HTTP

- HyperText Transfer Protocol (HTTP) is built on top of TCP* and is the basis for the web.
- An HTTP server will listen for HTTP requests and return the appropriate HTTP responses.
- Example: When a web browser tries to visit a web page, it'll send an HTTP request to the server for the page, which will be returned in the form of an HTTP response containing the information the browser needs to render the page.
- Stateless!

HTTP REQUEST FORMAT

```
[Method] [Path] HTTP/[version]
Header1: Value1
Header2: Value2

[body if present]
```

HTTP REQUEST LINE: METHODS

- **GET:** Request data from the server. Idempotent. Body is insignificant (and should be empty)
- **POST:** Send data to the server; in other words, submit a resource. Body is the payload of the request. Can have side effects.
- **PUT:** Creates a new resource or replaces one. Idempotent.
- **PATCH:** Partially modify/update a resource.
- **DELETE:** Remove the specified resource. Idempotent. Body should be empty.
- **HEAD:** Like GET, but only the headers.
- Others: **CONNECT, OPTIONS**, **TRACE**
  - Not even mentioned in the slides? Don't care!

```
[Method] [Path] HTTP/[version]
Header1: Value1
Header2: Value2

[body]
```

HTTP REQUEST LINE: THE TWO OTHER THINGS

- **Path:** The location on the server of the desired resource, aka everything coming after the port in the URL.
  - May have query strings (key-value pairs appended to path)
- **Version**: The version of the HTTP protocol being spoken.
  - So far, the HTTP versions are 1.0, 1.1, 2.0, or 3.0.
  - The version being studied in this course is HTTP/1.1
- Everything in the header line is ASCII.

```
[Method] [Path] HTTP/[version]
Header1: Value1
Header2: Value2

[body]
```

HTTP RESPONSE FORMAT

```
HTTP/[version] [code] [message]
Header1: Value1
Header2: Value2

[body]
```

HTTP RESPONSES

- Sent from the server to client after the server receives a request.
  - The server will never send data to the client unless it gets a request (client-server model)
- The status code and message indicates the outcome of the request
  - 100-level: informational, 200-level: success, 300-level: redirects, 400-level: client error, 500-level: server error
  - Examples: 200 OK, 302 Found, 403 Forbidden, 404 Not Found

```
HTTP/[version] [code] [message]
Header1: Value1
Header2: Value2

[body]
```

HTTP HEADERS

- The lines following the request/status line are headers.
- Take the form of key-values pairs separated by a colon (:) and (optionally) leading whitespace.
- Each header is separated by two characters: a carriage return (\r) and a newline (\n).
  - We call this \r\n sequence a CRLF
- Headers and their values are represented by ASCII characters.

```
[Method] [Path] HTTP/[version]
Header1: Value1
Header2: Value2

[body]
```

```
HTTP/[version] [code] [message]
Header1: Value1
Header2: Value2

[body]
```

```
[Method] [Path] HTTP/[version]\r\nHeader1: Value1\r\nHeader2: Value2\r\n\r\n[body]
```

HTTP BODY

- Responses and some request methods have a body, which is a sequence of bytes following the blank line delineating the end of the request headers.
- These are just bytes and have no guarantee of structure like the rest of the request.
  - Cannot assume these bytes represent ASCII characters like with the rest of the request
  - Cannot assume \r\n is meaningful in these bytes.
- The length of the body in bytes is indicated by the Content-Length header.
- The MIME type of the body should be indicated by the Content-Type header.

```
[Method] [Path] HTTP/[version]
Content-Length: [length]
Content-Type: [type]

[body]
```

```
HTTP/[version] [code] [message]
Content-Length: [length]
Content-Type: [type]

[body]
```

MIME TYPES

- Since the payload in HTTP messages is just bytes, the Content-Type header is used to indicate to the browser (and server) what to expect these bytes to be.
  - Examples: application/json, text/plain, video/mp4
- Many browsers implement MIME type sniffing which infers the "correct" MIME type of files even if it contradicts the Content-Type header.
  - This can open security vulnerabilities, and can be disabled by the X-Content-Options header being set to "nosniff"

COOKIES

- Since HTTP is stateless, Cookies are used to replicate having state.
- Responses can send Set-Cookie headers to tell the client to remember information in the form of key-value pairs.
  - Cookies sent by the server can contain directives, indicating more information about when the client should send the cookie and when it should be discarded.
- Any* future requests to this site after a response sets a Cookie will contain the cookies that were previously set by the server in the Cookie request header.
  - Directives are not sent as they are only important for the browser.
  - Based on the directives set, Cookies may or may not be sent.

```
GET /set-cookies HTTP/1.1
User-Agent: Mozilla/5.0
```

```
HTTP/1.1 302 Found
Set-Cookie: session=c8100039; Max-Age=7200
Set-Cookie: auth=fde7dee2; Max-Age=172800
Content-Length: 4
Content-Type: text/plain
Location: /

yeah
```

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0
Cookie: session=c8100039; auth=fde7dee2
```

DATABASES

- Web applications utilize databases to store data.
- Databases provide an API to our app to store data without having to handle it on a low level.
- Run as a separate service to our application with its own port.
  - Care should be taken not to expose this port to the outside world; users should only be able to access the database through interacting with our API.
- Vary in format:
  - MongoDB doesn't enforce structured data
  - SQL implementations enforce tables with structured columns

CRUD

- Our server's interactions with the database can be summarized with the acronym CRUD:
- **Create:** Create a new record, like a new user or new message.
  - insert_one in Mongo
- **Retrieve:** Retrieve a single record, typically we used an ID to identify a record.
  - find_one in Mongo
- **Update**: Update a record that already exists.
  - update_one in Mongo
- **Delete:** Remove an existing record (or at least, make inaccessible)
  - delete_one in Mongo
- **(List):** Retrieve all records
  - find in Mongo

API

- Users don't interact with the database directly for many reasons and will instead interact with the endpoints we control.
  - We will then handle the database operations on their behalf based on the request they send
- The request methods correspond to different operations for our server (see previous slide on methods).
  - Some methods (e.g., GET, DELETE, PUT) are required to be idempotent; that is, multiple identical requests must have the exact same result on the server.
  - POST requests are not required to be idempotent; for example, two POST requests creating identical users is permitted to result in two different users with different IDs.
  - Not abiding by these meanings is a violation of the HTTP protocol; however, a server that misuses them can still be functional. It's up to the implementer of the server to be correct.

REST API

- Set of guidelines and constraints to simplify APIs.
- Client-Server Architecture
  - Clients and servers are separate; clients interact with the server through an interface.
- Stateless
  - The client only interacts with the server and vice versa when the client queries it to.
- Cache
  - If a response can be cached, this should be indicated to avoid wasteful requests.

REST API

- Layered-System:
  - There can be layers between the server and client (e.g., VPNs, HTTPS) and usability shouldn't be impacted.
- Uniform Interface:
  - All necessary resources/information for a request to be fulfilled are contained in the request and response.
  - That is, the API should be self-contained and not reliant on anything except itself.

DOCKER

- Create lightweight containers to run services for application.
- Not a full VM*. All containers share a kernel and emulate their own file system.
- Many benefits for development:
  - Consistent environment across different machines
  - Isolates your application from the hardware running it (security)
  - Can automate starting up other necessary services via Docker Compose
  - Easier configuration; e.g., setting up a service for development and another one for production with different parameters set

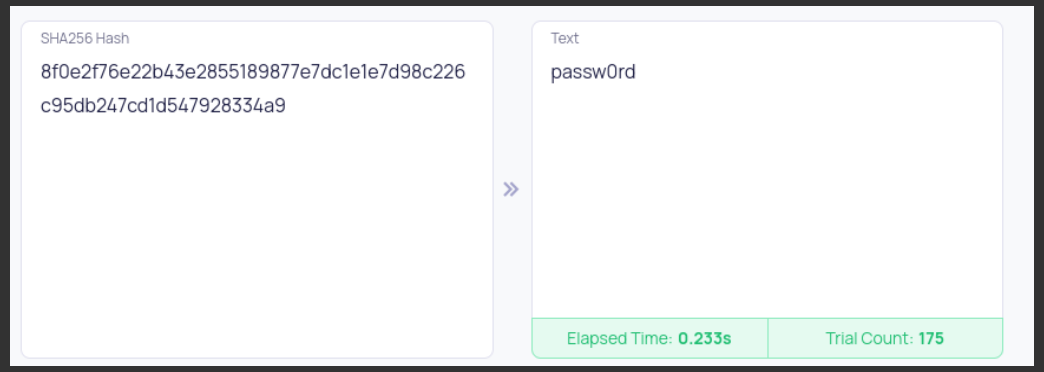
PASSWORD SECURITY

- **Hashing:** Passwords are hashed by a cryptographic hash function that is difficult to reverse, preventing passwords from being read by anyone with access to the database.
  - When a user attempts to log in, the password they provide is hashed and compared against the hash stored in the database.
- **Salting:** Random high-entropy strings are appended to the end of passwords upon registration before hashing, which allows the same password to have different hashes.
  - For example, this prevents someone from compiling a list of hashes for common passwords and using it to reverse-engineer passwords after a database leak.
  - Salts need to be stored in the database so the same hash can be computed when a user tries to log in. Since it's server-generated, there's no benefit to hashing the salt.
  - Knowing the salt doesn't make it any easier to reverse the hash.

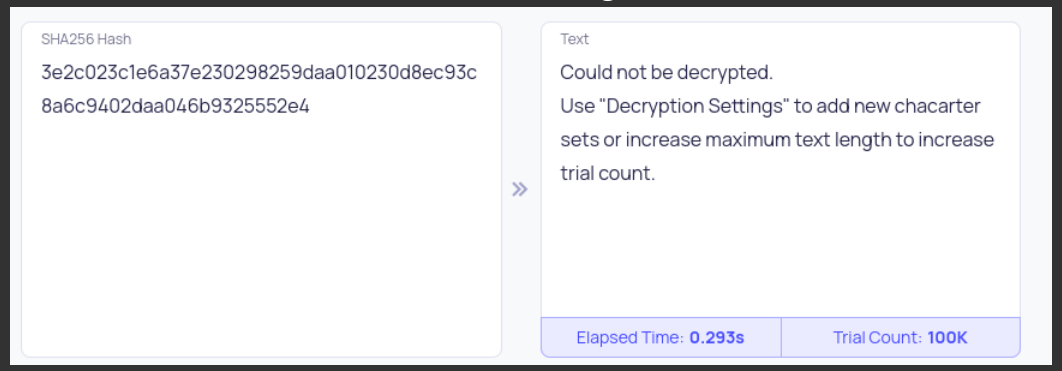
HASHING AND SALTING

1. A user creates an account with the terrible password "passw0rd".
2. Our server generates the string "tIbyrikpPOXyFk" and appends it to "passw0rd" to get "passw0rdtIbyrikpPOXyFk".
3. Our server hashes that to get "3e2c023c1e6a37e230298259daa010230d8ec93c8a6c9402daa046b9325552e4"
4. Our server stores that hash and tIbyrikpPOXyFk in that user's entry in the database.
5. When someone tries to log in to that account, the password they provide has tIbyrikpPOXyFk appended to the end of it before it is hashed
6. The hash generated in step 5 is compared to the hash from step 3; if it matches, let the user in, otherwise, do not grant access to this user.

Without Salting:



With Salting:



- With uppercase/lowercase A-Z, 0-9, and 10 special characters, the upper bound on how many tries it would take to crack the salted password is 73,686,693,514,075,155,594,934,859,127,831,072,888,456.
- The unsalted password would have an upper bound of 732,376,025,552,520 tries , but since it's not salted and all the common passwords have been precomputed, it only took 175 tries.

AUTH TOKENS

- When a user successfully logs in, we can create an identifier for their session called an authentication token.
- This is typically some high entropy string generated when the user logs in and set as a cookie.
- At the same time, we can hash this string and store it in our database. Future requests from this user will have this authentication token in the Cookies, so we can hash it and compare it against what we have in the database to authenticate them rather than requiring a password.
  - The unhashed auth token should not be stored in the server, and since we generate the auth token there's no reason to salt it.
- **Security:** Since this grants access to a user account, it needs to be treated carefully: hashed in the database, appropriate directives set (Max-Age, httponly). Old auth tokens need to be invalidated server-side to prevent impersonation.

XSRF

- Other sites can make requests to your application, which without protections our server would have no way of differentiating from requests made by our application.
  - Example: You could have a POST route to transfer money from a user's account; another website could use AJAX to access that route.
- Preexisting protections:
  - Referrer header indicates where the request came from, but this can be spoofed.
  - SOP stops these requests from being made on the client-side; but allows all GET requests and POST requests from form submissions.
  - SameSite cookie directive determines when cookies are sent on 3rd party requests, but can affect usability of your app.

XSRF TOKENS

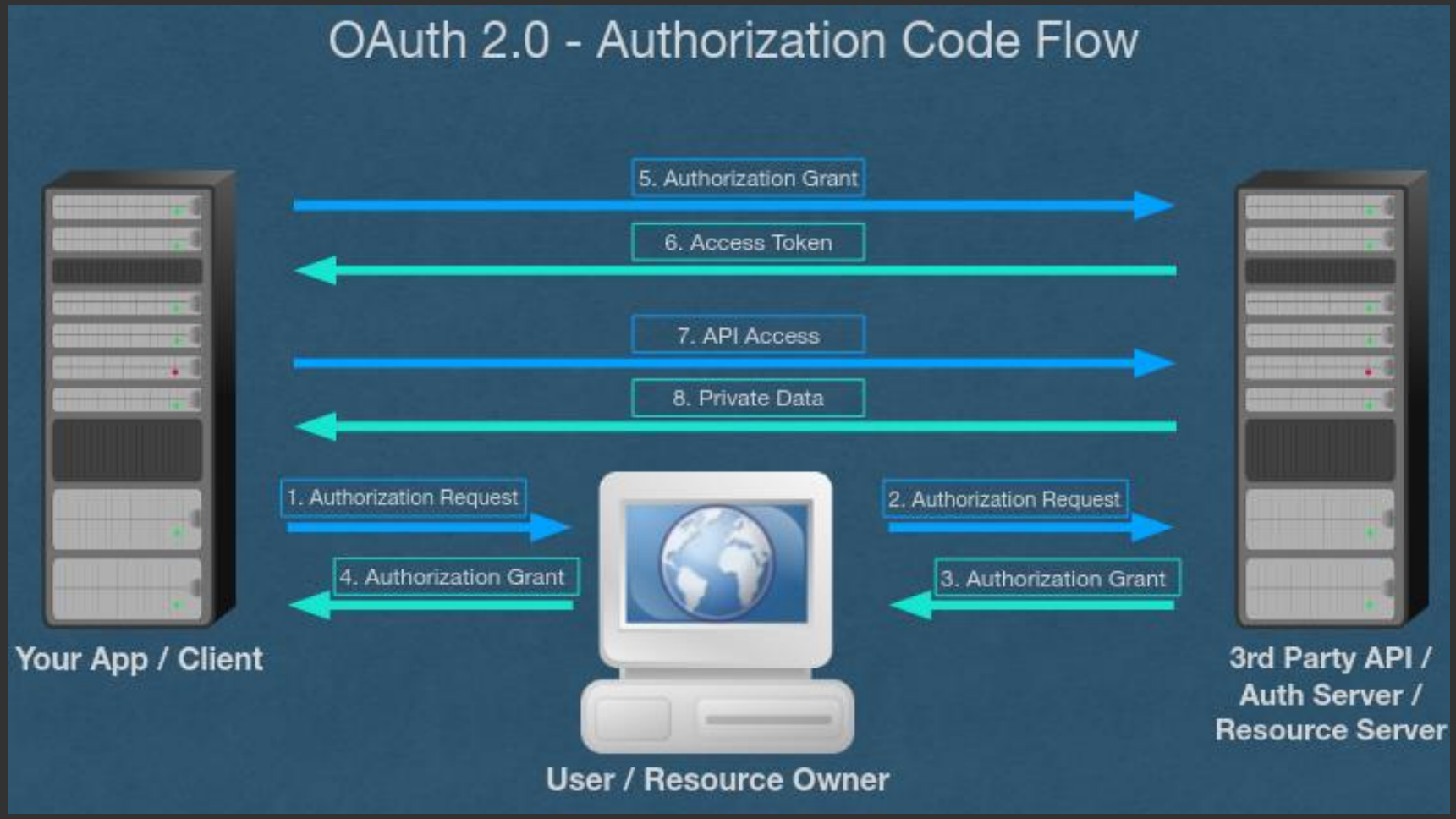
- On page load, generate a long high entropy token.
- Embed this in the page and store it in your database associated with the user.
- Have the frontend send this token on form submissions (by adding it as a hidden input to the form) to verify that these requests truly came from us.
- If you want to allow cross-site requests (like if you're making an API that other apps can use) you can do so through CORS by setting the Access-Control-Allow-Origin header

OAUTH 2.0

- Standard to safely use 3rd party APIs to act on a user's behalf.
  - E.g., log in with Google, track Spotify listening
- When you register your app with an OAuth provider, provide them a location to redirect users to and they will provide you with a client ID (public) and a client secret (...secret)

OAUTH 2.0 SUMMARIZED PROCESS

1. You ask the user to authorize with the provider
2. The user authorizes with the provider and permits access to the requested scopes
3. The user gets a grant containing an authorization code
4. The user provides you with the grant via the redirect URI
5. You (not the user!) use the authorization code the user provided to request the user's access token
6. The provider sends you a response containing the access token
7. You now can use this access token to make requests on their behalf. (But never let this token out of your server, even to the user it corresponds to!)


OAuth 2.0 - Authorization Code Flow
5. Authorization Grant
6. Access Token
7. API Access
8. Private Data
1. Authorization Request
2. Authorization Request
4. Authorization Grant
3. Authorization Grant
Your App / Client
User / Resource Owner
3rd Party API / Auth Server / Resource Server

FILE UPLOADS

- To allow uploading files in HTML forms, we set the encoding type to "multipart/form-data"
  - Otherwise, the POST request will only contain the filenames...
- The Content-Type of the POST request will indicate a boundary for each component of the form data, which supports transmitting arbitrary streams of bytes.
- These boundaries have headers like HTTP messages, and similarly the body of the boundary is separated by a blank line (\r\n).

```
POST /form-path HTTP/1.1
Content-Length: 9937
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarycriD3u6M0UuPR1ia

------WebKitFormBoundarycriD3u6M0UuPR1ia
Content-Disposition: form-data; name="commenter"

Jesse
------WebKitFormBoundarycriD3u6M0UuPR1ia
Content-Disposition: form-data; name="upload"; filename="discord.png"
Content-Type: image/png

<bytes_of_the_file>
------WebKitFormBoundarycriD3u6M0UuPR1ia--
```

PARSING MULTIPART

- Similar considerations to parsing HTTP messages should be made when parsing multipart as well:
  - The contents of each part should not be decoded until they've been separated from the rest of the request, and you know what the encoding type is.
  - \r\n is only significant in the headers! In the body of the parts in non-text-based encodings, these are just bytes and not necessarily less likely to appear than any other sequence of two bytes.
  - Decoding images as UTF-8 will most likely make it permanently unreadable.

```
POST /form-path HTTP/1.1
Content-Length: 9937
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarycriD3u6M0UuPR1ia

------WebKitFormBoundarycriD3u6M0UuPR1ia
Content-Disposition: form-data; name="commenter"

Jesse
------WebKitFormBoundarycriD3u6M0UuPR1ia
Content-Disposition: form-data; name="upload"; filename="discord.png"
Content-Type: image/png

<bytes_of_the_file>
------WebKitFormBoundarycriD3u6M0UuPR1ia--
```

BUFFERING

- When we read from a TCP socket, we may or may not receive the entire HTTP request in a single call to recv (especially when we allow file uploads)
- Thus, we may need to buffer requests, which entails continually calling recv until we have assembled the entire request.
- We can use the Content-Length header for this purpose (however; keep in mind this is not the length of the HTTP request; rather, it is the length of the body)

MEDIA PROCESSING

- Typically, we don't want to serve the exact files user uploads for several reasons (such as storage space).
- To that end, we typically process media to compress it or re-encode it in a consistent format based on our needs.
- FFmpeg is an open-source tool to accomplish just this, and has an extensive number of tools to process media files and is used virtually everywhere.

```
ffmpeg -i inputVideo.avi -s 640x360 -f mp4 outputVideo.mp4
```

ADAPTIVE BITRATE

- To account for the variety of internet speeds different users of the site may have, it's common practice to encode media content in multiple resolutions.
- This allows users with poor internet connections to access the content while allowing users with fast internet connections to access this content in high-quality.
- HLS and MPEG-DASH are two formats that split videos into segments and employ index files to allow resolutions to be switched on the fly.

```
#EXTM3U
#EXT-X-VERSION:7
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="group_A1",NAME="audio_1",DEFAULT=YES,URI="media_1.m3u8"
#EXT-X-STREAM-INF:BANDWIDTH=131049,RESOLUTION=540x960,CODECS="avc1.64001f,mp4a.40.2",AUDIO="group_A1"
media_0.m3u8

#EXT-X-STREAM-INF:BANDWIDTH=1131049,RESOLUTION=322x572,CODECS="avc1.64001e,mp4a.40.2",AUDIO="group_A1"
media_2.m3u8
```

WEBSOCKETS

- Enable two-way real time communication between the client and server on the web
  - Through HTTP, the server can only send data to the client in response to a request, WebSockets don't have this same restriction!
- Once the client requests to use the WebSocket protocol, the client and server engage in a handshake establishing this connection.
- After this, the connection between the client and server is kept open and messages are free to flow.

WEBSOCKET HANDSHAKE

Client Request:

```
GET /websocket HTTP/1.1
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
```

sha1(key + "258EAFA5-E914-47DA-95CA-C5AB0DC85B11") = 0xB37A4F2CC0624F1690F6406CF385945B2BEC4EA
base64(0xB37A4F2CC0624F1690F6406CF385945B2BEC4EA) = "s3pPLMBiTxaQ9kYGzzhZRbK+xOo="

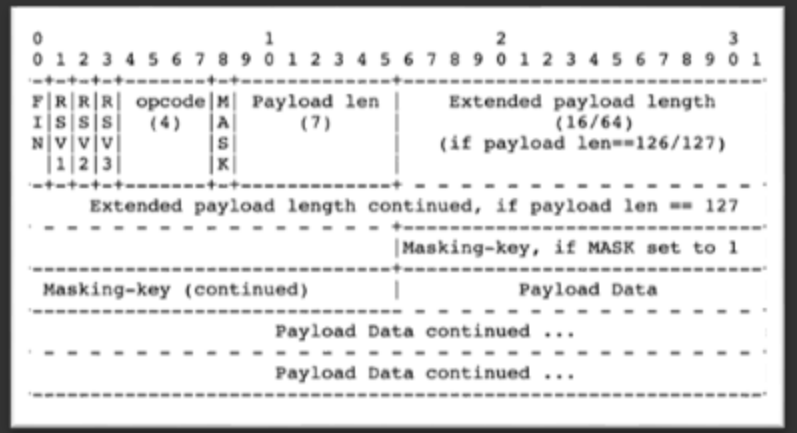
Server Response:

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


Example Source: MDN Docs

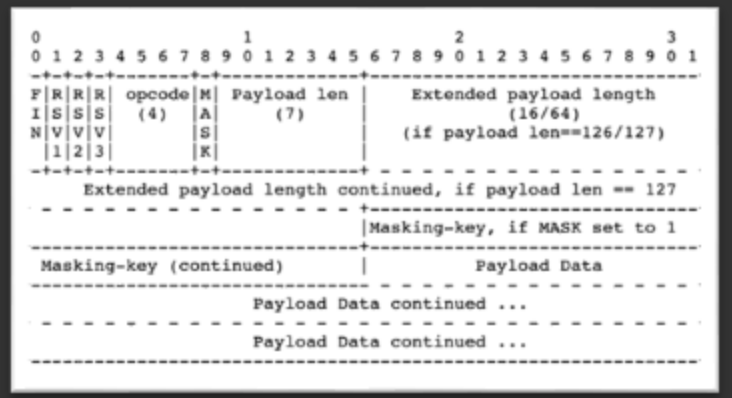
WEBSOCKET FRAMES

- WebSocket frames have much less overhead than HTTP
- Specify information including:
  - Whether or not this is the last frame of the message
  - Type of message (text, binary, close, ping/pong)
  - The length of the payload
  - If there is a mask, and what the masking key is



WEBSOCKET LENGTH

- If the 7 bits following the mask bit are less than 126, this is the number of bytes in the payload of this frame.
- If the 7 bits following the mask bit are exactly 126, the number of bytes in the payload are determined by the following two bytes.
- If the 7 bits following the mask bit are exactly 127, the number of bytes in the payload are determined by the following eight bytes.
- These bytes are big-endian.
- With a maximum length of 9,223,372,036,854,775,807, servers must also buffer bytes received from WebSockets as with HTTP messages as they can be larger than a call to recv.



PAYLOAD MASK

- If the MASK bit is 1, then there will be a four-byte mask key following the payload length.
- To parse the payload data, the payload data should be read in chunks of four-bytes and XORed with the mask as it's reassembled.