

Module Pattern in NodeJS -

Module is a mechanism for splitting JS programs into separate manageable chunk called module, that can be imported whenever needed.

Inside Nodejs we have two mechanism to prepare a module

- (i) CJS (traditional mechanism) - Common JS Module
- (ii) ESM - ES6 Module

CJS - Demo

searching.js

```
...
1 const linearSearch = function search(arr
2   , x) {
3     for (let i = 0; i < arr.length; i
4    ++) {
5       if (arr[i] == x) return i;
6     }
7   }
8 const binarySearch = function search(arr
9   , x) {
10   let lo = 0, hi = arr.length - 1;
11   while (lo <= hi) {
12     let mid = lo + Math.floor((hi -
13     lo) / 2);
14     if (arr[mid] == x) return mid;
15     else if (arr[mid] < x) {
16       lo = mid + 1;
17     } else {
18       hi = mid - 1;
19     }
20   }
21   return undefined;
22 }
23 module.exports = {
24   linearSearch,
25   binarySearch
26 }
```

it's a shorthand used since the key & the value name is same
this can also be written as
ls : linearSearch
bs : binarySearch

To use the module in a different file, we can use the global module to export a bunch of functions

This object is now going to be exposed to the outer world

In any other file we can start using it

index.js

```
...
1 const { linearSearch: ls, binarySearch: bs } = require('./searching');
2 console.log(ls([3, 2, 1, 5, 4, 23, 6], 23))
```

Benefit of destructuring is that whenever function or property we only those property is getting in a variable

It's another global, it's kind of like an import syntax, whatever are the modules that we are exporting from the file (for now it's searching.js) in the particular path we will get them all and we are storing it inside a variable

whatever func
need only those properties to go →
used
Note: Destructuring means unpacking

2 other way to
export module

```
1 module.exports.insertionsort = function insertionsort(arr) {
2   for (let i = 1; i < arr.length; i++) {
3     let currElement = arr[i];
4     let j = i - 1;
5     while (j >= 0 && arr[j] > currElement) {
6       arr[j + 1] = arr[j]; // shifting the element to the right index
7       j--;
8     }
9     arr[j + 1] = currElement;
10  }
11 }
```

```
1 module.exports.insertionsort = insertionsort
2 module.exports.bubbleSort = bubblesort
3 module.exports.selectionSort = selectionsort
```

Now converting them TO ES6 Moduling

↳ There are two ways

adding extension .mjs

updating type of the module

By default nodejs treat the following as Common JS

→ .cjs → .js

So, for ES6 moduling we need to configure package.json file and need to put the key as the type we want If the key is "commonjs"

then we enabled common js module although commonjs is the default.

Now if we put type = "module", things start to change

Through .mjs

exporting module

```
1 export function insertionsort(arr) {
2   for (let i = 1; i < arr.length; i++) {
3     let currElement = arr[i];
4     let j = i - 1;
5     while (j >= 0 && arr[j] > currElement) {
6       arr[j + 1] = arr[j]; // shifting the element to the right index
7       j--;
8     }
9     arr[j + 1] = currElement;
10  }
11 }
```

importing the module

```
1 import searchingAlgo from './searching.js';
2 // import * as sorting from './sorting.mjs'; // it brings all the named export and club them in the variable sorting which is technically going to be a object and then we can access as sorting.bubblesort(arr);
3 import Mergesort, { bubblesort as bs } from './sorting.mjs';
```

defining path mentioning its getting imported from cjs file

default import decompress import with alias file type esm

default export

```
1 export default function mergesort(arr) {
2   return f(arr, 0, arr.length - 1);
3 }
```

named export and together in sorting

```

import * as sorting from './sorting.js';
// it brings all the named export and club them together
// and club them together

```

```

let arr = [5, 4, 3, 2, 1];
sorting.bubblesort(arr)
console.log(arr)

```

accessing the object

→ NPM - Node Package Manager

Multiple modules clubbed together to form a complete end to end functionality is called a package.

There are multiple packages we can find in NPM.

What NPM help us to do is - it help us to install the packages and even do dependency resolution

for example - let's say there's a package (A) that is dependent on package (B) might be using some functionality of it! but we don't know about it

When we install (A) it'll directly manage package

(B) also. It will do topological sorting of all the dependencies and install everything one-by-one

→ We can very easily download module by running the module name with npm

ex - **npm i dotenv**

The moment we enter this the module gets downloaded, when we check our code editor we don't just see the module we wanted we see

3 other things as well - (1) Node modules - it has all the dependencies present that our install module require

(ii) **Package.json** - It helps us to understand metadata about our project
for ex-

(1) Who this project is getting prepared for

- 1 (i) Who this project is getting prepared for
 - (ii) What are the dependent packages on the dependency
 - (iii) Are there any script configured for this project
- (iii) Package.lock.json - It has record of all the dependency of our installed module
- on - Name of the dependency
Version etc

We can prepare our very own package.json from very scratch what we need to do is execute a command "npm init" what npm init does is, it creates a package.json file for you technically saying it helps in creating npm project.

↓
project having a package.json file is a simple npm project

Project 0 - Telegram Bot

- ↳ npm init > basic config
- ↳ in package.json file add type = "module" or "commonjs" whichever modular pattern you want in your project. By default it's commonjs
- ↳ Copy telegraph npm & paste it, it'll install telegraph for me
- ↳ Create a file index.js
- ↳ How to use telegraph or any other node module?
It's the documentation we should always refer
- ↳ In the index.js file add

↳ In the index.js file add

```
const { Telegraf } = require('telegraf');
```

common
js

using absolute path

↳ Now in order to create a new bot, we need bot token's from telegram

→ All we need to do is open telegram and search for botfather

→ Botfather is a telegram bot that helps in getting telegram bot token.

What is bot token?

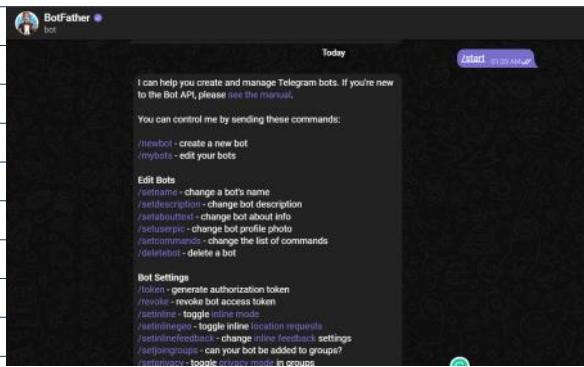
→ Token through which telegram will be able to say who owns this bot

→ A lot of time when you'll use 3rd party services they will give you similar token as API key, API token.

Using these token they are actually identifying who is using the particular app

→ In BOTFATHER → start anything with "/" treated as command.

it replies with multiple commands



↳ To create a new bot enter /newbot

username

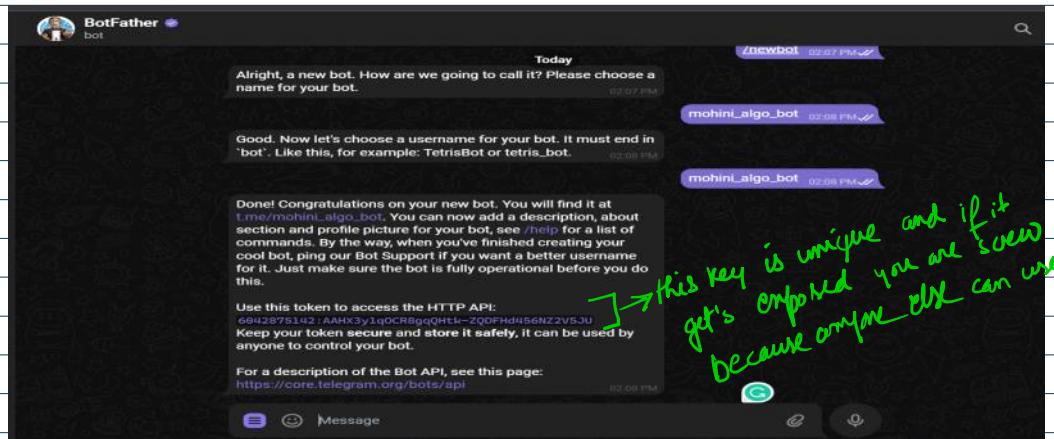
↳ It'll instantly ask to name the bot, enter the name of the bot whatever you want

.....

whatever you want

→ Done our bot is ready it gives a link through which anyone can access the bot

↳ With the done message we also get a API Key, through which we can customize the bot however we want now



So, there are mechanisms through which we can hide our secret key, we can actually store them in environment variable.

Every process we run has a key value pair in operating system and we can add more key value pairs. How to do that?

→ that's what the dotenv package does

so now npm install dotenv

→ Once dotenv is installed create a .env file inside the project

→ Inside the .env file we can write secret keys, password, emails etc with a name you want to use it as.

Now to import you can use both CJS version, ESM version depending on the project

Usage

Create a .env file in the root of your project:

```
S3_BUCKET="YOURS3BUCKET"
SECRET_KEY="YOURSECRETKEYGOESHHERE"
```

As early as possible in your application, import and configure dotenv:

```
require('dotenv').config()
console.log(process.env) // remove this after
```

CJS [

```
... or using ESM
```

ESM [

```
import 'dotenv/config'

That's it. process.env now has the keys and values you defined in your .env file:
```

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

...
s3.getBucketCors({Bucket: process.env.S3_BUCKET}, function(err, data) {})
```

To use it, we need to use "process.env"

→ .env variables

To use it, we need to use process.env



global, it helps to access all the env variables and our own variable that we want to use is BOT-TOKEN

∴ const bot = new Telegraph(process.env.BOT-TOKEN")

This is how we can actually hide our secrets

↳ How does .env file secure this file?

⇒ Whenever we push the code, we will not push the .env file using .gitignore file. But let's say you want to deploy it on the server.

On that server we have to manually make this .env file or we need to prepare a pipeline that can manually push the .env file

So, this is how we create a bot & we are good to go

→ Now, to start the bot

```
bot.start((ctx) => ctx.reply('Welcome'));
```

↓ telegram bot object

in the telegram bot chat if we enter /start, the bot will get activated

↳ to give a particular let's say /binarysearch

bot.command('binarysearch', (ctx) => ctx.reply('binarySearch'))
↓ event, it's for custom reply

↳ Configuring an emoji

bot.on('sticker'), (ctx) => ctx.reply('😊'),
↓ event

↳ At the end when the bot is launched

bot.launch()

↳ To handle messages we will bind them inside try catch

Code Snippet -

```
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {
        if(prices.size() < 2) return 0;
        if(k == 0) return 0;
        if(k >= prices.size()) return maxProfit(prices);
        if(prices.size() <= 1000) {
            vector<int> dp(prices.size(), 0);
            for(int i = 0; i < k; i++) {
                for(int j = 1; j < prices.size(); j++) {
                    dp[j] = max(dp[j], dp[j-1] + prices[j] - prices[j-1]);
                }
            }
            return dp[prices.size() - 1];
        } else {
            vector<int> dp(prices.size(), 0);
            for(int i = 0; i < k; i++) {
                for(int j = 1; j < prices.size(); j++) {
                    dp[j] = max(dp[j], dp[j-1] + prices[j] - prices[j-1]);
                }
            }
            return dp[prices.size() - 1];
        }
    }
};

class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {
        if(prices.size() < 2) return 0;
        if(k == 0) return 0;
        if(k >= prices.size()) return maxProfit(prices);
        if(prices.size() <= 1000) {
            vector<int> dp(prices.size(), 0);
            for(int i = 0; i < k; i++) {
                for(int j = 1; j < prices.size(); j++) {
                    dp[j] = max(dp[j], dp[j-1] + prices[j] - prices[j-1]);
                }
            }
            return dp[prices.size() - 1];
        } else {
            vector<int> dp(prices.size(), 0);
            for(int i = 0; i < k; i++) {
                for(int j = 1; j < prices.size(); j++) {
                    dp[j] = max(dp[j], dp[j-1] + prices[j] - prices[j-1]);
                }
            }
            return dp[prices.size() - 1];
        }
    }
};
```

Functional Chat -

