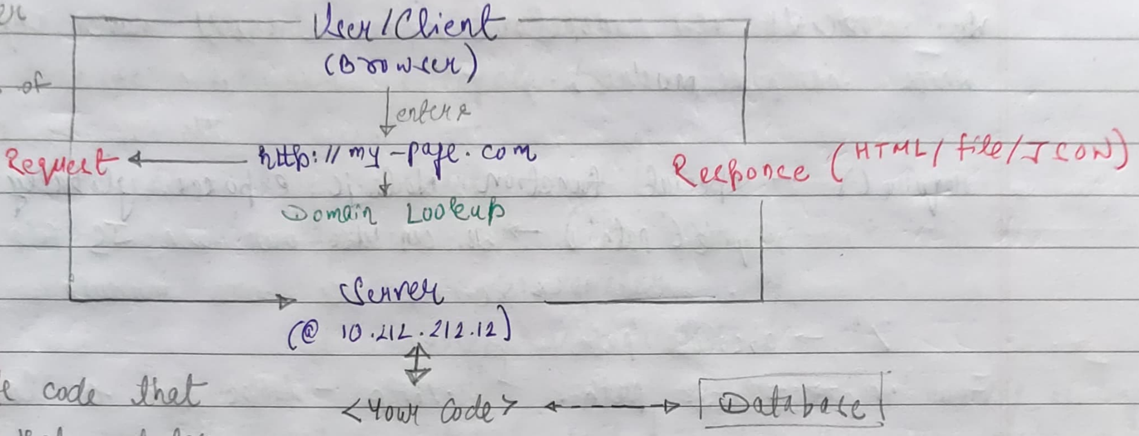
User enters some url into the browser, browser reaches out to some domain name servers to look that domain up.
As this domain entered is not address of server but a human readable form of it.

② Section II : Node.js basics

→ In the end, we enter url and it leads to some server having some IP Address

How the web works ?

browser, thus sends a request to that server with given IP address of domain



We write code that runs on that computer in the Internet which has IP address
We write code that spins out server which is able to handle incoming request and do something with it.

Communicating with the DB, that typically runs on separate DB server, but we reach out to from our backend ie server-side code.

The 'request' and 'response' transmission is done through some protocols. (standardized way of communicating).

HTTP
how a valid request looks like and how data should be transferred from Browser ⟷ Server.

HTTPS
Same as HTTP with SSL encryption turned on ie all data being transmitted is encrypted.

Creating new server →
Core Modules (Node.js)

Node app can send request to another server ie multiple servers communicate with each other.

helpful when working with request and responses
- http – launch a server, send requests
- https – launch a SSL server
- fs – managing file system

To use http module, we need to import it → ie make sure we can use features from http module which node.js ships with, but is still not available globally by default.

Date __/__/__

Path - helps to construct paths to files that can work in any OS (Windows, Mac, Linux).

os - os related informating.

We create a new constant (as we will never change whats inside module).

require () - special function Node.js exposes globally.

require ('path') → we can use our own JS file also
↳ we can use core module.

'/' → absolute path
'./' → relative path.
} ./http - looks for a local file http.

* If we omit / or ./, it will not look for local file even if we had one, but for a global module.

function createServer (requestListener ? (request, response) => void)

Node.js automatically gives us some object that represents the incoming request, and allows us to read data from that request.

↳ a function that will execute for every incoming request.

] Also gives us a response object with wh we can return a response.

```
                        anonymous function
http.createServer (function (req, res) {   → Event-driven
             if x happens, do Y        ←  architechture Node
       (incoming request, execute function)  uses extensively
});
```

Arrow function based.

```
http.createServer((req, res) => {   } createServer() callback
    log (req);                      } function,
});                                    called by Node.js when
                                       request reaches our serv
```

Nothing happens as we did not send a req to the serve we don't know address of server.

createServer () returns a server.

We use listen on this server to specify address of it.

listen starts a process, where node will not immediately exit our script, but instead keep this running to listen for incoming requests.

Page No.

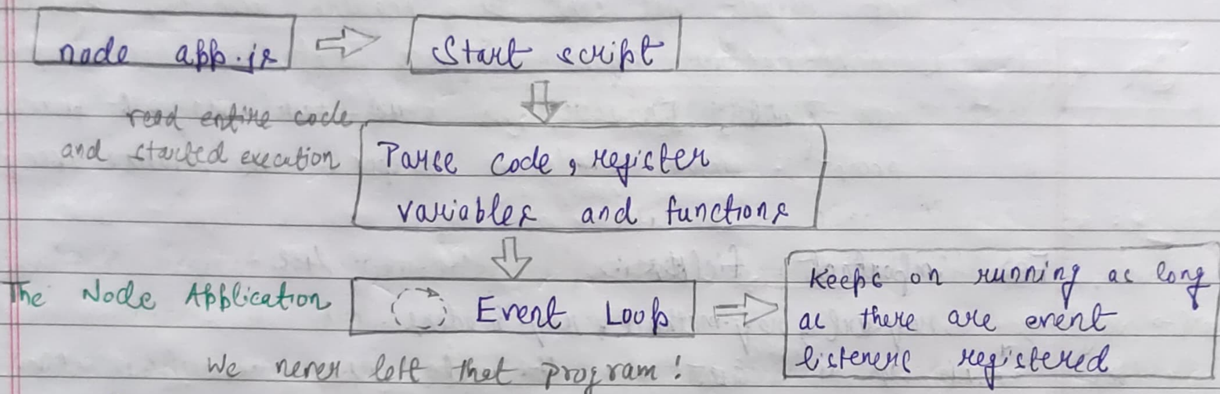
listen (port) → production → default of port 80
local environment → we can define our port / hostname (localhost)

node app.js

Date __/__/__

cursor in terminal does not go to a new line, as the process is now still running - it didn't finish.
As we now have an ongoing looping process where it keeps on listening for requests.

Node.js Program Lifecycle →



node app.js
Start script
~~read entire code~~ and started execution
Parse code, register variables and functions
The Node Application
Event Loop
Keeps on running as long as there are event listeners registered
We never left that program!

A loop process managed by Node.js which keeps on running as long as there is work to do.

We passed a function to create server → that is basically an ongoing event listener, one we didn't unregister from & we shouldn't as it is a server & must stay up and running.

∴ Our core node application basically is managed by this event loop.
Node.js uses an event-driven approach for all kind of stuff.
↳ DB → send data request - register some function once request is fulfilled.
Node uses such a pattern because it actually executes single-threaded JS.
↳ entire node process basically uses one thread on our computer it's running on.

JS does some multi-threading by leveraging the OS.

server must handle 100-1000s of incoming requests - if it will pause & do something with each request - not that great.
Thus, event loop concept wherein it always keeps on running & executes code whenever request/event occurs.
(i.e always available) To unregister, we can use process.exit().

Page No.

Saathi

Date __ / __ / ____

Understanding Requests →

console.log(req);

↳ object generated by node.js with all the data of incoming request.

headers → metadata / meta information added to request and responses.

host request was sent to, headers attached by Browser

(cache-control : how response data should be cached,

user-agent : browser we used for request,

accept : which kind of response we would accept.)

Few important fields :

url → everything after localhost:3000

req.url, req.method, req.headers.

/ GET metadata.

Sending Responses →

res does not hold any useful data.

Instead, we use it to fill it with data we want to send back.

res.setHeader('Content-Type', 'text/html');

(default header which browser knows) ↳ attaches header to response.

(*) Only a certain set of supported headers the browser understands

res.write('<html>');

:

write some data to response

↳ works in chunks / multiple lines res.write('</html>');

To send the response, res.end();

After res.end(), no more res.write();

(done after setting all headers and loading data to response body)

Routing Requests →

Connecting request and response.

'/' → load page where user enters data & we store in a file on the server.

↳ Parsing the url.

Page No.

Date ___/___/___

We want to return response that holds some html that gives user an input form & button that sends a new request in return & : a GET request.

<button type="submit">

↳ default HTML behaviour we use here where a button having this type in a form will send a new request.

<form action=" ___ "> method="POST">

↳ url the request will be generated automatically will be sent to.

(*) GET request is automatically sent when we click a link/ visit a url.

POST request must be set up by us by creating form or other ways.

Form not only sends request automatically, but also looks into the form to detect any input/related element, and puts that into the request sent.

→ Adds any input data to the request, and make it accessible via assigned name to the tag.

Redirecting Request →

url = /message & we're handling a POST request.

(↳ parse method using req.method)

Here → 1. store message entered by user in a new file

2. redirect user back to '/'

```
fs.writeFileSync('message.txt', 'DUMMY');
res.statusCode = 302;   → for redirecting
res.setHeader('Location', '/');
return res.end();
```

Parsing Request Body →

Time to parse the incoming request, and get the data that is part of the request as that data should be one entered by user.

Nothing like req.data exists. | Our stream is an ongoing process.

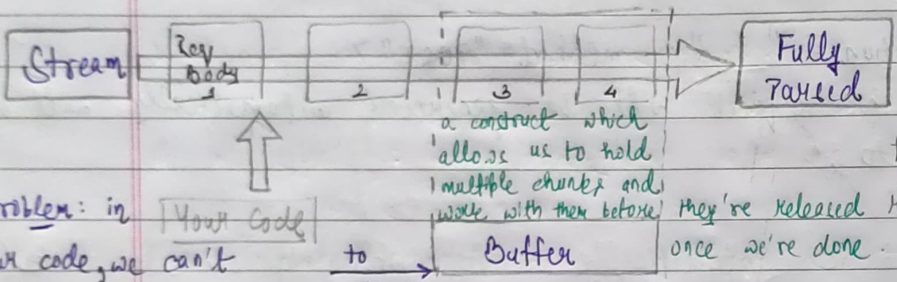
The request is read by Node in chunks (multiple parts) and in the end it's done parsing.

Saathi

Date ___/___/___

Instead, incoming data is sent as a stream of data, and that is a special construct JS knows but Node uses a lot.

Stream and Buffers. other streams →
ex: Incoming Request   working with files.



Stream
Req Body
1
2
3
4
Fully Parsed
Your Code
a construct which allows us to hold multiple chunks and work with them before they're released once we're done.
Buffer
to organize chunks

This is done so that we can start working on each chunk, without having to wait for full request being read.

Problem: in our code, we can't arbitrarily try to work with these chunks.

like a bus stop
to climb/leave the ~~bus~~ (always moving), we need a stop.

Note:
For a simple request (like ours), this is not really needed. (As we've only one input field data).
Consider a file being uploaded. It takes longer and thus streaming that data makes sense because it could allow you to start writing it to our disk (harddrive on server) whilst data is coming in.
So that we don't have to parse entire file, and we have to wait for it being fully uploaded before we can do anything with it.

This is how Node handles all requests as it does not know in advance how complex and big they are.
→ ie we can start working on data early.

We go to req and register an event listener.
req.on() allows us to listen to certain events (here, data)
data event will be fired whenever a new chunk is ready to be read. req.on('data', () => { callback fxn });

Buffer available globally by Node

end event fired once it's done parsing the incoming requests/data

Page No.

log (chunk) → <Buffer 6d 65 73 73 61 [illegible] >

log (parsed body) → message = fasd fasdf
(key = value) pair] automatic parsing of input field using HTML forms.

Date ___/___/___

Saathi

Event Driven Code Execution → Order of execution ≠ Order in which we write code

```
req.on('end', () => {
        :
    fs.writeFileSync(...)   → Executes after this i.e even executes after we've sent a response.
})

{ res.statusCode = 302;
  = }
```

Eventually, will be executed but too late → Error: cannot set header after they are sent

Implications —

- Sending a response does not mean event listeners are dead (they still execute).
- Thus, if we do something in event listener that should influence response, it is wrong way as done above

✓ × { We should move the response code in the event listener too

Look video 10:27

With req.on / createServer, Node uses a pattern wherein we pass a function to a function, and Node will execute the passed-in function at a later point in time (Asynchronous).
↳ Node uses this pattern extensively.

Node manages all these listeners internally (end / data / ...)
Once it is done parsing request, it is triggered automatically
ie Node has some internal registry of events and listeners to these events.

↳ Once it has parsed through request, it goes through registry & looks for attached listeners
↳ but it does not pause other code execution

Blocking / Non-blocking code →

writeFile	v/s writeFileSync (synchronous)
∴ We must use writeFile (path, data, callback) receives an error object	special method that blocks code execution until this file (message.txt) is created. blocks execution of next line of code until file is done.

file until operation is done

Not good for large files (new incoming requests, other work will not be handled)