



BACKEND DEVELOPEMENT - BY ANGELA YU

SPECIAL THANKS TO **NINAD DHULAP**

INDEX

- *UNIX Command Lines*
- *Backend Development*
- *Node.js*
- *Express.js*
- *API-Application Programming Interfaces*
- *Git, GitHub and Version Control*
- *EJS - Embedded JavaScript templating*
- *Databases*
- *SQL DB*
- *MongoDB*
- *Mongoose*
- *Build your own RESTful API from scratch*
- *Authentication & Security*

UNIX Command Lines

Kernel: In computing the kernel refers to the actual program that interfaces with the hardware. So it's the core of your operating system.

Shell: shell in computing refers to the user interface for you as a human to be able to interact with the kernel and in turn with the hardware of your computer.

There are two variants in shell: 1) graphical user

interface

2) command line interface

Bash stands for **Bourne Again Shell**. Is a command line interpreter for unix system.

Unix like systems: ex: Linux, Mac OS,

But windows use DOS (Disk Operating System)

For better control we use command lines.

\$ called as prompt

Some Command lines:

[illegible]

~ shows you that you are in your user directory. This is a starting point when you open terminal.

If you type enough letters of folder then press tab to automatically gets folder name.

Alt + click / arrow to directly go to letter you want to edit in path.

Up and down arrow to see history

If you want to go at first letter of entire path then use ctrl + A and for end letter use ctrl + B

Ctrl + U to clear entire line before execution

Great power comes with great responsibility so be aware when removing all files in folder You check your path with pwd before executing commands

Extra Knowledge: sudo command stands for super user do, which uses your admin privileges to basically allow you to do things are probably a little bit dangerous

Sudo rm -rf --no-preserve-root/

Rm -rf here f means force means it doesn't ask for confirmation

--no-preserve-root/ it wipes your hard disk to the point where you can't recover it.

Backend Development:

Backend consists of things such as,

- 1) **server** which will serve up your files your HTML, CSS and Javascript
- 2) **database** which can store your user data such as their log-ins and passwords, as well as apps you determine how your web app works.

Eg. calculating flight prices, or making payments

Client-side ← server ← database

Backend technologies: node Js, ruby, php, java

Frameworks: cake php, ruby rails, nodejs express, ...

Why Backend?

We don't want to show our processing code to users. Ex. Chef not cooking in front of customers and doesn't show their secret recipe and it also time consuming to send all data on computer and process it so it is better to process on the server and then output is served to you directly.

NodeJs:

NodeJS interacts with databases or have business logic on a server. It's a low level technology that interact directly with hardware.

NodeJs allows us to take Javascript out of the browser and liberates it, allowing to interact directly with the hardware of a computer. So we can use node to make desktop app (Ex. Atom)

We can also use nodeJs to run JS code on somebody else's computer or server.

REPL: REPL stands for Read Evaluation Print Loop

It allows you to execute code in byte sized chunks. Means you can run single lines of code (like in chrome dev tools).

To access node REPL use 'node' inside command line and enter.

```
$ node
Welcome to Node.js v18.12.1.
Type ".help" for more information.
> console.log("Hey there! I am using Node REPL :)");
Hey there! I am using Node REPL :)
undefined
```

If you double TAB after writing few letters of command it shows you all possible commands.

To exit REPL use '.exit' / CTRL + C twice.

So by using node Js we're using javascript independently from browser.

In order to use module we've to first require it.

Var creates variable which can vary/ changeable, but const stands for constant and doesn't change value.

Ex. You can use const for pi → `const pi = 3.14`

NPM package manager for external module (library .

Initialize NPM by `npm init`

Json is a file format used to organize data

Express Js:

Is a framework of nodeJs. So it saves our time of writing repetitive code in NodeJs.

'--save' it adds Express into packages.json under the dependencies. But this part isn't required it is taken care by default.

Use -> Npm install express

After installing express 1st step is to require express. And next step is to create const called 'app' this function represents express module.

```
const express = require("express");  
const app = express();
```

Now you can use method 'listen' tells it to listen on a specific port for any HTTP requests that get sent to our server.

Port: port is basically a channel that we've tuned our server to.

CNTR + C to stop port.

```
app.listen(3000);
```

localhost:3000/ is our homepage. And that we will call route.

```
app.listen(3000, function() {  
  console.log("Server started on port 3000 :");  
});
```

```
$ node server.js  
Server started on port 3000 :)
```


How it works?: when we load up a web site google.com, then our browser will send a request to Google's servers to get some data for this location, and Google servers, when it sees that request, it will send our browser a response, and that response includes the HTML, the CSS and the Javascript that's needed to render website.

app.get(): allows us to specify what should happen when a browser gets in touch with our server and makes a get request.

```
app.get("/", function(request, response){  
  })
```

This defines what should happen when someone makes get request to home route

And following callback function tells what should happen.

```
const express = require("express");  
const app = express();  
app.get("/", function(request, response) {  
  response.send("<strong>Hello :)</strong>");  
});  
app.listen(3000, function() {  
  console.log("Server started on port 3000 :)");  
});
```

response.send()
to send
response.

Make many routes:

```
app.get("/contact", function(req, res){  
  res.send("Contact me at: theninad17@gmail.com");  
});  
  
app.get("/about", function(req, res) {  
  res.send("This website is owned by The great Ninad w");  
});  
  
app.get("/hobbies", function(req, res) {  
  res.send("listen BTS and Kpop and watch Kdramas.");  
});
```

Nodemon: To restart server automatically use nodemon server.js (after installing nodemon).

[If nodemon not working →](#)

To send full file (html, css..) when homepage called we have to use 'res.sendFile(

When we deploy our server we've no idea about exact location of file therefore instead of sending relative file path (like 'index.html' only) we use constant called `__dirname + "/index.html"`

`__dirname` shows current file's location.

If you want to see `__dirname` path then just console.log it. You didn't need to find location of your root in server, because it is done by `__dirname`.

Ex.

```
$ node path.js
N:\NINAD --{ NAM DO-SAN }--\DEVELOPEMENT\BACKEND DEVELOPMENT\Calculator
```

```
app.get("/", function(req, res) {
  res.sendFile(__dirname + "/index.html");
});
```

Path: to our project folder/ index.html

Action means location where we're going to send post req to home route, method post means send data

```
<form action="/" method="post">
  <input type="text" name="num1" placeholder="1st Number">
  <input type="text" name="num2" placeholder="2nd Number">
  <button type="submit" name="submit">Calculate</button>
</form>
```

Name= "..." is a variable name.

app.post: To handle post request use app.post

```
app.post("/", function(req, res) {  
  res.send("Thanks for Posting. ..|.");  
})
```

Body Parser:

Install body-parser: npm install body-parser

It allow us to parse the information that we get sent from the post request.

bodyParser.text: to parse all request into text.

bodyParser.json: to parse request into json format.

bodyParser.urlencoded: whenever you're trying to grab the information that gets posted to your server from an HTML form use this.

```
app.use(bodyParser.urlencoded({extended: true}));
```

Body Parser allows you to go into any of your routes, and you can tap into something called **request.body**, and this is the parsed version of the HTTP request.

```
const query = req.body.location;
```

by using Body Parser, we're able to parse the HTTP request that we get

by using urlencoded we can get access to the form data, and we can then tap into each of these

```
app.post("/", function(req, res) {  
  console.log(req.body);  
  res.send("Thanks for Posting. ..|.");  
});
```

```
Sever is running on port 3000.  
{ num1: '12', num2: '12', submit: '' }
```

Calculator App:

```
var num1 = req.body.num1;  
var num2 = req.body.num2;
```

req.body.num1 is parsed in text format so we've to explicitly convert it to number format

as `Number(req.body.num1)`

```
app.post("/", function(req, res) {  
  var num1 = Number(req.body.num1);  
  var num2 = Number(req.body.num2);  
  var result = num1 + num2;  
  res.send("The result of calculation is: " + result);  
});
```

Because of using Calculator code in backend (server side) we hide this code from user and only simple html code is visible to user when we see code in browser by 'view source code'.

API-Application Programming Interfaces

APIs - API is a set of commands, functions, protocols and objects that programmers can use to create software or interact with an external system.

API providers allows users to access some data but certain data aren't accessible to you. So they tell you what data you can access.

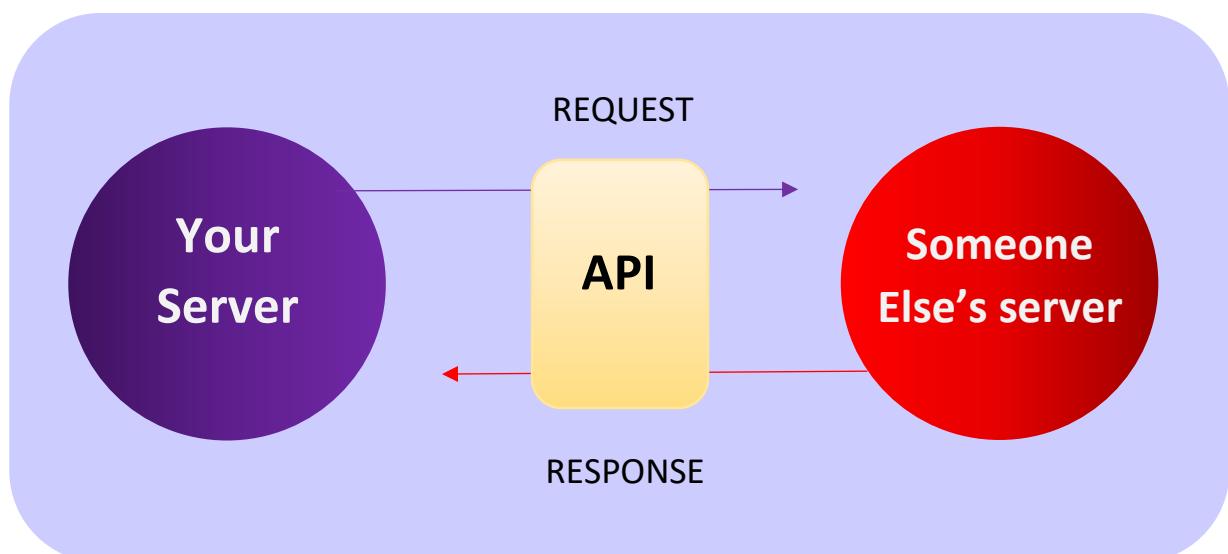
Ex. Weather API: temperature, weather condition, img etc.

It is like contract between developer and provider as, these are all the things that developers can access, and these are the methods, the objects, the protocols that you would use to access them.

Provider has responsibility to not change any method/function before notify to you(developer).

API allows you to interact with external system, create software.

Jquery is an API that allows us to create software.



API:

▪ Endpoint

Every API that interacts with external system has endpoint.

Ex. <https://api.kanye.rest/>

▪ Paths

Paths and Parameters use to narrow down on a specific piece of data you want from an external server.

In API which contains many categories, we need to **add path** to specific category after **endpoint**.

Ex. <https://v2.jokeapi.dev/joke/Programming>

If API doesn't have path for specific query API allow you parameters (API becomes flexible to deal with custom queries.)

▪ Parameters

Parameters goes at end of URL after '?' then key-value pair goes into url. (**key-value** called **contains-query**)

If we've more than one query/parameters →

1st query follows a '?' and every subsequent query follows an ampersand(&)

<https://v2.jokeapi.dev/joke/Programming?blacklistFlags=nsfw&type=single&contains=debugging>

▪ Authentication

API provider keeps track on how you using their to get data server and charge you/ limit you accordingly.

We get a unique API key.

Use Postman to test API parameters (Install Postman).

API is like restaurant menu that shows what they'll provide to you and serve to you you can pass various parameters in it also (such as in restaurant if you want half quantity of particular dish then you've to say).

Get request = Client browser → your server → someone else's server (if your website is taking data from another server).

Response = someone else's server → your server → client browser (Html/css/js files are response)

JSON - JavaScript Object Notation

We use json because it's in a format that can be readable by a human, but it can also be easily collapsed down to take up as little space as possible.

It is similar like JS objects(wardrobe) to transport we use json(pack furniture at IKEA) i.e we collapse all data into single string, once we receive json as a string we build it back up to original object.

In addition to json there are XML, HTML also used. But json is favoured format as it is lighter weight and easily convert to js object.

How to make request to external server:

There are many methods. But http module is native node module

https module:

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

https.get() to get data from somebody else's server.

```
const url = "https://api.openweathermap.org/data/2.5/weather";
https.get(url, function(response) {
  console.log(response);
});
```

In url "https://" is must to write.

```
console.log(response.statusCode);
```

You can use . after response to access specific data.

We can also use response.on to get data

```
https.get(url, function(response) {
  console.log(response.statusCode);
  response.on("data", function(data) {
    console.log(data);
  });
});
```

Here data you get is in hexadecimal code format,

```
200
<Buffer 7b 22 63 6f 6f 72 64 22 3a 7b 22 6c 6f 6e 22 3a 31 32 36 2e 39 37 37 38 2c 22 6c 61 74 22 3a
65 61 74 68 65 72 22 3a ... 469 more bytes>
```

If we convert this code then it is same as json which we get from console.log(response). [Convert hex →](#)

To parse data from hex, bin, txt to JSON use `JSON.parse(...)`

```
const weatherData = JSON.parse(data);  
console.log(weatherData);
```

To turn JS object into string format use `JSON.stringify(...)`

```
const object = {  
  name: "Ninad",  
  food: "Ramen"  
}  
console.log(JSON.stringify(object));
```

```
{"name": "Ninad", "food": "Ramen"}
```

How to get specific data?

If you want temperature data from below data,

```
{  
  coord: { lon: 126.9778, lat: 37.5683 },  
  weather: [ { id: 800, main: 'Clear', description: 'clear sky', icon: '01n' } ],  
  base: 'stations',  
  main: {  
    temp: -4.62,  
    feels_like: -8.58,  
    temp_min: -6.24,  
    temp_max: -3.22,  
    pressure: 1031,  
    humidity: 39  
  },  
}
```

Use,

```
const temp = weatherData.main.temp;  
console.log(temp);
```

data → main → temp

if the data is big and you're not able to find correct path (data → main → data) then use extension [json viewer pro](#) and copy path from it after selecting particular data.

You can able to add `res.send` only once in each app method.

```
res.send("<h1>The temparature of Seoul is " + temp
```

By above method you can only able to send one data.

To send many use `res.write();`

```
res.write("<h1>The temperature of Seoul is " + temp + " degree C");
res.write("<h4>Weather Description: " + weatherDes + " </h4>");
res.write("");
res.send();
```

Using Axios:

```
app.get('/api', async (req, res) => {
  console.log(req._parsedUrl.query)
  let url = `https://newsapi.org/v2/everything?${`
  let r = await axios(url)
  let a = r.data
  res.json(a);
})
```

Make your function async to know our app that it's asynchronous function. And then use await while using api request.

```
// Making API call to newsapi.org
let response = await axios.get(url);
let data = response.data;
```

Promise.all():

To make concurrent/ parallel api calls use Promise.all()

```
const requests = urls.map((url) => axios.get(url));
const response = await Promise.all(requests); // Res

const data = response.map((res) => res.data); // Get
console.log(data);
```

Refer links: [1>>](#) , [2>>](#) , [3>>](#)

MailChimp API project:

In order for our server to serve up static files such as CSS and images then you have to use special function of express called static.

```
app.use(express.static("public"));
```

Public is a folder where static files are exist

Ex. styles.css, image...

Mailchimp wants data in JSON format

```
var data = {  
  members: [  
    {  
      email_address: email,  
      status: "subscribed",  
      merge_fields: {  
        FNAME: fName,  
        LNAME: lName  
      }  
    }  
  ]  
}
```

data in JS object

members is define in api document.

```
var jsonData = JSON.stringify(data);
```

To make it In string format to send data to server.

To post data in external server:

```
https.request(url, [options], [callback])
```

you store this all in const request and then request.write(jsonData) to pass data to server.

Then request.end()

If you want to redirect user to different page

```
app.post("/failure.html", function(req, res) {  
  res.redirect("/");  
})
```

Git, GitHub and Version Control

To initialize git use -> **git init**

```
$ git init
Initialized empty Git repository in N:/NINAD --{ NAM DO-SAN }--/DEVELOPEMENT/GIT & GITHUB/story/.git/
```

To see hidden git folder use -> **ls -a**

```
$ ls -a
./ ../ .git/ chapter1.txt
```

.git is used to track all your changes, to commit your changes and to perform version control.

Working directory is a directory where git is initialized.

In order to start tracking the changes of your files we need to add this file to staging area.

Staging area: Is an intermediate place where you can pick and choose which files inside your working that you want to commit.

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    chapter1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Here git status is showing that there are untracked files in working directory, but it's not yet in staging area.

Git Commands:

[illegible]

To add it in staging area and start tracking changes in it use -> **git add**

```
$ git add chapter1.txt
```

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   chapter1.txt
```

This shows that file is in staging area and it's ready to commit.

To commit file use -> **git commit -m "messege"**

-m flag is very important helps you to keep track of what changes you have made in each commit. It helps to understand what changes you've made between last save point and current save point. (write in present tense.)

```
$ git commit -m "Complete Chapter-1"
[master (root-commit) 53a18d9] Complete Chapter-1
 1 file changed, 2 insertions(+)
 create mode 100644 chapter1.txt
```

Commit is lightweight, it is basically a snapshot of the project. It doesn't just blindly copy the entire directory every time you commit. It can (when possible) compress a commit as a set of changes, or a "delta", from one version of the repository to the next.

To see what commits you've made use -> **git log**

```
$ git log
commit 53a18d9f63ec504cde93cca3027d3acc20f4445f (HEAD -> master)
Author: NINAD [REDACTED]
Date:   Sat Dec 17 13:57:33 2022 +0530

    Complete Chapter-1
```

Commit hash (in yellow colour) uniquely identifies this particular commit.

If you want to add all directory files to staging area then use '.' After command

Here we created two files chap2, chap3.txt

`$ git add .` after these command they are now in staging area.

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   chapter2.txt
        new file:   chapter3.txt
```

Staging area

```
$ git commit -m "Complete Chapter2 and Chapter3"
[master 0912208] Complete Chapter2 and Chapter3
 2 files changed, 2 insertions(+)
 create mode 100644 chapter2.txt
 create mode 100644 chapter3.txt
```

Files committed.

```
$ git log
commit 0912208fff5dca043843004a7a710054097cdfc9 (HEAD -> master)
Author: NINAD <[redacted]>
Date:   Sat Dec 17 14:09:10 2022 +0530

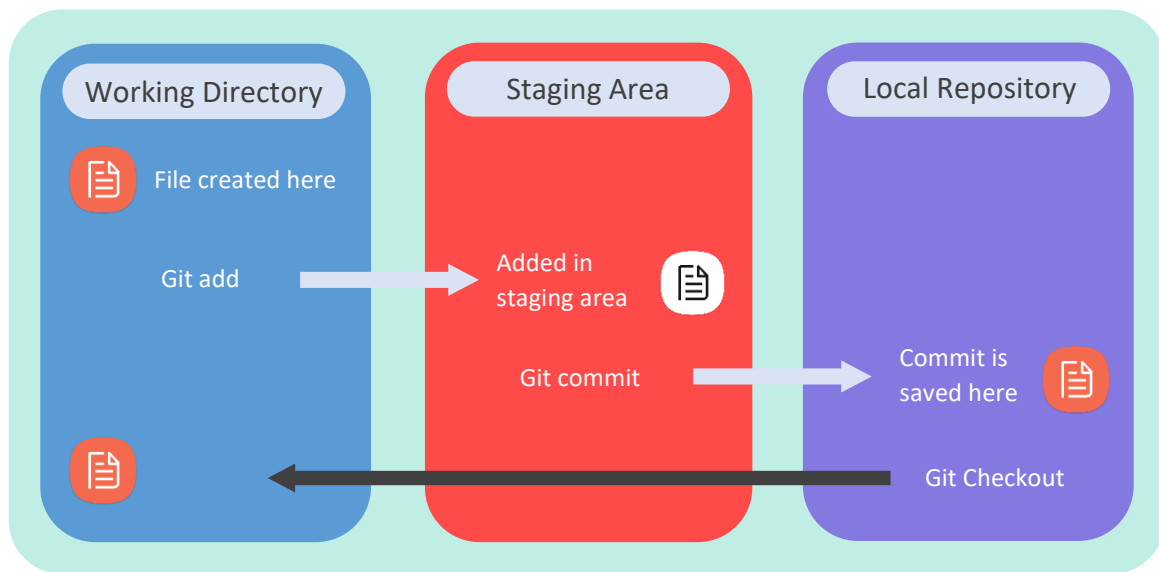
    Complete Chapter2 and Chapter3

commit 53a18d9f63ec504cde93cca3027d3acc20f4445f
Author: NINAD <[redacted]>
Date:   Sat Dec 17 13:57:33 2022 +0530

    Complete Chapter-1
```

(HEAD -> master) shows that you're currently here.

Head means position or current state that we're in.



After initializing git, it is going to try and track the changes that it sees between the working directory And the local repository.

Why staging area is require?

sometimes you might not want to add all of your files to be tracked or all of your files to be committed. So the staging area is a good place to try and figure out what are the things that you want Git to ignore and what are the thing to be tracked.

Local repository is .git and version name of file is commit message.

So that we can use last version of our file by using command `git checkout` to revert back or rollback to the last position in our local repository.


```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   chapter2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

If we modified our file and save it but not committed then if you want to revert it back,

- 1) To see diff between current saved file and one which is in repository use -> **git diff fileName**

```
$ git diff chapter3.txt
diff --git a/chapter3.txt b/chapter3.txt
index 348b4ed..b61650e 100644
--- a/chapter3.txt
+++ b/chapter3.txt
@@ -1,1 @@
-You and I best Moment is Yet to Come.
\ No newline at end of file
+jhfshfjkk
\ No newline at end of file
```

- 2) To roll back to last version in local repository Use -> **git checkout filename**

```
$ git checkout chapter3.txt
Updated 1 path from the index
```

If you want to push both commits, i.e

```
$ git log
commit 0912208fff5dca043843004a7a710054097cdfc9 (HEAD -> master)
Author: NINAD [redacted]
Date: Sat Dec 17 14:09:10 2022 +0530

    Complete Chapter2 and Chapter3

commit 53a18d9f63ec504cde93cca3027d3acc20f4445f
Author: NINAD [redacted]
Date: Sat Dec 17 13:57:33 2022 +0530

    Complete Chapter-1
```

on github then,

use -> **git remote add origin 'URL of remote repo'**

git remote is telling to local repository is that we've created remote repository on cloud and I want to transfer

all my commits over there. 'origin' is name of your remote.

```
$ git remote add origin https://github.com/NINAD-17/story.git
```

You get this url after creating repository on github.

Push local repository to remote repository(origin)

Use -> **Git push -u origin master**

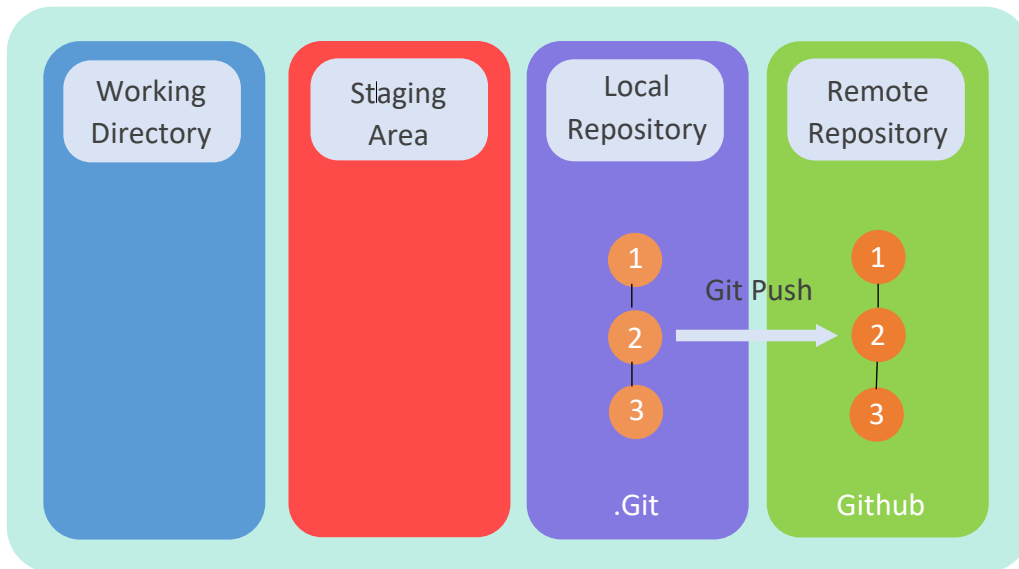
Here u option basically links up your remote and your local repositories and we're going to push it towards the remote that's called origin and we're going to push it to the branch that's called master.

Master is default or main branch of all of your commits.

On git hub repository go to insights -> network to see graph of your commits.

Master Branch: Master branch is your main branch of commits or save points and it is sequential. And this is usually where your main progress is saved or committed.

Note: you can have a local repository completely in parallel with a remote repository. Check the differences between them but you can also sync them so that they all the same.



Gitignore:

Set rules to prevent committing certain files to your local and remote Git repositories.

Ex. Api keys, passwords

Utility files, setting files...

Creating and using gitignore file

Use -> `touch .gitignore` (correct spelling is imp)

```
$ touch .gitignore
```

If we added files in staging area but if we want to undo it then use -> `git rm --cached -r`

```
$ git rm --cached -r .  
rm '.gitignore'  
rm 'file1.txt'  
rm 'file2.txt'  
rm 'file3.txt'  
rm 'secrets.txt'
```

Gitignore: add file names that you want to ignore when we're adding and committing to git.

In order to add individual files you can simply specify the file name on each and every new line.

You can use # to comment in gitignore file.

'*.extension' used to ignore all files with specific extension. Ex. *.txt

Ex. `secrets.txt`
`#comments`

After adding all files secrets.txt is ignored.

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   file1.txt
    new file:   file2.txt
    new file:   file3.txt
```

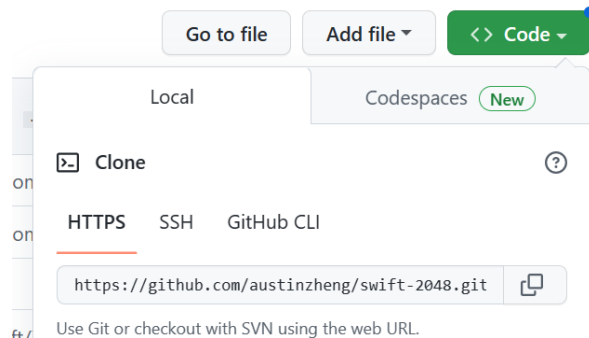
Github/gitignore: is a git repository of github which includes all coding language's files that should be ignore by user so just copy and paste it to your gitignore according to your language.

Cloning remote repository to pull it into your local machine: (like downloading all commits and versions to local machine and store in working directory)

Use -> `git clone fileLink`

Pull down all of the commits and all of the versions of a particular remote repository and to store the files inside your working directory.

If you want to clone of github project then,



copy the url

then use ->

git clone "paste url here"

```
$ git clone https://github.com/austinzheng/swift-2048.git
Cloning into 'swift-2048'...
remote: Enumerating objects: 287, done.

Receiving objects: 100% (287/287), 93.70 KiB | 710.00 KiB/s, done.
Resolving deltas: 100% (155/155), done.
```

If you clone project then you can also able to use git log to see their previous commits.

Branching and Merging:

Whenever we have to build a new feature or try/ experiment new feature It makes some bugs so instead of committing it in main branch we create new branch so our main branch is working fine & simultaneously we can create new features after those features are working well then we can merge all those code in main branch.

```
commit 0912208fff5dca043843004a7a710054097cdfc9 (HEAD -> master, origin/master)
```

Here, this is recent commit which is also mirrored in remote. Denoted by Head -> master, **origin/master**

Branch:

Branches in Git are incredibly lightweight as well. They are simply pointers to a specific commit -- nothing more. This is why many Git enthusiasts chant the mantra:

branch early, and branch often

Because there is no storage / memory overhead with making many branches, it's easier to logically divide up your work than have big beefy branches.

Use -> **git branch "new branch name"**

```
$ git branch alien-plot
```

Use -> **git branch** to see all branches.

```
$ git branch
alien-plot
* master
```

In this example there are two branches, one is master and another is alien-plot.

'*' shows which branch you are currently on.

To switch to another branch

use -> **git checkout "branchName"**

```
$ git checkout alien-plot
Switched to branch 'alien-plot'
```

here's a shortcut: if you want to create a new branch AND check it out at the same time, you can simply

use -> **git checkout -b [yourbranchname]**.

After editing your files for experiment(or you're not sure but researching on changes) and then if you added files to staging area by add command and then commit. Then this commit is for branch alien-plot not for master.

```
$ git log
commit f7e274b4f33abb1ed1dcba99e3f98d7bda818ca2 (HEAD -> alien-plot)
Author: NINAD [redacted]
Date: Sun Dec 18 13:48:05 2022 +0530

    Change chapter1 with negetive words

commit 0912208fff5dca043843004a7a710054097cdfc9 (origin/master, master)
Author: NINAD [redacted]
Date: Sat Dec 17 14:09:10 2022 +0530

    Complete Chapter2 and Chapter3

commit 53a18d9f63ec504cde93cca3027d3acc20f4445f
Author: NINAD [redacted]
Date: Sat Dec 17 13:57:33 2022 +0530

    Complete Chapter-1
```

Use -> `git checkout master` to go to the master branch.

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

On github (origin/master is latest commit but for local,

```
$ git log
commit eb863f8dc33385b5f23c1de9f244555fa3a7afa2 (HEAD -> master)
Author: NINAD [redacted]
Date: Sun Dec 18 14:00:09 2022 +0530

    add chapter4.txt

commit 0912208fff5dca043843004a7a710054097cdfc9 (origin/master)
Author: NINAD [redacted]
Date: Sat Dec 17 14:09:10 2022 +0530

    Complete Chapter2 and Chapter3

commit 53a18d9f63ec504cde93cca3027d3acc20f4445f
Author: NINAD [redacted]
Date: Sat Dec 17 13:57:33 2022 +0530

    Complete Chapter-1
```

it is (head
-> master)

If you changed branch then your local files are also change according to the branch in file manager.

Merging:

This will allow us to branch off, develop a new feature, and then combine it back in.

So if you satisfied with those experimental changes in alien branch and want to merge with original master branch then,

1st go back to the master branch

Now use -> `git merge "branch name"`

```
$ git merge alien-plot
Merge made by the 'ort' strategy.
 chapter1.txt | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

```
$ git log
commit 23e7ab4af6765ff50d01e76270bd2900a6bf3fbd (HEAD -> master)
Merge: eb863f8 f7e274b
Author: NINAD <[redacted]>
Date: Sun Dec 18 14:06:36 2022 +0530

Merge branch 'alien-plot'
```

So this is latest commit in master branch.

Now push changes to remote folder using `git push origin master -u`



this is graph of our commits and branches.

At point 2 we made alien-plot branch and then at final point we merge it to master.

README: is a text file that tells other people what your github repository is all about.

Forking and Pull request:

If anybody wants to fix bugs or add new feature in your project/ you wanted to do this in others project you've to make a copy of it (not cloning)

Git clone is just grabbing at the entirety of the repository, and then cloning it to your local work environment.

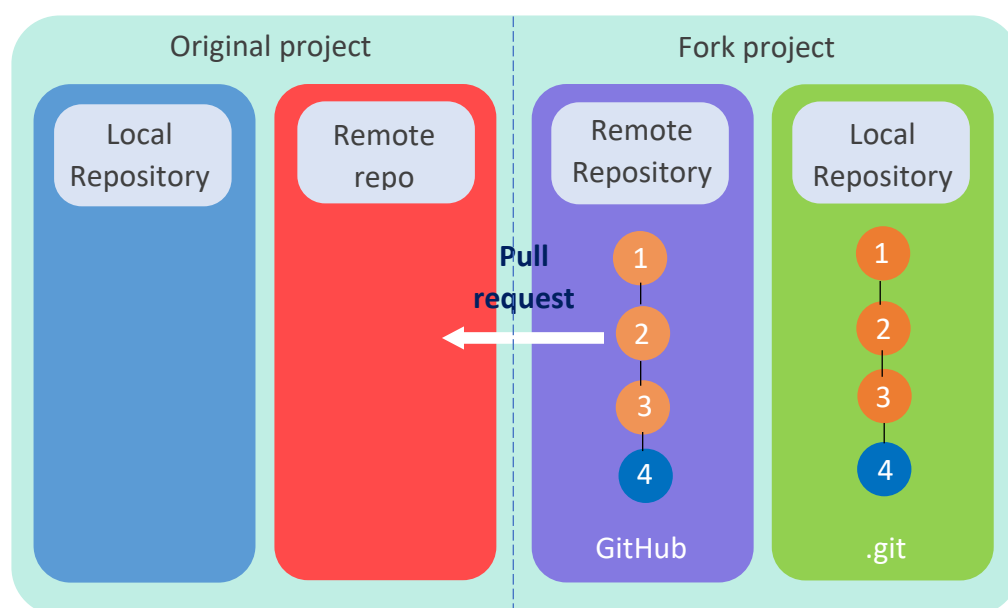
Now in this case we're basically just copying a repository that's hosted on GitHub and we're keeping the copy under our own GitHub account where we can make changes to it.

It is called forking.

Cloning permission isn't accessible to all in big projects, write is accessible to team members only. So if you want copy then 1st fork it and then clone it.

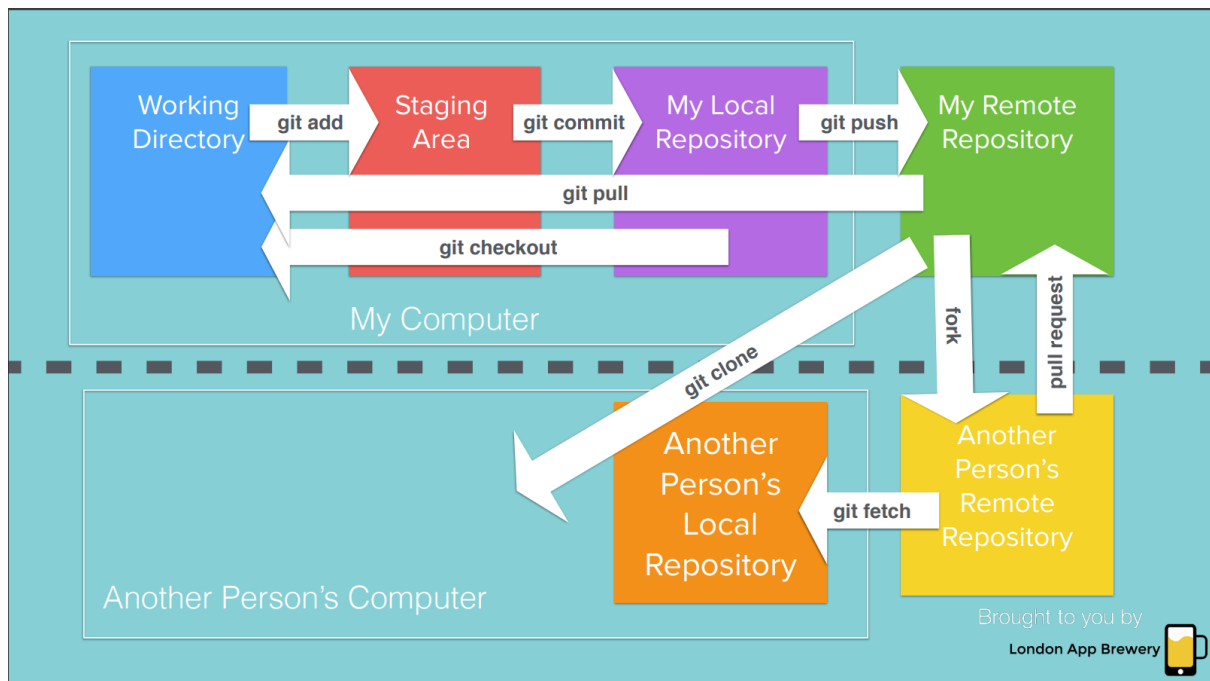
After forking you own that project and you've full permissions to do whatever you want.

After fixing bug or adding feature you've to push it to remote repo from local and if you want to add your feature in there original code you've to make a pull request.



Original project's team member (who's access to write) reviews your pull request and decides to merge or not your code in original repo.

Git CheetSheet:



EJS - Embedded JavaScript Templating

new Date():

it is a function in JS without arguments, creates a date object with the current date and time.

Tue Dec 20 2022 13:35:56 GMT+0530 (India Standard Time) It gives data like this.

Ex.

```
var today = new Date();
if(today.
    number getDate()
    number getDay()
    number getFullYear()
    number getHours()
    number getMilliseconds()
    number getMinutes()
    number getMonth()
    number getSeconds()
    number getTime()
```

You can get many options by this.

It is used to get current date, time, year ...

Res.write to send multiple pieces of data.

```
var today = new Date();

if(today.getDay() === 2){
    res.write("<h1>Yay! it's the weekend! :)");
    res.send();
}
else {
    res.write("<p>It's not a weekend, ");
    res.write("<h1>Boo! I have to work. :(");
    res.send();
}
});
```

0 - 6 === Sunday - Saturday

Templating:

It is HTML template where we can change certain parts of the HTML depending on the logic in our server.

Install ejs using: `npm install ejs`.

To use ejs in our app, use ->

```
app.use("view engine", "ejs");
```

uses the view engine that we set up here to render a particular page.

To use ejs it requires a separate directory called 'views' which contains a file called 'fileName.ejs'. So view engine goes to the folder and looks for files that we're trying to render.

```
views\lists.ejs
```

inside this list.ejs file you're essentially writing HTML code.

Marker `<%= EJS %>` tells the file that this is where you should place a particular variable. (in .ejs file)

```
<h1>It's a <%= kindOfDay %>! </h1>
```

```
res.render('lists', {  
  kindOfDay: dayName  
});
```

In app.js

It passes the name of day to markup in .ejs file i.e kindOfDay.

Note: If there are more than 5 statement then use switch statement, else use if else statement.

Scriptlet tag: `<% script %>`

scriptlet tag allows us to run some basic Javascript code namely for control flow purposes.

Ex. if-else, switch

```
<% if(kindOfDay === "saturday") { %>
  <h1 style="color: purple"><%= kindOfDay %></h1>
<% } %>
<% else { %>
  <h1 style="color: blue"><%= kindOfDay %></h1>
<% } %>
```

But, always try to do all of your logic inside your server and only in select situations where it really is about modifying the content based on the variable do you actually add in the scriptlet tags.

[How to change date format?](#)

```
var options = {
  weekday: "long",
  day: "numeric",
  month: "long"
};
```

```
var day = today.toLocaleDateString("en-US", options);
```

A ToDo list:

```
let items = ["Buy food", "Cook food", "Eat food"];

app.get("/", function(req, res) {
  let date = new Date();
  let options = {
    weekday: "long",
    month: "long",
    day: "numeric"
  }
  let day = date.toLocaleDateString("en-US", options);
  res.render("lists", {todayDay: day, newTask: items});
});

app.post("/", function(req, res) {
  item = req.body.newItem;
  items.push(item);
  res.redirect("/");
})
```

When user send response from form then it stored in items array.

Then we redirect it to home route and via render user's response is displayed in form of list.

`let items = ["Buy food", "Cook food", "Eat food"];` This is globally declared array. Whenever we add a task in to do list then size of items array is increased by 1 and task is pushed in array.

```
<ul> |
<%   for(let i = 0; i < newTask.length; i++) { %>
    <li><%= newTask[i] %></li>
<%   }      %>
</ul>
```

We're using for loop to display each array element in different

Scoop:

Ex.

```
function a {  
  var x = 2;  
  console.log(x);  
}  
console.log(x);
```

Here scope of x is local to func a only and not outside of function.

```
var x = 2;  
function a {  
  console.log(x);  
}  
console.log(x);
```

Here scope of x is inside of function and outside of function also.

It is called global variables.

```
if(true) {  
  var x = 2;  
  console.log(x);  
}  
console.log(x);
```

This is possible. If we created var in if, while, for loop or anything other than function you can access var outside also.

```
var x = 2;  
let y = 3;  
const z = 4;
```

const: value can't be changed once you give the value.

Let: can be able to change value.

Var: can be able to change value.

If they declared in function then they are local variables and if they created outside of function then they're global variables.

If we use var in if, for loop, while loop then we can access it outside of function also (It is automatically a global variable) but it doesn't work for let and const.

Therefore try to avoid using var. Instead use **let** or **const** always.

Whenever you want to write var replace it with let.

when you use Express, it doesn't automatically serve up all the files in your project. In fact it only serves up the main access point which we define in our package.json as app.js and also serves views folder.

So we've to tell explicitly to express to serve those files(css, js, sound, images...) so we've to create new folder "public" and move those files in it and use,

```
app.use(express.static("public"));
```


EJS layouts:

```
<form class="item" action="/" method="post">
  <input type="text" name="newItem" placeholder="New Item" autocomplete="off">
  <button type="submit" name="list" value=<%= listTitle %>>></button>
</form>
```

In to do list we want two list first is personal and second is work list so we modified our code as we gave name to button 'lists' and value is dynamic if list title Is work then only it go and stores data in list item;

```
{ newItem: 'dkjs', button: '' }
```

```
{ newItem: 'djf', list: 'Work' }
```

```
<button type="submit" name="list" value=<%= listTitle %>>></button>
```

after this if you console.log

```
console.log(req.body.list);
```

then you get value of list button i.e, work

```
Server is started on port 3000 .|..|.
Work
```

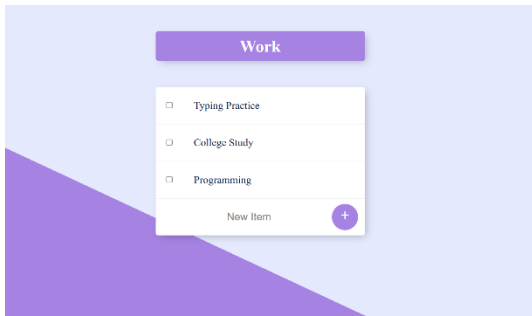
```
app.post("/", function(req, res) {
  console.log(req.body.list);
  item = req.body.newItem;
  if(req.body.list === "Work") {
    workItems.push(item);
    res.redirect("/work");
  } else {
    items.push(item);
    res.redirect("/");
  }
});

app.get("/work", function(req, res) {
  res.render("lists", {listTitle: "Work List", newTask: workItems});
});
```

So now you can use this logic.

EJS layouts/ Partial:

Many times we want same style in all our webpages but not want same content like this if you want only style of background, text but you don't want that table and all stuff. So there is option of copy and pasting but this isn't good if your site contains many webpages ,



therefore there is a option is
ejs called layouts/partial

What is exports in modules?

How Node modules like `ejs`, `body-parser`, `express` are work by creating a module of our own from scratch.

Code that's not strictly related to the route shouldn't actually be in here cluttering the app.

Here we created `date.js` file in which contains a function that returns day. To export it / to get out it from `date.js` file to available elsewhere if it'

```
function getDate() {  
  let date = new Date();  
  let options = {  
    weekday: "long",  
    month: "long",  
    day: "numeric"  
  }  
  let day = date.toLocaleDateString("en-US", options);  
  return day;  
}
```

Module object is basically something that gives you a reference to the object that represents the current module.

Inside that date.js file it has access to module.

To use this module we've to require it in app.js.

But this module is local so you've to require it as ,

```
const date = require(__dirname + "/date.js");
```

After call,

```
Module {
  id: 'N:\\NINAD --{ NAM DO-SAN }--\\DEVELOPEMENT\\todoList-v1.0\\date.js',
  path: 'N:\\NINAD --{ NAM DO-SAN }--\\DEVELOPEMENT\\todoList-v1.0',
  exports: {},
  filename: 'N:\\NINAD --{ NAM DO-SAN }--\\DEVELOPEMENT\\todoList-v1.0\\date.js',
  loaded: false,
  children: [],
  paths: [
    'N:\\NINAD --{ NAM DO-SAN }--\\DEVELOPEMENT\\todoList-v1.0\\node_modules',
    'N:\\NINAD --{ NAM DO-SAN }--\\DEVELOPEMENT\\node_modules',
    'N:\\NINAD --{ NAM DO-SAN }--\\node_modules',
    'N:\\node_modules'
  ]
}
```

Parent module is a module who requires it or a module who calls that file.

```
module.exports = getDate;
```

here we exported getDate function.

```
console.log(date()); Tuesday, December 27
```

```
let day = date();
res.render("lists", {listTitle: day, newTask: items});
```

we have this module called date and we're requiring it up here which binds all of the exports to this constant called date. And then down here we call the function that's bound to that constant date and we activate our getDate function. And now we have the result of it stored inside day

Whenever we use const to create array it is possible to push elements in that array, But it's not possible to simply assign it to a brand new array.

```
exports.getDate = function() {
  let date = new Date();
  let options = {
    weekday: "long",
    month: "long",
    day: "numeric"
  }
  return date.toLocaleDateString("en-US", options);
};
```

```
let day = date.getDate();
res.render("lists", {listTitle: day, newTask: items});
```

Express Routing Parameters: ->

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.params object

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

```
Route path: /flights/:from-:to
Request URL: http://localhost:3000/flights/LAX-SFO
req.params: { "from": "LAX", "to": "SFO" }
```

```
Route path: /plantae/:genus.:species
Request URL: http://localhost:3000/plantae/Prunus.persica
req.params: { "genus": "Prunus", "species": "persica" }
```

Use -> :parameter

Use colon and then parameter name

```
app.get('/blogs/:blogPage', (req, res) => {
  console.log(req.params.blogPage);
});
```

Because of this feature we don't need to create many pages for project,

Ex.

Home Explained Political Pulse Cities Opinion Entertainment Lifestyle Technology Sports Premium

Challenge: Blog post

Whenever you copy/download project from github, folder 'Node modules' isn't found in downloaded project, but if we see dependencies in json file you will see those modules which that project owner installed. To install all this modules then simply use 'node install' it will install all dependency modules in your project.

In views folder most developer creates 'partials' folder to keep partials (header, footer) i.e partially completed html files.

```
<%- include('partials/header'); -%>
```

Routing parameters:

```
app.get('/blogs/:blogPage', (req, res) => {  
  const pageName = req.params.blogPage;  
  
  blogsArr.forEach(blog => {  
    if (pageName === blog.blogTitle) {  
      res.render('post', {blog});  
    } else {  
      console.log('No Match found!');  
    }  
  });  
});
```

Whenever we want to see blog post page we aren't creating page for each post instead we use routing parameters so only one page can change contain dynamically.

Instead of for loop you can use for each loop

```
<% for(let i = 0; i < postArr.length; i++) { %>  
<% console.log(postArr[i].postTitle); %>  
<% } %>
```

```
<% postArr.forEach(function(post) { %>  
<% console.log(post.postTitle); %>  
<% }); %>
```

Some blog post websites use urls with hypens this word formatting called kebab case in programming.

Lodash: is a utility library that makes it easier to work with Javascript inside your Node apps.

It is very useful to convert url in lowercase.

```
const blogName = _.lowerCase(blog.blogTitle);
```

Truncate Text:

```
<p><%= blogPost.substring(0, 100) + '...' %>
```

Databases

There are a lot databases. Depending on the type of data you're looking to store and the structure of your data, you might favor one of these databases over another.

But there are two types of data bases:

- 1) SQL based
- 2) NoSQL based

SQL - Structured Query Language

It is old technology. But businesses are using it for many many years.

NoSQL - Not only structured Query Language

It is new compare to SQL.

Top popular databases that work with,
SQL are MySQL and Postgres and for NoSQL are mongoDB and redis

Structure:

SQL database will group your data into tables.

They are inflexible. It's relational.

If in your sql based database if there are specific columns and if you want to take input of another data from user then, it isn't possible as it is fixed table with column, so you've to put another column with that property

and then put that user's data but for all other users it's value is initialize with NULL.

Ex. if the data is of email address then unnecessarily emails are send to NULL.

If user didn't put some data's information then your database is irregular.

But In NoSQL data base data represented as json object so if you want to put extra info of somebody then it's possible without disturbing other.

So it's flexible. It is non-relational.

But NoSQL has good scalability than sql.

If you direct use table then same users data repeates if they put info again. So if you use many tables then it is possible to point towards that one user and link to it.

Ex. Instagram user produces many data - post , comment, images..... so in this case noSQL based (mongoDb) is good to use.

If SQL data has more information and file is so big then it requires more powerful computer. Otherwise it can't handle it.

It is like you're building a building vertically after at some point it goes unstable and might collapse and also costly.

But noSQL is horizontal building. It allows distributed system So your database can be distributed amongst lots and lots of different computers.

Schema: before creating SQL database you should have structure in mind before you create database.

How many column/row needs...

So, if you're a young startup and you haven't really figured out how you're going to organize your data or what kind of data you're going to store, then it might be a better idea to choose this particular database.

My SQL	MongoDB
More Mature	Shiny and new
Table structure	Document structure
Requires schema	More flexible to changes
Great with relationships	Not great with complex relationships
Scales vertically	Scales horizontally

Main things of every database are:

- 1) Create
- 2) Read
- 3) Update
- 4) Destroy

SQL:

Use sqlonline.com to practice sql.

▪ Create table:

```
1 CREATE TABLE products (  
2   id INT NOT NULL,  
3   name STRING,  
4   price MONEY,  
5   PRIMARY KEY (id)  
6 )
```

Here id is a primary key. So we'll be able to identify each row by its id. So there isn't another product with same id.

id INT NOT NULL: This guarantees that whenever new records are being created inside this table if the record doesn't provide an ID then it will not allow it to be created so it cannot be null.

Primary Key: It allows a particular column to uniquely identify each record in a database.

▪ Add data into table:

```
1 INSERT INTO products  
2 VALUES (1, "Pen-1", 10.0)
```

id	name	price
1	Pen-1	10

Here we added values of product 1 in table.

If you don't have data for all columns then you've to specify it first that which column's data you have.

```
1 INSERT INTO products (id, name)
2 VALUES (2, "Pencil")
```

Here we don't have data for price so we write what values you're going to put

id	name	price
1	Pen-1	10
2	Pencil	NULL

If you're creating a table which don't have any Id(i.e primary key) then it gives you an error because you set it as 'NOT NULL'

```
1 INSERT INTO products (name, price)
2 VALUES ("rubber", 5)
```

SQLite.js

NOT NULL constraint failed: products.id

Read Table:

To display everything from table,

```
SELECT * FROM products
```

Here * is a wildcard (means select everything from products table)

If you only wanted to see one column or two columns then,

```
SELECT name, price FROM products
```

name	price
Pen-1	10
Pencil	NULL

Here only two columns are shown but id column isn't displayed.

If you wanted to see particular row,

SQL where:

If you want to search id=1 then,

```
SELECT * FROM products WHERE id=1
```

It gives output of id=1.

- **Update:** How do you update records in database?

```
1 UPDATE products
2 SET price = 5.0
3 WHERE id=2
```

Here Null value of id=2 in price column is updated with 0.5

2	Pencil	5
---	--------	---

If you wanted to change table instead of particular record,

Means, now if we want to add new column called 'stock' in existing table contains id, name, price then we've to use,

Alter Table:

```
1 ALTER TABLE products add col stock of datatype int
2 ADD stock INT
```

id	name	price	stock
1	Pen-1	10	NULL
2	Pencil	5	NULL

So we added new column called stocks and value of all products-stock is initialized with NULL.

- **Delete:**

```
1 DELETE FROM products
2 WHERE id=2
```

It deletes id=2 row

id	name	price	stock
1	Pen-1	10	32

Relationship in SQL:

Foreign Key:

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Creating another table called 'orders'

```

1 CREATE TABLE orders (
2   id INT NOT NULL,
3   order_number INT,
4   customer_id INT,
5   product_id INT,
6   PRIMARY KEY (id),
7   FOREIGN KEY (customer_id) REFERENCES customers(id),
8   FOREIGN KEY (product_id) REFERENCES products(id)
9 )

```

Here id is a primary key and customer_id and product_id are foreign keys. They are linked to customers and products tables respectively.

We also created table customers,

id	first_name	last_name	address
1	Sejoon	Park	Seollite tower
2	Namjoon	Kim	Han tower

So now we're going to add data in orders table,

1st : Namjoon (id=2) bought product 1 (i.e pen)

```
1 INSERT INTO orders
2 VALUES (1, 4321, 2, 1)
```

id	order_number	customer_id	product_id
1	4321	2	1

Join: To join tables together.

Inner join:

```
1 SELECT orders.order_number, customers.first_name, customers.last_name, customers.address
2 FROM orders
3 INNER JOIN customers
4 ON orders.customer_id = customers.id;
```

From the foreign keys inside our orders table. Means in orders table you'll find particular key match.

After the keyword 'on' is going to be the fields that will match. Foreign key from orders (orders.customer_id) = primary key of customers table(customers.id)

Select: after select you've to select which columns you want in new table.

This is new table.

order_number	first_name	last_name	address
4321	Namjoon	Kim	Han tower

MongoDB:

1st create folder 'data' in C drive inside drive make 'db' folder where our databases can store locally.

[Click here to read complete setup -->](#)

1st create file .bash_profile using hyperterminal at root path .

```
ninad@LAPTOP-65K3VRVH MINGW64 ~  
$ touch .bash_profile
```

It is going to tell our hyperterminal that when it starts up and we want to access mongo and mongod we have a shortcut for it

Now, edit that file using vim

```
ninad@LAPTOP-65K3VRVH MINGW64 ~  
$ vim .bash_profile
```

To type anything in vim editor 1st initialize it by pressing i button. (insert mode)

And type:

```
alias mongod="/c/Program\ files/MongoDB/Server/6.0/bin/mongod.exe"  
alias mongo="/c/Program\ Files/MongoDB/Server/6.0/bin/mongo.exe"
```

Use :wq! To save and exit.

But the above step of vim is outdated so I used,

- 1) Download: MongoDB shell which is new mongosh.exe which is replacement of mongo.exe
- 2) Extract and then cut all files and paste it inside the bin folder where mongod.exe is located (cut bin folder files and paste in bin folder which contain mongod.exe)

3) Update vim as,

```
alias mongod="/c/Program\ files/MongoDB/Server/6.0/bin/mongod.exe"  
alias mongos="/c/Program\ Files/MongoDB/Server/6.0/bin/mongos.exe"  
alias mongosh="/c/Program\ Files/MongoDB/Server/6.0/bin/mongosh.exe"
```

Type mongod, this will spin up your mongo server

```
ninad@LAPTOP-65K3VRVH MINGW64 ~  
$ mongod
```

```
{ "t": { "$date": "2022-12-29T19:33:24.580+05:30" }, "s": "I", "c": "NETWORK", "id": 23016, "ctx": "listener", "msg": "Waiting for connection  
s", "attr": { "port": 27017, "ssl": "off" } }
```

This is last line “waiting for connections” this is similar like localhost: 3000 but in this case our local host is ‘27017’

On another window use,

```
ninad@LAPTOP-65K3VRVH MINGW64 ~  
$ mongosh
```

to initialize mongodb shell

```
test> 
```

after that you’re ready to use mongoShell

Mongo Shell: Mongo shell is basically just a way for us to be able to interact with our MongoDB databases on our local system using the command line.

Use -> help for get help to find commands

'show dbs': Print a list of all available databases.

```
test> show dbs  
admin    40.00 KiB  
config   60.00 KiB  
local    40.00 KiB
```

MongoDB is preloaded with three databases admin, config, local.

Use -> **use <DB name>**

```
test> use shopDB
switched to db shopDB
```

Here we created DB called shopDB. To listed it with above 3 databases it has to have some content.

Use -> **db** to know which db you're currently in

```
shopDB> db
shopDB
```

- Create:

`db.collection.insertOne()` / here collection is a name of collection. (similar to SQL table)
`db.collection.insertMany()`

If that collection name isn't currently exists then by executing it will create that collection.

Inside function you would pass over JS object which is called as document.

```
shopDB> db.products.insertOne({_id: 1, name: "pen", price: 1.20})
{ acknowledged: true, insertedId: 1 }
```

Use -> **show collections** To show collection in database

```
shopDB> show collections
products
```

- Read:

`db.collection.find()` To get all data

```
shopDB> db.products.find()
[
  { _id: 1, name: 'pen', price: 1.2 },
  { _id: 2, name: 'pencil', price: 0.8 }
]
```

You can also use (query, projection) in this function such as,

Ex. if you find data which contains name == pencil

```
shopDB> db.products.find({name: "pencil"})
[ { _id: 2, name: 'pencil', price: 0.8 } ]
```

If you don't want some data then use projection,

```
shopDB> db.products.find({_id: 1}, {name: 1, _id: 0})
[ { name: 'pen' } ]
```

Here we set name as true, and id as false

- Update:

```
db.collection.updateOne()
```

```
db.collection.updateMany()
```

```
db.collection.replaceOne()
```

Now we want to add info about 'stock' in one of the records then,

Use -> db.products.updateOne(...)

```
shopDB> db.products.updateOne({_id: 1}, {$set: {stock: 32}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Updated record:

```
shopDB> db.products.find()
[
  { _id: 1, name: 'pen', price: 1.2, stock: 32 },
  { _id: 2, name: 'pencil', price: 0.8 }
]
```

▪ Delete:

```
db.collection.deleteOne() ^
```

```
db.collection.deleteMany()
```

```
shopDB> db.products.deleteOne({_id: 1})
{ acknowledged: true, deletedCount: 1 }
shopDB> db.products.find()
[ { _id: 2, name: 'pencil', price: 0.8, stock: 12 } ]
```

Relationships in MongoDB:

```
shopDB> db.products.insertOne(
...   {
...     _id: 3,
...     name: "rubber",
...     price: 1.30,
...     stock: 43,
...     reviews: [
...       {
...         authorName: "Kim Ninadeu",
...         rating: 5,
...         review: "Best rubber ever"
...       },
...       {
...         authorName: "Park Hyungshik",
...         rating: 1,
...         review: "Ah! not that best :("
...       }
...     ]
...   }
... )
```

As users reviews it will
append to reviews array

So we established
relationship between single
product document and single
review document.

To delete database, go to that database using use <DBname>

And then use -> db.dropDatabase()

```
fruitsDB> db.dropDatabase()
{ ok: 1, dropped: 'fruitsDB' }
```

```
// Products collection
{
  id: 1,
  name: 'pen',
  price: 20
}
{
  id: 2,
  name: 'pencil',
  price: 10
}

// Orders Collection
{
  orderNo: 3312,
  orderedProducts: [1, 2]
}
```

Integrate MongoDB with NodeJS:

To connect node app with mongoDB there are two ways:

- 1) MongoDB native driver
- 2) Use Mongoose

MongoDB native Driver:

Driver is used to enable mongoDB to interact with our app.

1st install npm package of mongoDB

```
$ npm i mongodb
```

Everything related to assert is for testing.

Starting code

```
const { MongoClient } = require('mongodb');
// or as an es module:
// import { MongoClient } from 'mongodb'

// Connection URL
const url = 'mongodb://0.0.0.0:27017';
const client = new MongoClient(url);

// Database Name
const dbName = 'fruitsDB';

async function main() {
  // Use connect method to connect to the server
  await client.connect();
  console.log('Connected successfully to server');
  const db = client.db(dbName);
  const collection = db.collection('documents');

  // the following code examples can be pasted here...

  return 'done.';
}

main()
  .then(console.log)
  .catch(console.error)
  .finally(() => client.close());
```

Instead of localhost
use 0.0.0.0

here client is
trying to connect to
mongoDB database &
if fruitsDB doesn't
exists then it will
create new one.

If all happen
without errors then
it console.log as
successful.

Then we're going to
close connection.

Adding data to database: {_,_} it is individual document or
record

Mongoose

Mongoose is a ODM (Object Document Mapper)

It allows nodejs (who speaks language of JS object) to able to talk with mongoDB database (which speaks language of documents, collections & databases)

Install Mongoose:

```
$ npm i mongoose
```

Connect to mongoDB:

```
mongoose.set('strictQuery', true);  
mongoose.connect('mongodb://127.0.0.1:27017/fruitsDB', { useNewUrlParser: true });
```

Create Schema:

```
const fruitsSchema = new mongoose.Schema ({  
  name: String,  
  rating: Number,  
  review: String  
});
```

Using Schema create mongoose model (table):

```
const Fruit = mongoose.model('Fruit', fruitsSchema)
```

'Fruit' is a collection name. It will converted as fruits in plural form automatically by mongoDB (it uses lodash)

Creating document(row) from model:

```
const fruit = new Fruit ({
  name: 'Apple',
  rating: 7,
  review: 'Pretty solid as a fruit!'
})
```

Fruit.save() to save in DB

insertMany:

```
Fruit.insertMany([mangu, orange, banana], function(err) {
  if(err) {
    console.log(err);
  }
  else {
    console.log('Successfully saved all fruits in fruits DB');
  }
});
```

Find() in NodeJS:

```
Fruit.find(function(err, fruitsData) {
  if(err) {
    console.log(err);
  } else {
    console.log('All Fruits in DB: \n', fruitsData);
  }
});
```

It gives data in array.

Disconnect mongodb:

```
mongoose.connection.close(function() {
  console.log('Mongoose default connection disconnected through app termination');
  process.exit(0);
});
```

Use this in last function.

Data Validation with Mongoose:

```
rating: {  
  type: Number,  
  min: 1,  
  max: 10  
},
```

[Validation Docs on mongoDB](#)

```
name: {  
  type: String,  
  required: [true, 'Please check your data entry, No name specified!'];  
},
```

Here name is compulsory required if user not give name then it will show error msg i.e 'please check..... ified'

Update:

```
Fruit.updateOne({_id: '63ce9447002b55dfc4e5f6a7'}, {name: 'Peach'}, function(err) {  
  if(err) {  
    console.log(err);  
  } else {  
    console.log('Successfully Updated the document :');  
  }  
})
```

Delete:

deleteOne

```
Fruit.deleteOne({_id: '63ce9447002b55dfc4e5f6a7'}, function(err) {  
  if(err) {  
    console.log(err);  
  } else {  
    console.log('Successfully deleted one document :');  
  }  
})
```

deleteMany:


```
Person.deleteMany({name: 'Ninad'}, function(err) {
  if(err) {
    console.log(err);
  } else {
    console.log('Deleted all Successfully');
  }
})
```

Relationships between your documents:

```
const personSchema = new mongoose.Schema({
  name: String,
  age: Number,
  favoriteFruit: fruitsSchema
});
```

Favoritefruit: fruitSchema tells mongoose that we're embedding a fruit document in a person document.

Creating new fruit and person(amy) -

```
const pineapple = new Fruit({
  name: 'PineApple',
  rating: 8,
  review: 'Eat in summer only pineapple'
});
```

```
const person2 = new Person({
  name: 'Amy',
  age: 12,
  favoriteFruit: pineapple
});
```

Now amy's favorite food is linked to fruits model/collection (table)

How to submit a form when there is no submit button?

```
<input type="checkbox" name="checkbox" value="<%=item._id %>" onchange="this.form.submit()">
```

Here onchange allows you to post when you click the checkbox.

Via value attribute we assign id of list element to each checkboxes so when user checks checkbox app knows that which checkbox (/which task have to delete)

Express allows us to use route parameters to create dynamic routes.

Ex.

```
app.get("/:customListName", function(req, res) {  
  console.log(req.params.customListName);  
});
```

If we search localhost://3000/home then it logs home.

```
<input type="hidden" name="listName" value="<%= listTitle %>"></input>
```

It type hidden means it's not visible to user and can't modified by user.

Build Your Own RESTful API from Scratch

What is REST? & What does it mean to make an API RESTful?

REST = Representational State Transfer

When client make request to the sever classically on internet it's done through a HTTP request (Hypertext Transfer Protocol request)

HTTPS request = here 's' stands for secure request

All your requests and responses are going across the internet and can be potentially tapped in by lots of people.

REST is an architectural style for designing APIs.

Other architectural styles: SOAP, GraphQL, FALCOR, ...

You've bunch of rules to making an API RESTful, but two most important ones are,

1. Using HTTP request verb
2. Using specific pattern of Routes/Endpoint URLs

HTTP verbs:

1. GET
2. POST
3. PUT
4. PATCH
5. DELETE

GET:

It's same as READ.

Whenever you want our server to serve up some resource, we've been using `app.get((req, res))`.

We passing a callback that response the request and sends result back & if the request involved something that relates to our database then that's the equivalent of searching through our DB and returning data as result.

POST:

It's corresponds to 'CREATE' word in CRUD functions.

When data is posted to our server we create new entry to DB and we save that data for later.

In this case request contains data and response is simply success/ gives error code.

PUT and PATCH:

It's similar to UPDATE

`app.put / app.patch ((req, res))`

PUT is equivalent to Updating our DB by sending entire entry to replace previous one.

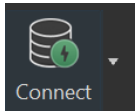
PATCH is equivalent to Updating our DB by only piece of data that needs to be update.

DELETE:

Deletes particular piece of data in our DB.

Robo 3T: Is a graphical user interface that is commonly used with MongoDB.

1st create new connection. And then start your mongo server using mongod in terminal and then connect new connection.



Now create new database called 'wikiDB' and add collection to it i.e 'articles'.

Read naming conventions of MongoDB -->

Click on add documents button to add new document.



To add many docs at once, right click on area of collection and click on paste document(s).



Paste Document(s)...

Ctrl+V

Using Express App:

1st setup your server.

Get request:

```
app.get ("/articles", (req, res) => {  
  Article.find ({}, (err, foundArticles) => {  
    if (!err)  
      res.send (foundArticles);  
    else  
      res.send (err);  
  });  
});
```

It sends all documents which are present in Article collection.

Post request:


```
app.post("/articles", (req, res) => {
  const newArticle = new Article ({
    title: req.body.title,
    content: req.body.content
  });

  newArticle.save ((err) => {
    if (!err)
      res.send ("Successfully added new article!");
    else
      res.send (err);
  });
});
```

It saves data which is send by user and sends successful / err message.

To make all this requests (post) we need frontend and form. But if you don't want this and only want to test your API, then use Postman.

POST localhost:3000/articles

And use  x-www-form-urlencoded from body.

Delete method:

```
app.delete("/articles", (req, res) => {
  Article.deleteMany ({}, (err) => {
    if (!err)
      res.send ("Successfully deleted all articles!");
    else
      res.send (err);
  });
});
```

It deletes all docs from articles collection.

[App.route\(\)](#): chaining method

Whenever your app contains one route with different methods then use this.

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .put((req, res) => {
    res.send('Update the book')
  })
```

Space in url is represent as %20.

[Read all url encoding -->](#)

You can also use express routing parameters to access a specific article.

Ex. `app.route ("/articles/:title")`

App.put():

You can use replaceOne() to update entire document.

```
Article.replaceOne (
  {title: articleTitle},
  {title: req.body.title, content: req.body.content},
  (err) => {
    if (!err) {
      console.log ("Successfully updated article");
      res.send ("Successfully updated article");
    } else {
      res.send (err);
    }
  }
)
```

App.patch():

```
.patch(function(req, res){
  Article.update(
    {title: req.params.articleTitle},
    {$set: req.body},
    function(err){
      if(!err){
        res.send("Successfully updated article.");
      } else {
        res.send(err);
      }
    }
  )
})
```

To update some part of document.

Authentication & Security

Why Authenticate?

When we create web app for users they start generating data (ex. Like certain post, interact with other user, messages) so in order to associate those data with individual users, we need account for each user.

Using username & password we essentially create kind of like ID card for each user to uniquely identify them on our database and save all the data.

So when next time they come back to our website they'll be able to use username and password and log in to website & be able to access all of their private data.

Restrict Access:

Sometimes we need authentication to restrict some users from accessing certain area of our website depending on status of user.

Ex. Spotify, Netflix subscription

Login and signup is simple but difficult part is how secure you're going to make website.

There are 6 levels of security.

Making an app which shows secrets of anonymous user:

Level 1: Register user with Username & password


```

app.post ("/register", (req, res) => {
  const newUser = new User ({
    email: req.body.email,
    password: req.body.password
  });

  newUser.save ((err) => {
    if (err)
      console.log (err);
    else
      res.render ("secrets");
  });
});

```

We not created
 app.get("/secrets") route
 because we don't want any user
 to visit that page without log
 in so we render this page only
 if user register / log in our
 site.

In 1st level of authentication we're storing user password
 as a plain text so it's very dangerous because any
 employee of our company can see it and can get access of
 any user's account.

Login:

```

app.post ("/login", (req, res) => {
  const userName = req.body.username;
  const password = req.body.password;

  User.findOne ({email: userName}, (err, foundUser) => {
    if (err)
      console.log (err);
    else
      if (foundUser) {
        if (foundUser.password === password) {
          res.render ("secrets");
        }
      }
  });
});

```

Level 2: Database Encryption (AES algorithm)

What is Encryption?

It's just scrambling something so that people can't tell
 what the original was unless they were in on the secret
 and they knew how to unscramble it.

E.g [Enigma machine](#), Caesar cipher use cryptii.com

In modern world these methods are easy to crack there are new methods which are now used for encryption.

But all encryptions works same.

You have a way of scrambling your message and it requires a key to be able to unscramble that message.

Install [mongoose-encryption](#)

```
$ npm i mongoose-encryption
```

Then require it,

```
const encrypt = require ("mongoose-encryption");
```

Use method: Secret string instead of two keys

```
const secret = "Thisisourlittlesecret.";
userSchema.plugin (encrypt, {secret: secret});
```

Plugins: These are extra code which can add to the mongoose schemas to extend their functionality.

Here we're using secret to encrypt our DB. We do that by adding mongoose encrypt as a plugin to our userSchema and passing secret as a JS object.

Note add this plugin before creating mongoose model.

Here our plugin encrypt our entire DB. We don't want to encrypt email field and we only want to encrypt password field.

So use 'only encrypt certain fields':

Add 'encryptedFields: ['']' to your plugin.

```
userSchema.plugin (encrypt, {secret: secret, encryptedFields: ["password"]});
```

It is work as

When you save it will encrypt and decrypt when you find.

In our case mongoose will decrypt our password when we call findOne to check password that user enter is matching or not.

After encryption,

```
{
  "_id" : ObjectId("63eb8bf7ae7104bcce0cf10a"),
  "email" : "123@gmail.com",
  "_ct" : BinData(0, "YWFIFXxa91jE+0N0bFKW0PzOZeDiqpIXTZ063RbFT41vNB12iEamgUX7",
  "_ac" : BinData(0, "YYWWF8Rv408SEGjPx46UiGsc6jt02xmOrID0fjrviisaWyJfaWQiLCJ-",
  "__v" : NumberInt(0)
```

Our password is save as big string which is difficult to know password.

But this encryption isn't safe because if hacker hack our app.js which is easy to hack, then he can see our secret key and easily decrypt our data.

Using Environment variables to keep secrets safe

Environment Variables: It's a very simple file used to keep sensitive variables such as encryption key and API keys.

[Dotenv:](#)

Install dotenv,

```
$ npm i dotenv
```

At the top of your app.js file use,

```
require('dotenv').config()
```

Now create '.env' hidden file in root directory of your project.

```
$ touch .env
```

Now add environment variable to that file.

Format is NAME=VALUE without any spaces.

Use snake case for NAME and it's Capitalize.

Now add our SECRET const to .env file and remove it from app.js

```
SECRET=Thisisourlittlesecret.  
API_KEY=kdsjfiueu8479hfdjn
```

You can access this data as,

```
console.log (process.env.API_KEY);
```

Now change your plugin as,

```
userSchema.plugin (encrypt, [{secret: process.env.SECRET, encryptedFields: ["password"]}]);
```

While uploading your project to github you've to ignore files such as .env, nodemodules so create .gitignore file to ignore these files from uploading.

You can just copy gitignore file contain from github/gitignore for node.

Level 3: Hashing password

Biggest flaw in our current auth method is that we need an encryption key to encrypt and decrypt password. So if someone hack our environment variable then our password gets leak.

In hashing we no longer require encryption key.

We use hash function to turn password into hash and we store hash in DB.

It's almost impossible to turn a hash back into original text/password.

Ex. What are the factors of 377 other than 1 & 377. There are two numbers only.

Your job is to find those numbers, so you're going to divide 377 by 2 (i.e 188.5) then 3 then 4, 5, after lots of effort you can find those numbers i.e 13 & 29.

It takes long time.

But if we ask to calc $13 * 29$ You get 377 at very short time. So going backward is very time consuming. And going forward is simple/short time.

Hash functions are designed to be calculated very quickly going forwards but almost impossible to go backwards.

Calc hash takes milliseconds but to go backwards it takes (loooooong time)

When user log ins then password becomes hash & this hash compare against hash stored in DB.

Use -> md5 for hashing

Install md5

```
$ npm i md5
```

Require md5

```
const md5 = require ("md5");
```

Use md5 at registration password

```
password: md5(req.body.password)
```

Now after registration users password is stored as,

```
{
  "_id" : ObjectId("63eced993821a54cbc6d6ba1"),
  "email" : "nd@hash.com",
  "password" : "44f437ced647ec3f40fa0841041871cd",
  "__v" : NumberInt(0)
}
```

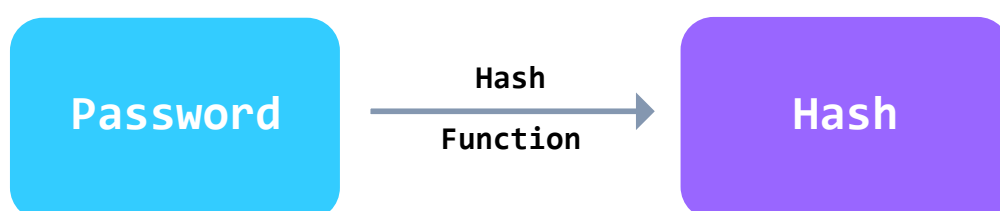
Which is very difficult to decrypt because there is no encryption key.

Note whenever you're going to hash same string, then hash created is always same for all same strings.

After console logging same string as password we get same hash.

```
console.log (md5("rrr"));
```

```
44f437ced647ec3f40fa0841041871cd
```



Hacking: ☠️

After creating account on specific website some website email you your email and password.

Ex. Go to this site to see examples --> [-->](#)

Here they're sending your username and password in email and says to you to log in and change password (current password is made by that website)

Here they're able to get password it means either they're using encryption key to store password (level-2 security) or they're using password in text format (level-1 security) which is worst possible type of security.

ihavebeenpwned.com use this website to check is your account is leaked in data breach / not.

Problem with level-3 security:

Same password == same hash

After hacking DB hackers see the common hashesh

Hackers make hash table which contains some commonly used passwords and their hashesh.

Then they compare users hashed password with hash table.

How hackers make Hash Tables:

- All words form a dictionary ($\approx 1,50,000$)

It creates hashes from all possible words.

- All numbers from telephone book. ($\approx 5,000,000$)
- All combination of charactors upto 6 places. ($\approx 19,770,609,664$)

With one of the latest GPUs we can calculate
20,000,000,000 MD5 hashes/Second

So it's easy to compare your password with 19,770... it
takes only (0.9 sec)

MD5 is one of the quickest hash to calculate.

As number of characters in password increases, computation
time to crack it increases exponentially.

[Password checker online:](#) use this site to check how much
time require to crack your account's password.

Hashing and Salting:

Salting: in addition to password we add random set of
charactors, so resulting hash created from both.

So salting increases password's complexity and increases
number of charactors.

If password of multiple accounts is same, by salting it
makes different hashes.

But if we use salting in md5 then it's also able to crack
it by table of combination of different salts because md5
is fast algorithm.

Bcrypt: It's very slow hashing algorithm.

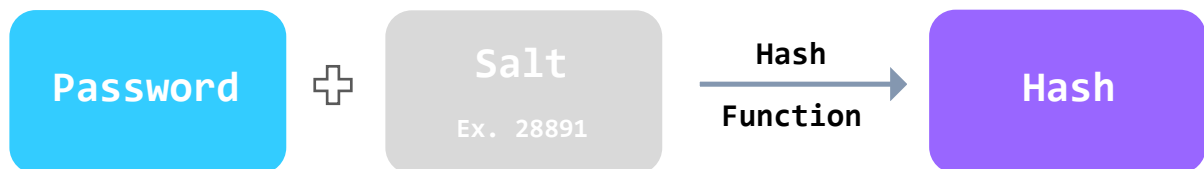
It's one of the industry standard hashing algorithms that
developers use to keep their users passwords safe.

while you can calculate 20 billion MD5 hashes per second,
bcrypt can only calculate 17,000 bcrypt hashes per second.

Md5 hashed password can be cracked within 3 sec, where as bcrypt needs 8 months.

To make bcrypt more secure, it contains salt rounds.

More rounds you do saltier, more secure your password.



In first round we add salt to password and generate hash. In second round we add salt to the hash which generated in round 1 and we add same salt as before and then running again to bcrypt and we get different hash this process continues until specified rounds.

Every year processor are going faster. So we've to increase number of round according to it. As number of round increases it takes double time to create hash.

In DB we store user's username, salted hashed password, randomly generated salt and salting rounds.

So at login when user types password we use salt from DB and processing typed password according to salt rounds and make hash and compare new hash with DB's password hash.

Bcrypt needs latest stable version of node.

If you want to update or degrade your node version then 1st install [nvm](#) and then type

Nvm install nodeVersion

```
const bcryptjs = require ("bcryptjs");  
const saltRounds = 10;
```

```
app.post("/register", (req, res) => {  
  bcryptjs.hash(req.body.password, saltRounds, (err, hash) => {  
    const newUser = new User({  
      email: req.body.username,  
      password: hash  
    });  
  
    newUser.save((err) => {  
      if (err)  
        console.log(err);  
      else  
        res.render("secrets");  
    });  
  });  
});
```

It stores password as,

```
{  
  "_id" : ObjectId("63ee46c7bd61499572a7201b"),  
  "email" : "nd@bcryptjs.com",  
  "password" : "$2a$10$9Q8kC99FYoh8Sdu0ikVKEuXp.TWHfAr002uClAPiu",  
  "__v" : NumberInt(0)  
}
```

Level 5: Cookies and Sessions

When you go to amazon site (without login) and add a item in the cart but because of some distraction you not buy that item and close browser.

After revisiting amazon site your product in the cart is still shows, it's because of Cookie.

After deleting cookies of amazon from setting and refresh amazon then your cart content will be gone.

Name
session-id
Content
259-6879970-7056432
Domain
.amazon.in

Here content id target to product in your cart.

After get request to amazon.in, server sends html, css, js. If you add an item to cart then it makes post request to server. And in response amazon server creates cookie of that data of item and cookie is send to browser and said to save.

Next time when we make get request to amazon then cookie gets sent along with get request to allow server to be able to identify who I'm and see if I had any previous sessions on amazon

session is a period of time when a browser interacts with a server.

When you login to website, session is start at that time and your cookie is created. It contain information of user credentials that says this user is logged in and has been successfully authenticated. So it don't say to log in again and again.

When you log out this cookie is deleted.

Passport:

Install packages: passport, passport-local, passport-local-mongoose, express-session NOT express-sessions

```
$ npm i passport passport-local passport-local-mongoose express-session
```

1st read doc of [express-session](#).

You don't need to require passport-local because it's dependency of passport and passport-local-mongoose

```
const session = require ("express-session");  
const passport = require("passport");  
const passportLocalMongoose = require ("passport-local-mongoose");
```

Then

```
app.use (session ({  
  secret: "Our Little Secret.",  
  resave: false,  
  saveUninitialized: false  
}))
```

To use passport you've to initialize it first.

```
app.use (passport.initialize());
```

Now tell our app to use passport to set up our session.

```
app.use (passport.session());
```

```
userSchema.plugin (passportLocalMongoose)
```

This plugin used to hash and salt passwords and to save in DB.