https://www.geeksforgeeks.org/how-to-deploy-node-js-app-on-heroku-from-github/

Node.js Introduction

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications. ; open-source and cross-platform JavaScript runtime environment which allows backend programming through JavaScript

1. Easy—Node.js is quite easy to start with. It's a go-to choice for web development beginners. With a lot of tutorials and a large community—getting started is very easy.
2. Scalable—It provides vast scalability for applications. Node.js, being single-threaded, is capable of handling a huge number of simultaneous connections with high throughput.
3. Speed—Non-blocking thread execution makes Node.js even faster and more efficient.
4. Packages—A vast set of open-source Node.js packages is available that can simplify your work. There are more than one million packages in the NPM ecosystem today.
5. Strong backend—Node.js is written in C and C++, which makes it speedy and adds features like networking support.
6. Multi-platform—Cross-platform support allows you to create SaaS websites, desktop apps, and even mobile apps, all using Node.js.
7. Maintainable—Node.js is an easy choice for developers since both the frontend and backend can be managed with JavaScript as a single language.

Install Node.js on Mac without admin privileges:

https://medium.com/javascript-first/how-to-install-npm-nvm-on-mac-without-admin-sudo-or-homebrew-cef757275bbb

https://blog.teamtreehouse.com/install-node-js-npm-mac +
https://superuser.com/questions/619498/can-i-install-homebrew-without-sudo-privileges

demo.js:

```
let name = 'Pesho';

console.log(`My name is ${name}`);
```

- >

```
[11:04:43] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/1-intro-to-nodejs ] $ node demo.js
My name is Pesho
```

Initialize a node.js project - through Node Package Manager - npm:

```
npm init -y
```

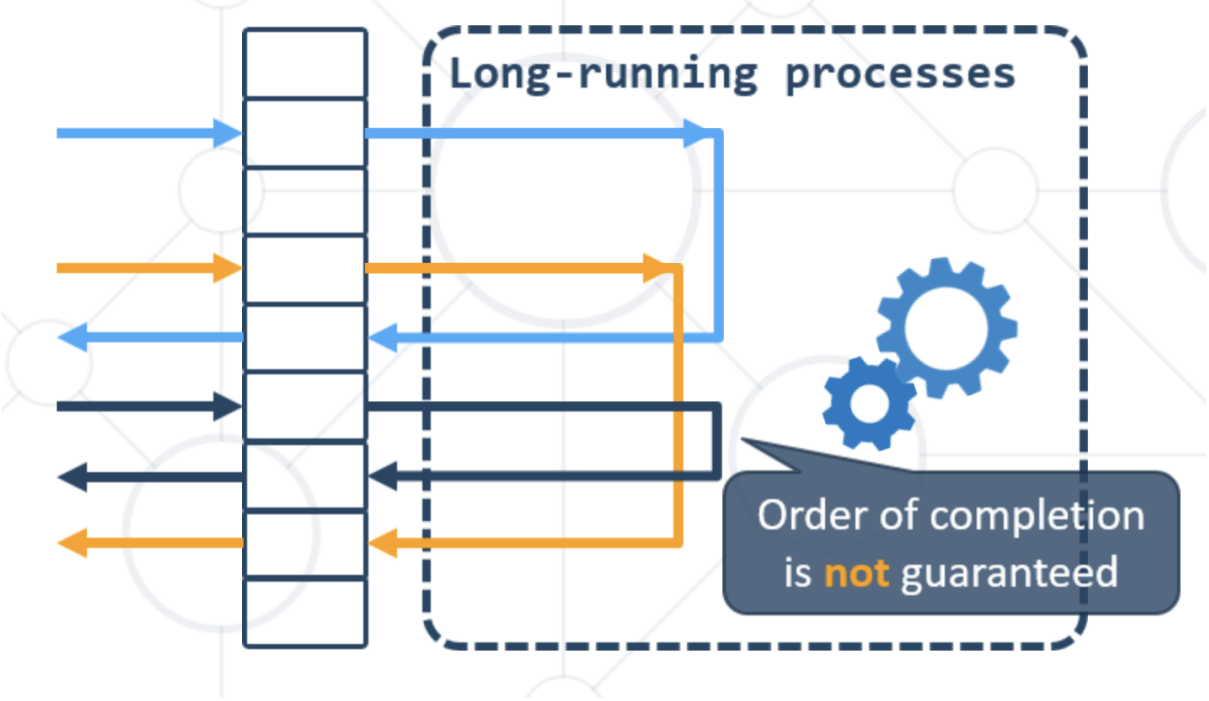
package.json - the file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies. ; it is used as a manifest, storing information about applications, modules, packages, and more. ; It is a JSON file that lives in the root directory of your project. Your package.json holds important information about the project. It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application. ; which node versions to be used, the scripts which can be executed on the project, the repository of the project and etc:

https://nodesource.com/blog/the-basics-of-package-json

Event Loop

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible. ; an event-listener which functions inside the NodeJS environment and is always ready to listen, process, and output for an event. ;

1. Event loop is an endless loop, which waits for tasks, executes them and then sleeps until it receives more tasks.
2. The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
3. The event loop allows us to use callbacks and promises.
4. The event loop executes the tasks starting from the oldest first.



https://blog.risingstack.com/node-js-at-scale-understanding-node-js-event-loop/ - explanation with the call stack

!!!!

```
function func() {
    console.log('start');

    setTimeout(() => {
        console.log('executed');
    }, 0)

    console.log('end');
}

func();
```

- >

start
end
executed

due to:

The event loop executes tasks from the event queue only when the call stack is empty

1) console.log(‘start’);

2) console.log('executed'); is registered in the Event Callback

3) console.log(‘end’)

4) the call stack is empty, so the console.log('executed') is executed !!!!

Modules

In Node.js, Modules are the blocks of encapsulated code that communicates with an external application on the basis of their related functionality. Modules can be a single file or a collection of multiples files/folders. The reason programmers are heavily reliant on modules is because of their re-usability as well as the ability to break down a complex piece of code into manageable chunks. ;

- Allow larger apps to be split and organized
- Each module has its own context
- It cannot pollute the global scope

Node.js includes three types of modules

Core Modules: Node.js has many built-in modules that are part of the platform and comes with Node.js installation. These modules can be loaded into the program by using the require function.

Local Modules: Unlike built-in and external modules, local modules are created locally in your Node.js application

Two ways of importing modules:

- **common.js**: parses automatically JSON to JavaScript objects

1) default exporting (in the other module it can be calc or anything else we want):

calc.js:

```
function calc(a,b) {
    return a + b;
}

module.exports = calc;
```

calc-application.js:

```
const calculator = require('./calc')
```

```
function init() {
    let result = calculator(2,3);
    console.log(result);
}
```

2) named export (it can only be calc):

```
function calc(a,b) {
    return a + b;
}

exports.calc = calc;
```

```
const { calc } = require('./calc')

function init() {
    let result = calc(2,3);
    console.log(result);
}
```

https://nodejs.org/api/modules.html

- **es6 modules:** does not parse automatically JSON to JavaScript objects

```
function calc(a,b) {
    return a + b;
}

export { calc }
```

```
import { calc} from "./calc";

function init() {
    let result = calc(2,3);
    console.log(result);
}
```

https://nodejs.org/api/esm.html

Third-Party Modules: Third-party modules are modules that are available online using the Node Package Manager(NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are mongoose, express, angular, and react. https://www.npmjs.com/

Example Common Module:

https://nodejs.org/docs/latest-v16.x/api/url.html

url:

const url = require('url');

The url module provides utilities for URL resolution and parsing

```
const url = require('url');

let softUniURL =
url.parse('https://softuni.bg/trainings/3473/angular-july-2021');

console.log(softUniURL);
```

```
Url {
  protocol: 'https:',
  slashes: true,
  auth: null,
  host: 'softuni.bg',
  port: null,
  hostname: 'softuni.bg',
  hash: null,
  search: null,
  query: null,
  pathname: '/trainings/3473/angular-july-2021',
  path: '/trainings/3473/angular-july-2021',
  href: 'https://softuni.bg/trainings/3473/angular-july-2021'
}
```

!!!! There is also **URL**, which is included in the basic nodeJS library by default (without import):

```
let softUniURL2 = new
URL('https://softuni.bg/trainings/3473/angular-july-2021');

console.log(softUniURL2);
```

->

```
URL {
  href: 'https://softuni.bg/trainings/3473/angular-july-2021',
```

```
  origin: 'https://softuni.bg',
  protocol: 'https:',
  username: '',
  password: '',
  host: 'softuni.bg',
  hostname: 'softuni.bg',
  port: '',
  pathname: '/trainings/3473/angular-july-2021',
  search: '',
  searchParams: URLSearchParams {},
  hash: ''
}
```

!!!!

NodeJS Web Server Basics

Web servers are software products that use the operating system to handle web requests and serve Web content ; the requests are redirected to other software products (ASP.NET, PHP, etc.), depending on the webserver settings

Creating simple NodeJS web server with the HTTP module:

index.js:

```
const http = require('http');
const port = 5000;

const server = http.createServer((req, res) => {
    res.write('Hello from NodeJS!');
    res.end();
});

server.listen(port, () => {
    console.log(`Server is listening on port ${port}...`);
});
```

!!! By default you must restart the web server for any changes to take effect

However, we may install and use **nodemon** - the third-party module will monitor for any changes in our source code and automatically restart our server.:

https://nodemon.io/

1) npm i nodemon

2) in package.json ->

```
"scripts": {
    "start": "nodemon index.js",
```

3) npm start !!!!

!!! We must end the res with res.end, otherwise no content will be served to the visitor !!!

Basic way to serve content based on the URL in the request:

```
  if (req.url === '/cats') {
        res.write('Some Cats');
  } else if (req.url === '/dogs') {
        res.write('Some Dogs');
  } else {
        res.write('Hello from NodeJS!');
  }
```

The request wrapper (req) has many properties, which we may use, such as:

```
console.log('Method: ', req.method);
console.log('URL: ', req.url);
console.log('Headers: ', req.headers);
```

the console log on the node.js server (not on the browser):

```
Method:  GET
URL:  /dogs
Headers:  {
  host: 'localhost:5000',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'sec-ch-ua': '" Not A;Brand";v="99", "Chromium";v="101", "Google
Chrome";v="101"',
  'sec-ch-ua-mobile': '?0',
  'sec-ch-ua-platform': '"macOS"',
  'upgrade-insecure-requests': '1',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/101.0.4951.64
Safari/537.36',
  accept:
'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9',
  'sec-fetch-site': 'none',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-user': '?1',
  'sec-fetch-dest': 'document',
  'accept-encoding': 'gzip, deflate, br',
  'accept-language':
'en-US,en;q=0.9,bg;q=0.8,ja;q=0.7,ar;q=0.6,sl;q=0.5',
```

```
  cookie: 'Webstorm-f2b156=8a94a01a-6b01-41b9-b2ff-dd2f71fab48d'
}
```

With the response wrapper we may send content to the client and change the response headers, including for example the content type of the response:

```
  res.writeHead(200, {
        'Content-type': 'text/plain'
    });
```

If we set the content type to be html, we can set HTML in our response:

```
 res.writeHead(200, {
        'Content-type': 'text/html'
    });
 res.write('<h1>Hello from NodeJS!</h1>');
```

we can serve CSS from JavaScript string by setting the content-type of the response to text/css:

index.js:

```
const http = require('http');
const homePage = require('./home');
const siteCSS = require('./resources/content/styles/site.js');

const server = http.createServer((req, res) => {
    if (req.url == '/resources/content/styles/site.css') {
        res.writeHead(200, {
            'Content-type': 'text/css'
        });
        res.write(siteCSS);
    } else {
        res.writeHead(200, {
            'Content-type': 'text/html'
        });

    }
    res.end();
}).listen(5000, () => {
    console.log('Server is listening on port 5000...')
})
```

site.js:

```
module.exports = `
@import
url('https://fonts.googleapis.com/css?family=Lato:400,700&display=swap')
;

* {
    margin: 0;
    padding: 0;
}

html, body {
    font-family: 'Lato', sans-serif;
    background-color: #F2F2F2;
    color: #333;
}

..........
`;
```

Publish-Subscribe Patterns

The Publish/Subscribe pattern, also known as pub/sub, is an architectural design pattern that provides a framework for exchanging messages between publishers and subscribers. This pattern involves the publisher and the subscriber relying on a message broker that relays messages from the publisher to the subscribers. The host (publisher) publishes messages (events) to a channel that subscribers can then sign up to.

the pattern is used to communicate messages between different system components without them knowing anything about each other's identity

!!! we can achieve loose coupling in this way (**loose coupling** is an approach to interconnecting the components in a system or network so that those components, also called elements, depend on each other to the least extent practicable / are not that connected between each other.). !!!

Advantages:

1) Decouple and Scale Independently
2) Eliminate Polling
3) Simplify Communication

Example Subscribe and Unsubscribe eventBus

eventBus.js:

```
const subscribers = {};

//addEventListener
exports.subscribe = (eventType, callback) => {

    if (!subscribers[eventType]) {
        subscribers[eventType] = [];
    }

    subscribers[eventType].push(callback);

    return () => {
        subscribers[eventType] = subscribers[eventType].filter(x => x !=
callback);
    }
};
```

```
// Emit, trigger
exports.publish = (eventType, ...params) => {

    subscribers[eventType].forEach(s => s.apply(null, params));

};
```

demo.js:

```
const eventBus = require('./eventBus');

// the eventBus.subscribe will return a function to the
firstSayHelloUnsubscribe
let firstSayHelloUnsubscribe = eventBus.subscribe('say-hello', (name,
secondName) => { console.log('event say-hello executed - ' + name + " "
+ secondName) });
eventBus.subscribe('say-hello', (name, secondName) => {
console.log('event say-hello executed second time -' + name + " " +
secondName) });
eventBus.subscribe('say-bye', (name) => { console.log('event say-bye
executed - ' + name)});

eventBus.publish('say-hello', 'Gosho', 'Ivan');
firstSayHelloUnsubscribe(); // it will execute the returned function
from subscribe -
//         subscribers[eventType] = subscribers[eventType].filter(x => x
!= callback);
eventBus.publish('say-hello', 'Pesho'); // only the executed second time
subscription will be executed
eventBus.publish('say-bye', 'Gosho');
```

Events Module

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") emit named events that cause Function objects ("listeners") to be called.

All objects that emit events are instances of the EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

The eventEmitter.on() method is used to register listeners, while the eventEmitter.emit() method is used to trigger the event.

Simple Events Example:

```
const EventEmitter = require('events');

const eventEmitter = new EventEmitter();

eventEmitter.on('sing', (songTitle) => {
    console.log(`${songTitle} - LaLaLa`);
});

eventEmitter.emit('sing', 'Nothing Else Matters');
```

Streams

Collections of data that is not available at once - data may come continuously in chunks. They are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way. Streams basically provide two major advantages over using other data handling methods:

1) **Memory efficiency**: you don't need to load large amounts of data in memory before you are able to process it
2) **Time efficiency**: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

!!! What makes streams unique, is that instead of a program reading a file into memory all at once like in the traditional way, streams read chunks of data piece by piece, processing its content without keeping it all in memory. !!!

Types:

1) Readable - can only read - process.stdin
2) Writable - can only be written to - process.stdout
3) Duplex - both Readable and Writable - TCP sockets
4) Transform - the output is computed (zlib, crypto)

Readable:

functions:
read()
pause()
resume()

events:
data - chunk is available for reading
end - no more data
error - an exception has occurred

An example of reading a file:

```
const fs = require('fs');

const readStream = fs.createReadStream('./largeFile.txt', {encoding:
'utf-8'});
```

```
readStream.on('data', (chunk) => {
    console.log(chunk);
});
```

without encoding:

```
[10:51:23] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/2-streams-utilities/streams ] $ node demo.js
<Buffer 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 6f 6c 6f 72 20 73 69 74
20 61 6d 65 74 2c 20 63 6f 6e 73 65 63 74 65 74 75 72 20 61 64 69 70 69
73 63 69 6e 67 ... 65486 more bytes>
<Buffer 62 68 20 75 74 2c 20 70 6c 61 63 65 72 61 74 20 6a 75 73 74 6f
2e 20 43 75 72 61 62 69 74 75 72 20 70 65 6c 6c 65 6e 74 65 73 71 75 65
20 73 65 6d 20 ... 46638 more bytes>
```

with encoding:

```
[10:51:25] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/2-streams-utilities/streams ] $ node demo.js
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam congue
metus nibh, at faucibus ante ullamcorper in. Nunc mollis orci in magna
rutrum, et tincidunt dui dictum. Quisque vitae lacus vitae neque dapibus
posuere non ac mauris. Proin ullamcorper facilisis odio tempor laoreet.
Nulla in turpis neque. Pellentesque risus elit, tristique a gravida ut,
vulputate non nisi. Aliquam cursus scelerisque felis, condimentum
eleifend ligula bibendum sed. Vivamus quis ornare dui. Phasellus sit
amet tellus vel neque vulputate accumsan nec in arcu. Proin id dapibus
neque. Morbi vulputate scelerisque ligula id fermentum. Pellentesque
lobortis malesuada nibh, et varius elit pulvinar sit amet. Phasellus
quis consectetur dui.
```

a function, which is completed, when the stream is ended:

```
readStream.on('end', () => {
    console.log('Finished');
});
```

Writable:

functions:
write()
end()

events:
drain - stream can receive more data
finish - all data has been flushed
error

Example of writing to a file (the file does not need to exist):

```
const writeStream = fs.createWriteStream('./copyFile.txt', {encoding:
'utf-8'});

writeStream.write('Hello World!');
writeStream.write('\n')
writeStream.write('Hello World 2');

writeStream.on('finish', () => console.log('File is Saved'));

writeStream.end();
```

Basic copy-paste from one file to other one:

```
const fs = require('fs');

const readStream = fs.createReadStream('./largeFile.txt', {encoding:
'utf-8'});
const writeStream = fs.createWriteStream('./copyFile.txt', {encoding:
'utf-8'});


readStream.on('data', (chunk) => {
    writeStream.write(chunk);
});

readStream.on('end', () => {
    writeStream.end();
    console.log('Finished');
});

writeStream.on('finish', () => console.log('File is Saved'));
```

Alternative to:

```
readStream.on('data', (chunk) => {
    writeStream.write(chunk);
});

readStream.on('end', () => {
    writeStream.end();
    console.log('Finished');
});
```

is

```
readStream.pipe(writeStream);
```

Example of Transform Stream (gzipping a file):

```
const { createGzip } = require('zlib');
const { pipeline } = require('stream');
const {
  createReadStream,
  createWriteStream
} = require('fs');

const gzip = createGzip();
const source = createReadStream('input.txt');
const destination = createWriteStream('input.txt.gz');

pipeline(source, gzip, destination, (err) => {
  if (err) {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  }
});
```

FS Module

The fs module provides a lot of very useful functionality to access and interact with the file system. One peculiar thing about the fs module is that all the methods are asynchronous by default, but they can also work synchronously by appending Sync:

fs.rename()
fs.renameSync()
fs.write()
fs.writeSync()

Example Read File via fs (sync):

```
const fs = require('fs');

const text = fs.readFileSync('./text.txt', { encoding: 'utf-8'});

console.log(text);
```

via fs (async):

```
const text = fs.readFile('./text.txt', {encoding: "utf-8"}, (err,data)
=> {
    if (err) {
        console.log(err);
        return;
    }

    console.log(data);
});
```

via promises:

```
const fsp = require('fs/promises');

fsp.readFile('./text.txt', {encoding: 'utf-8'})
    .then((data) => console.log(data));
```

Listing a directory:

```
fs.readdir('./', (err, data) => {
    if (err) {
        return;
    }

    console.log(data);
});
```

Rename a file / directory:

```
fs.rename('./oldName', './newName', err => {

  if (err) {

    console.log(err);

    return;

  }

});
```

Save data to a file:

```
data="Praesent justo leo, varius tempor odio at, pharetra posuere magna. Donec in tincidunt libero. Donec elementum sem sit amet erat efficitur consequat. Nulla eu nisl sed lorem lacinia sodales. Aliquam in laoreet felis, et congue libero. Vestibulum in sapien bibendum, sagittis nibh ut, placerat justo."

fsp.writeFile('./text2.txt', data, { encoding: "utf-8"})
    .then(() => console.log('Finished'))
    .catch((err) => {
        console.log(err);
    });
```

Express

Express is the most popular Node web framework, and is the underlying library for a number of other popular Node web frameworks. It provides mechanisms to:

1) Write handlers for requests with different HTTP verbs at different URL paths (routes).
2) Integrate with "view" rendering engines in order to generate responses by inserting data into templates.
3) Set common web application settings like the port to use for connecting, and the location of templates that are used for rendering the response.
4) Add additional request processing "middleware" at any point within the request handling pipeline.

Basic Express Server:

1) npm i express

2) index.js:

```
const express = require('express');

const app = express();

app.get('/', (req, res) => {
   res.write('Hello World');
   res.end();
});

app.listen(5000, () => {
    console.log('Server is listening on port 5000...');
})
```

or simpler:

```
const express = require('express');

const app = express();

app.get('/', (req, res) => {
    res.send('Hello World');
```

```
});

app.listen(5000, () => {
    console.log('Server is listening on port 5000...');
});
```

Router in Express

```
app.METHOD(PATH, HANDLER/ACTION);
```

such as:

```
app.get('/', (req, res) => {
    res.send('Hello World');
});
```

!!! PATH can contain special characters or regex:

```
app.get(/.*fly$/, (req, res) => {

  res.send('butterfly, dragonfly') })
```

```
app.get(/[0-9]+/, (req, res) => {
    res.send('This route starts with number');
});
```

Custom 404 Page Not Found (at the end of the app chain):

```
app.all('*', (req, res) => {
    res.status(404);
    res.send('This page was not found');
});
```

!!! Always include the specific PATH at the top of the app chain while the more generic PATH should be at the bottom as the router will try to match the PATH from top to bottom !!!

!!! PATH + METHOD = ENDPOINT !!!

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, with the name of the route parameter specified in the path as their respective keys:

```
app.post('/cats/:catName', (req, res) => {
    console.log(req.params);

    res.send('Add New Cat To The Collection');
})
```

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params)
})
```

You may also use regular expressions to set up validation of the parameters:

```
app.get('/users/:userId(\\d+)', (req, res) => {

  const paramsObj = req.params

  res.send(paramsObj) });
```

Download a file

1) Via streams and file system:

```
app.get('/download', (req,res) => {
    res.writeHead(200, {
        'content-disposition': 'attachment; filename="sample.pdf"'
    });

    const readStream = fs.createReadStream('sample.pdf');

    readStream.on('data', (data) => {
        res.write(data);
    });

    readStream.on('end', () => {
        res.end();
    });

});
```

Alternatively:

```
app.get('/download', (req,res) => {
    res.writeHead(200, {
        'content-disposition': 'attachment; filename="sample.pdf"'
    });

    const readStream = fs.createReadStream('sample.pdf');

    readStream.pipe(res);

});
```

!!!! If we want the client to open the file, but not download it

```
'content-type': 'application/pdf',
'content-disposition': 'inline'
```

2) Through Express:

```
app.get('/express-download', (req,res) => {

    res.download('./sample.pdf');

});
```

!!! if we want the client to open the file, but not download it (we must provide absolute path to the file, not relative one):

```
app.get('/express-download', (req,res) => {

    res.sendFile(__dirname + '/sample.pdf');

});
```

Serving static files (such as images, CSS and etc.) from a folder:

1) Basic way:

```
const path = require('path');

.................................

app.get('/img/:imgName', (req, res) => {
    res.sendFile(path.resolve('/img', req.params.imgName));
});
```

2) Express:

```
app.use('/static', express.static('./public'));
```

URL: http://localhost:5000/static/img/Cat03.jpeg

Redirect a request to particular page:

1) via streams:

```
app.get('/redirect', (req, res) => {
    res.writeHead(301, {
        'Location' : '/cats'
    });

    res.end();
});
```

2) via Express:

```
app.get('/express-redirect', (req,res) => {
    res.redirect('/cats');
});
```

Middleware

Middleware is software containing functions that execute during the request-response cycle and have access to both the request object (req) and the response object (res). Middleware is executed during the window between when a server receives a request and when it sends a response.Express middleware includes application-level, router-level, and error handling functionality and can be built-in or from a third party. Since Express.js has limited functionality of its own, an Express app is largely comprised of multiple middleware function calls.

Example of Basic Middleware:

1) middlewares.js

```
const cats = [];

exports.catMiddleware = (req, res, next) => {
    req.cats = cats;

    next();
};
```

2) index.js

```
const { catMiddleware } = require('./middlewares');

.....................

app.get('/cats', catMiddleware, (req, res) => {
    if (req.cats.length < 1){
        res.send('No cats here!');
    } else {
        res.send(req.cats.join(', '));
    }
});
```

!!! If we want the Middleware to be used on all routes / on application level we set:

```
app.use(catMiddleware);
```

at the top of the file and we do not need to include the Middleware in the requests (app.get)

Templating

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.

!!! Server will combine data + template and then send the ready HTML to the client !!!

Basic Handlebars for Express Example:

1) npm i express-handlebars

2) index.js:

```
const handlebars = require('express-handlebars');

.................

app.engine('hbs', handlebars.engine({
    extname: 'hbs'
})); // register engine
app.set('view engine', 'hbs'); // set which engine we will use

app.get('/:name?', (req, res) => {
    res.render('home', {name: req.params.name || 'Guest'});

});
```

3) Create views folder, than layouts folder inside views and finally create main.hbs (constant template) file inside it:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
```

```
    <title>Express Templating</title>
</head>
<body>
{{{body}}}
</body>
<footer>
    <span>All Rights Reserved! &copy;</span>
</footer>
</html>
```

4) In views create home.hbs:

```
<h1>Hello World</h1>

<h2>From {{name}}</h2>
```

Results:

URL: http://localhost:5000/

Hello World
From Guest
All Rights Reserved! ©

URL: http://localhost:5000/Doni

Hello World
From Doni
All Rights Reserved! ©

Foreach in Express-Handlebars

1) index.js:

```
const users = [
    { name: 'Pesho', age: 20},
    { name: 'Tosho', age: 25},
    { name: 'Penka', age: 40},
];

.................

app.get('/:name?', (req, res) => {
     res.render('home', {
        name: req.params.name || 'Guest',
        users
});

});
```

2) views/home.hbs:

```
<ul>
    {{#each users}}
    <li>{{name}} : {{age}}</li>
    {{/each}}
</ul>
```

If/else in Express-Handlebars:

1) index.js:

```
  res.render('home', {
        name: req.params.name || 'Guest',
        users,
        isAuth: false
});
```

2) views/home.hbs:

```
{{#if isAuth}}
    <h3>Welcome user</h3>
{{else}}
```

```
    <h3>Welcome guest</h3>
{{/if}}
```

Partials - template which can be used in other templates

1) views/partials/**user**.hbs:

```
<li>{{name}} : {{age}} years old</li>
```

2) views/home.hbs:

```
 {{#each users}}
        {{>user}}
    {{/each}}
```

!!! {{}} - can escape code, which can be executed, {{{}}} - does not escape the code and it is going to be run !!!

!!! npm i -D - install a software which will be used only for development !!!

!!! req.params - get the parameters
req.query - get the query strings (?name=George)
req.body - get the body of sent data (forms) !!!!

!!!! To read sent data - add the below to index.js:

```
app.use(express.urlencoded({ extended: false}));
```

extended: true -> https://www.npmjs.com/package/qs

!!!!

NoSQL and SQL

NoSQL = Non-Relational - Non-relational databases (often called NoSQL databases) are different from traditional relational databases in that they store their data in a non-tabular form. Instead, non-relational databases might be based on data structures like documents (object-like). A document can be highly detailed while containing a range of different types of information in different formats. This ability to digest and organize various types of information side-by-side makes non-relational databases much more flexible than relational databases. - MangoDB, Redis, etc.

There are four common types of NoSQL databases, specifically:

1) Key-value store model—the least complex NoSQL option, which stores data in a schema-less way that consists of indexed keys and values. Examples: Cassandra, Azure, LevelDB, and Riak.
2) Column store—wide-column store stores data tables as columns rather than rows. It's more than just an inverted table—sectioning out columns allows for excellent scalability and high performance. Examples: HBase, BigTable, HyperTable.
3) Document database—taking the key-value concept and adding more complexity, each document in this type of database has its own data and unique key used to retrieve the data. It's a great option for storing, retrieving, and managing data that's document-oriented but still semi-structured. Examples: MongoDB, CouchDB.
4) Graph database—have data that is interconnected and best represented as a graph. This method is capable of lots of complexity. Examples: Polyglot, Neo4J.

MongoDB—the most popular NoSQL system, especially among startups. A document-oriented database with JSON-like documents in dynamic schemas instead of relational tables used on the back end of sites like Craigslist, eBay, Foursquare. It's open-source, so it's free, with good customer service.
Apache's CouchDB—is a true DB for the web. It uses the JSON data exchange format to store its documents; JavaScript for indexing, combining, and transforming documents; and HTTP for its API.
Redis—a popular key-value database.
Riak—an open-source key-value store database written in Erlang. It has fault-tolerance replication and automatic data distribution built-in for excellent performance.
HBase—another Apache project, developed as a part of Hadoop, this open-source, non-relational "column store" NoSQL DB is written in Java and provides BigTable-like capabilities.
Oracle NoSQL—Oracle's entry into the NoSQL category.

Apache's Cassandra DB—born at Facebook, Cassandra is a distributed database that's great at handling massive amounts of structured data. Anticipating a growing application? Cassandra is excellent at scaling up. Examples: Instagram, Comcast, Apple, and Spotify.

Relational Databases - A relational database is a collection of data items with pre-defined relationships between them. These items are organized as a set of tables with columns and rows. Tables are used to hold information about the objects to be represented in the database. Each column in a table holds a certain kind of data and a field stores the actual value of an attribute. The rows in the table represent a collection of related values of one object or entity. Each row in a table could be marked with a unique identifier called a primary key, and rows among multiple tables can be made related using foreign keys. Almost all of the relational databases use SQL. - Oracle, MariaDB, MySQL

SQL = Structured Query Language - a programming language designed to get information out of and put it into a relational database. Queries are constructed from a command language that lets you select, insert, update and locate data.

Oracle—an object-relational DBMS written in the C++ language. If you have the budget, this is a full-service option with great customer service and reliability. Oracle has also released an Oracle NoSQL database.
MySQL—the most popular open-source database, excellent for CMS sites and blogs.
Google Cloud SQL—a fully managed relational database service for MySQL, PostgreSQL, and SQL Server offered through Google.
Sybase—a relational model database server product for businesses primarily used on the Unix OS, the first enterprise-level DBMS for Linux.
Amazon RDS—Amazon RDS for SQL Server makes it easy to set up, operate, and scale SQL Server deployments in the cloud.
Microsoft Azure—a cloud computing platform that supports any operating system while letting you store, compute, and scale data in one place. A recent survey even put it ahead of Amazon Web Services and Google Cloud Storage for corporate data storage.
MariaDB—an enhanced, drop-in version of MySQL.
MS SQL Server—a Microsoft-developed RDBMS for enterprise-level databases that supports both SQL and NoSQL architectures.
PostgreSQL—an enterprise-level, object-relational DBMS that uses procedural languages like Perl and Python, in addition to SQL-level code.

!!! Why to use databases instead of files - security, performance/speed and flexibility !!!

https://www.integrate.io/blog/the-sql-vs-nosql-difference/

https://pandorafms.com/blog/nosql-vs-sql-key-differences/

MongoDB

Installing MongoDB for MAC:

https://stackoverflow.com/questions/57856809/installing-mongodb-with-homebrew

https://www.knowledgehut.com/blog/web-development/install-mongodb-mac

MongoDB desktop client:

https://www.mongodb.com/try/download/compass

!!! MongoDB Database - > Collections - > Documents

vs

MySQL Database - > Tables - > Rows !!!!

Example MongoDB usage:

1) **mongo** in the terminal

2) **show dbs**

3) **use [database_name]**

4) **show collections**

5) **db.[collection_name].find()**

Result:

```
> db.themes.find()
{ "_id" : ObjectId("5fa64a9f2183ce1728ff371a"), "subscribers" : [
ObjectId("5fa64a072183ce1728ff3719"),
ObjectId("5fa64ca72183ce1728ff3726"),
ObjectId("5fa64c1f2183ce1728ff3723"),
ObjectId("5fa64b972183ce1728ff3720"),
```

```
ObjectId("5fa65bac2183ce1728ff372b") ], "posts" : [
ObjectId("5fa64a9f2183ce1728ff371b"),
ObjectId("5fa65be82183ce1728ff372d") ], "themeName" : "Angular 10",
"userId" : ObjectId("5fa64a072183ce1728ff3719"), "created_at" :
ISODate("2020-11-07T07:19:59.933Z"), "updatedAt" :
ISODate("2020-11-07T08:33:44.801Z"), "__v" : 0 }
```

or

5) **db.[collection_name].find().pretty()**

Result:

```
> db.themes.find().pretty()
{
      "_id" : ObjectId("5fa64a9f2183ce1728ff371a"),
      "subscribers" : [
            ObjectId("5fa64a072183ce1728ff3719"),
            ObjectId("5fa64ca72183ce1728ff3726"),
            ObjectId("5fa64c1f2183ce1728ff3723"),
            ObjectId("5fa64b972183ce1728ff3720"),
            ObjectId("5fa65bac2183ce1728ff372b")
      ],
```

Update a document in a collection:

```
db.themes.updateOne({themeName: 'Angular 10'}, {$set: {themeName:
'Angular 15'}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Delete a document in a collection:

```
db.themes.deleteOne({themeName: 'React 11'})
{ "acknowledged" : true, "deletedCount" : 1 }
```

Create new database:

```
> use testDB
switched to db testDB
```

```
> db.testCollection.insertOne({name: 'Test Name', age: 20});
{
      "acknowledged" : true,
      "insertedId" : ObjectId("62b16424084f5e13e694d4c6")
}
>
```

Basic Implementation of MongoDB in Node.js

1) npm i mongodb

2) index.js:

```
const { MongoClient } = require('mongodb');

.....................................................

const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

client.connect()
    .then(() => {
        console.log('DB Connected Successfully');
    })
    .catch( err => {
        console.log(err);
    });

const db = client.db('forum');
const themesCollection = db.collection('themes');

.....................................................

app.get('/themes', async (req, res) => {
   const themes = await themesCollection.find({themeName: 'Angular
10'}).toArray();

   res.render('themes', { themes });
});
```

3) views/themes.hbs:

```
<h1>Themes List</h1>
```

```
<ul>
    {{#each themes}}
        {{> theme}}
    {{/each}}
</ul>
```

4) views/partials/theme.hbs:

```
<li>Theme Name: {{themeName}}</li>
<li>Created At: {{created_at}}</li>
```

Mongoose ODM

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.

Mongoose Models - Models are fancy constructors compiled from Schema definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database

!!! **Model == Class == Table == Collection** !!!!

Basic Mongoose Example:

1) npm i mongoose

2) index.js:

```
const mongoose = require('mongoose');

const url = 'mongodb://localhost:27017/forum';
const themeController = require('./controllers/themeController');

mongoose.connect(url)
    .then(() => {
        console.log('Db is connected successfully!')
    })
    .catch( err => {
        console.log('DB error: ', err);
    });

app.use('/themes', themeController);
```

3) controllers/themeController.js

```
const router = require('express').Router();
const { Theme } = require('../models/Theme');

router.get('/', async (req, res) => {
```

```
    const themes = await Theme.find().lean();

    res.render('themes', {themes});
});

module.exports = router;
```

4) views/themes.hbs:

```
<h1>Themes List</h1>

<ul>
    {{#each themes}}
        {{> theme}}
    {{/each}}
</ul>
```

5) models/Theme.js

```
const mongoose = require('mongoose');

const themeSchema = new mongoose.Schema({
    themeName: String,
    created_at: Date,
    _id: Number
});

const Theme = mongoose.model('Theme', themeSchema); // the collection's
name is themes,
// mongoose converts Theme to themes when selecting the collection

exports.Theme = Theme;
```

Create Db Document Example:

0) The above code

1) createTheme.hbs:

```
<form action="/themes/create" method="post">
<div class="mb-3">
    <label for="themeName" class="form-label">Theme Name</label>
```

```
    <input type="text" class="form-control" id="themeName"
name="themeName" placeholder="Angular">
</div>
    <div class="mb-3">
        <label for="created_at" class="form-label">Created At</label>
        <input type="text" class="form-control" id="created_at"
name="created_at" value="Nov 07 2020 09:19:59 GMT+0200 (Eastern European
Standard Time)">
    </div>
    <div>
        <button class="btn btn-primary">Create Theme</button>
    </div>
</form>
```

2) controllers/themeController.js

```
router.get('/create', (req, res) => {
    res.render('createTheme');
});

router.post('/create', async (req, res) => {

     // First Way to Create DB Document / Row:

    // const theme = new Theme(req.body);

    /* theme.themeName = req.body.themeName;
    theme.createdAt = req.body.createdAt; */

    //  const themeSaveResult = await theme.save();

   //  console.log(themeSaveResult);

    // Second Way to Create DB Document:

    let savedTheme = await Theme.create(req.body);

    console.log(savedTheme);

    res.redirect('/themes');
});
```

We can set custom methods to be used on the database:

1) models/Theme.js

```
// First Way to set method on the collection
themeSchema.method('getInfoFirst', function () {
    return `First Method: This theme is ${this.themeName} and was
created at: ${this.created_at}`;
});

// Second Way to set method on the collection
themeSchema.methods.getInfoSecond = function () {
    return `Second Method: This theme is ${this.themeName} and was
created at: ${this.created_at}`;
}
```

2) controllers/themeController.js

```
router.get('/', async (req, res) => {

    const themesResult = await Theme.find();

    themesResult.forEach(theme => {
        console.log(theme.getInfoFirst())
    });

    const themes = await Theme.find().lean();

    res.render('themes', {themes});
});
```

3) Result:

```
First: This theme is Angular 10 and was created at: Sat Nov 07 2020
09:19:59 GMT+0200 (Eastern European Standard Time)
First: This theme is JS and was created at: Sat Nov 07 2020 09:23:27
GMT+0200 (Eastern European Standard Time)
First: This theme is React hooks JS - Pros and Cons and was created at:
Sat Nov 07 2020 09:25:04 GMT+0200 (Eastern European Standard Time)
```

Virtual Properties - won't be saved on the database, but they can be used on the collection; we can set getters and setters on the properties:

1) models/Theme.js:

```
themeSchema.virtual('isNew')
    .get(function() {
        return new Date(this.created_at).getTime() > new
Date('2021-09-24').getTime();
    });
```

2) controllers/themeController.js:

```
 themesResult.forEach(theme => {
     console.log(theme.isNew);
    });
```

!!! While we set the database schema we can set different constraints:

```
themeName: {
        type: String,
        required: [true, 'Theme name is required'],
        minLength: 2
    },
```

->
All SchemaTypes have the built-in required validator. The required validator uses the SchemaType's checkRequired() function to determine if the value satisfies the required validator.
Numbers have **min** and **max** validators.
Strings have **enum**, **match**, **minLength**, and **maxLength** validators.

!!!!

Get document by Id in the parameters:

1) controllers/themeController.js

```
router.get('/:themeId', async (req, res) => {
    // let theme = await Theme.findOne({_id:
req.params.themeId}).lean();
```

```
    let theme = await Theme.findById(req.params.themeId).lean();

    res.render('themeDetails', { theme });
});
```

2) views/themeDetails.js

```
<h1>{{theme.themeName}}</h1>
<h2>{{theme.created_at}}</h2>
```

CRUD Operations in Mongoose:

https://www.geeksforgeeks.org/node-js-crud-operations-using-mongoose-and-mongodb-atlas/

https://mongoosejs.com/docs/queries.html

Update a Document in Mongoose:

First Way:

```
Student

    .findById('57fb9fe1853ab747b0f692d1')

    .then((student) => {

      student.name = 'Stamat'

      student.save()

    });
```

Second Way:

```
Student

    .findByIdAndUpdate('57fb9fe1853ab747b0f692d1', {

      $set: { name: 'Petar' }
```

```
    }).then(students => console.log(students))
```

Third Way:

```
Student

    .updateOne(

      { name: 'Petar' },

      { $set: { name: 'Kiril' } }). then(students =>
console.log(students))
```

Relations in Mongoose:

1) One (cube):

```
const accessorySchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    imageUrl: {
        type: String,
        required: true,
        validate: {
           /* function() {
                return this.imageUrl.startsWith('http')
            }*/
            validator: /^http/g,
            message: 'ImageUrl should start with http'
        }
    },
    description: {
        type: String,
        required: true,
        maxLength: 120
    },
    cube: {
        type: mongoose.Types.ObjectId,
        ref: 'Cube'
    }
});
```

2) Many (accessories - array):

```
const cubeSchema = new mongoose.Schema({
     name: {
        type: String,
         required: true
     },
    description: {
        type: String,
        required: true,
        maxLength: 120
    },
    imageUrl: {
        type: String,
        required: true,

    },
    difficultyLevel: {
        type: Number,
        required: true,
        min: 1,
        max: 6
    },
    accessories: [
        {
            type: mongoose.Types.ObjectId,
            ref: 'Accessory'
        }
    ]
```

Add Document to another Document (add accessory to cube - one relation and add cube to accessory - another relation):

```
exports.attachAccessory = async (cubeId, accessoryId) => {

   const cube = await Cube.findById(cubeId);
   const accessory = await Accessory.findById(accessoryId);

   cube.accessories.push(accessory);
```

```
    accessory.cubes.push(cube);

    await cube.save();
    await accessory.save();

    return cube;

}
```

```
  _id: ObjectId('62b32c2b2c3b5a5721b2a2a3')
  name: "Eco-Dark"
  imageUrl: "https://thingsidesire.com/wp-content/uploads/2018/06/Eco-Dark-Rubik%E2…"
  difficultyLevel: 6
  description: "Eco-Dark"
  __v: 1
▾ accessories: Array
    0: ObjectId('62b342d7a64ef541722bee18')
```

Populate() - Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s). We may populate a single document, multiple documents, a plain object, multiple plain objects, or all objects returned from a query.

1) services/cubeService.js:

```
exports.getOne = (cubeId) =>
Cube.findById(cubeId).populate('accessories'); // along with cube
generation generate the cube's accessories
```

2) controllers/cubeController.js:

```
router.get('/:id', async (req, res) => {

    const cube = await cubeService.getOne(req.params.id).lean();

    res.render('details', { cube });
});
```

3) views/details.hbs:

```
 <main>
       <h1>{{cube.name}}</h1>
       <img class="cube" src={{cube.imageUrl}}>
```

```
        <div class="details">
            <p><span>Description:</span>{{cube.description}}</p>
            <p><span>Difficulty level:</span>
{{cube.difficultyLevel}}</p>
            <a class="btn"
href="/cube/{{cube._id}}/attach-accessory">Attach</a>
            <a class="btn" href="/">Back</a>
        </div>
        <h2>Accessories</h2>
        <div class="accessories">
            {{#each cube.accessories}}
            <div class="accessory">
                <img src={{imageUrl}} alt="stickerName">
                <h3>{{name}}</h3>
                <p>{{description}}</p>
            </div>
            {{else}}
            <h3 class="italic">This cube has no accessories yet...</h3>
            {{/each}}
        </div>
    </main>
```

!!! We can populate the nested documents:

```
exports.getOne = (cubeId) => Cube.findById(cubeId).populate({
    path: 'accessories',
    populate: {
        path: 'cubes',
        model: 'Cube'
    }
});
```

Filter documents with regex match

```
    let cubes = await Cube.find({name: {$regex: new RegExp(search,
'i')}}).lean();
```

Combining MongoDB + Mongoose:

```
  let cubes = await Cube.find({name: {$regex: new RegExp(search, 'i')}})
        .where('difficultyLevel').lte(to).gte(from)
        .lean();
```

Cookies and Sessions

HTTP is a stateless protocol. A stateless protocol does not require the server to retain information or status about each user for the duration of multiple requests. The

But some web applications may have to track the user's progress from page to page, for example when a web server is required to customize the content of a web page for a user. Solutions for these cases include:

1) the use of HTTP cookies.
2) server side sessions,
3) hidden variables (when the current page contains a form), and
4) URL-rewriting using URI-encoded parameters, e.g., /index.php?session_id=some_unique_session_code.

HTTP cookies, or internet cookies, are built specifically for Internet web browsers to track, personalize, and save information about each user's session. It is a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests. Typically, an HTTP cookie is used to tell if two requests come from the same browser—keeping a user logged in, for example. It remembers stateful information for the stateless HTTP protocol.

Session cookie - exists on the server, it can store information about a client and it is used to persist state across requests

!!!! Session is preferred when you need to store short-term information/values

Cookies are preferred when you need to store long-term information/values

Session is safer because it is stored on the server and it is short-term. !!!!

Basic Example of Setting Cookies in Express:

1) npm i cookie-parser

2) index.js:

```
const express = require('express');
const cookieParser = require('cookie-parser');
```

```
const app = express();

app.use(cookieParser());

app.get('/', (req,res) => {
    res.cookie('test_key','test_value'); // this is equal to:
    // res.header('Set-Cookie', 'test_key=test_value');
    res.send('Hello World');
});

app.get('/cats', (req, res) => {
    let cookies = req.cookies;

    console.log(cookies);

    res.send('I love cats!');
});


app.listen(5000, () => console.log('Server is listening on port
5000...'));
```

Basic Example of Setting Session in Express:

1) npm i express-session (without cookie-parser)

2) index.js:

```
const express = require('express');
const expressSession = require('express-session');

const app = express();

app.use(expressSession({
    secret: 'keyboard cat',
    resave: false,
    saveUninitialized: true,
    cookie: { secure: false }
}));


app.get('/', (req,res) => {

    req.session.username = "Pesho" + Math.random();
    res.send('Home Page');
```

```
});

app.get('/cats', (req, res) => {

    console.log(req.session.username);

    res.send('I love cats!');

});


app.listen(5000, () => console.log('Server is listening on port
5000...'));
```

3) Result:

Pesho0.8450060279403158 in one browser

Pesho0.39509020291715213 in another browser

Authentication vs Authorisation

Authentication (AuthN) is a process that verifies that someone or something is who they say they are.

vs

Authorization is the security process that determines a user or service's level of access. In technology, we use authorization to give users or services permission to access some data or perform a particular action.

Authentication - It's built on several layers of abstraction

Cookies -> Sessions -> Security

Authentication is a cross-cutting concern, best handled away from business logic

Request -> Authentication -> Business Logic -> Response

Password Hashing

!!!! Hashing a password means taking a plaintext password and transforming it into a unique and concise string that represents the password in a way that does not reveal any information about the password itself.

In cryptography, a salt is random data (generated each time during hashing) that is used as an additional input to a one-way function that hashes data, a password or passphrase. Salts are used to safeguard passwords in storage. They add dynamic encryption to the static hashed password. !!!!!

Basic use of password hash and salt:

1) npm i bcrypt

2) index.js:

```
const express = require('express');
const cookieParser = require('cookie-parser');
const bcrypt = require('bcrypt');
```

```
const app = express();
const hashedPassword =
'$2b$10$/m98uTorMbpj7RQuvv.VHeBB8QMcnALTPpccYeFkhvDsvrXwrSmU2'; //==
secretpassword
const saltRounds = 10;


app.use(cookieParser());


app.get('/hash/:password?', async (req,res) => {

    const salt = await bcrypt.genSalt(saltRounds);

    const hash = await bcrypt.hash(req.params.password, salt);

    res.send(`Password: ${req.params.password} + salt: ${salt} + hash:
${hash}`);

});

app.get('/login/:password', async (req, res) => {

    const isValidPassword = await bcrypt.compare(req.params.password,
hashedPassword);

    if (isValidPassword) {
        res.send('Successful login');
    } else {
        res.send('Invalid password')
    }
});


app.listen(5000, () => console.log('Server is listening on port
5000...'));
```

JSON Web Token

A JSON web token (JWT) is a URL-safe method of transferring claims between two parties. The JWT encodes the claims in JavaScript object notation and optionally provides space for a signature or full encryption.

1) Flexibility and ease of use: JWTs are easy to use. Their self-containing nature helps you achieve what you need for verification without database lookups. This makes JWTs more suitable to use in an API, since the API server doesn't need to keep track of user sessions.

2) Cross-platform capabilities: Because of their stateless nature, tokens can be seamlessly implemented on mobile platforms and internet of things (IoT) applications, especially in comparison to cookies..

3) Multiple storage options: Tokens can be stored in a number of ways in browsers or front-end applications.

4) It is digitally signed: As JWT are digitally signed by the issuer, they can be used for authentication purposes by validating the signature, without having to expose a password to Db, verifying that the password has not been intercepted

Structure:

header.payload.signature -

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IlBlc2hvIiwiaWF0IjoxNjU2MDAzMjIzLCJleHAiOjE2NTYwODk2MjN9.3an3B8ZQ60SO3CYDlMn_9bq05Cu1JISi6njBOWxg4Cc

Basic JWT Example:

1) npm i jsonwebtoken

2) index.js

```
const express = require('express');
const cookieParser = require('cookie-parser');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
```

```
const app = express();
const hashedPassword =
'$2b$10$/m98uTorMbpj7RQuvv.VHeBB8QMcnALTPpccYeFkhvDsvrXwrSmU2'; //==
secretpassword
const saltRounds = 10;
const secret = 'Mysupersecret';


app.use(cookieParser());


app.get('/hash/:password?', async (req,res) => {

    const salt = await bcrypt.genSalt(saltRounds);

    const hash = await bcrypt.hash(req.params.password, salt);

    res.send(`Password: ${req.params.password} + salt: ${salt} + hash:
${hash}`);

});

app.get('/login/:password', async (req, res) => {

    const isValidPassword = await bcrypt.compare(req.params.password,
hashedPassword);

    if (isValidPassword) {
        const payload = {
            username: 'Pesho'
        }

        const options = {
            expiresIn: '1d'
        }


        const token = jwt.sign(payload, secret, options);

        res.send(`Successful login: token: ${token}`);

    } else {
        res.send('Invalid password')
    }
```

```
});

app.get('/verify/:token', (req,res) => {


        jwt.verify(req.params.token, secret, (err, decodedToken) => {

            if (err) {
                return res.status(401).send('You do not have
permissions!');
            }
            res.json(decodedToken);

        });

});


app.listen(5000, () => console.log('Server is listening on port
5000...'));
```

!!!! **Three Layer Architecture** -

Controller - > Service - > Data Layer:

http request comes to controller, the controller then tells the service to fetch some information, which the service requires from the database models / data layer.

the data layer gives the data to the service, the service handles the information, then returns the data to the controller, which makes a http request back

If we have several controllers, services and models, we must not use a service in another service if we want to get data from a model / database. Instead we need to ask the controller, where the service is required, for the information and then the controller should pass the information to the other service. Otherwise, we cause circular dependency !!!!

How to set only logged-in users to access certain page:

First, upon login a login token is attached to the cookies on the user's cookies.Then an auth middleware checks if we have a login token and verifies that the login token is the one we issued during login. If it is, it attaches the login token to the requests going forward. Finally, via isAuth middleware we redirect each request without a login token to a custom error page.

1) index.js:

```
const { auth } = require('./middlewares/authMiddleware');

app.use(auth);
app.use(routes);
```

2) middlewares/authMiddleware.js:

```
const jwt = require('jsonwebtoken');
const { sessionName, secret } = require('../config/constants');
const { promisify } = require('util');

exports.auth = async (req, res, next) => {

    let token = req.cookies[sessionName];

    const jwtVerify = promisify(jwt.verify);

    if (token) {

        try {

            let decodedToken = await jwtVerify(token, secret);

            req.user = decodedToken;

        } catch  {
            return res.redirect('404');
        }

    }

    next();

};
```

```
exports.isAuth = (req, res, next) => {

    if (!req.user) {
        res.redirect('/404');
    }

    next();

};
```

3) controllers/authController.js:

```
router.post('/login', async (req,res) => {

    let token = await authService.login(req.body);

    if (token) {
        res.cookie(sessionName, token);
        res.redirect('/');
    } else {
        res.redirect('404');
    }

});
```

4) services/authService.js:

```
const User = require('../models/User');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const {secret, saltRounds} = require('../config/constants');


exports.login = async ({username, password}) => {

    let user = await User.findOne({username});

    if (!user) {
        return false;
    }

    const isValid = await bcrypt.compare(password, user.password);

    if (!isValid) {
        return false;
```

```
    }

    let result = new Promise((resolve, reject) => {
        jwt.sign({_id: user._id, username: user.username}, secret, {
expiresIn: '2d'}, (err,decodedToken) => {

            if(err) {
                reject(err);
            }

            return resolve(decodedToken);

        });
    });

    return result;


};
```

5) config/constants.js:

```
exports.saltRounds = 10;
exports.secret = 'Mysecretpassword';
exports.sessionName = 'session';
```

6) controllers/cubeController.js - where we set the auth protection on the create page:

```
const { isAuth } = require('../middlewares/authMiddleware');

router.get('/create', isAuth, (req, res) => {
    res.render('create');
});
```

Creating an owner of a cube / product and only they may edit it (continuation of the above code):

1) models/Cube.js:

Schema ->

```
owner: {
         type: mongoose.Types.ObjectId,
        ref: 'User'
    }
```

2) controller/cubeController.js:

```
router.post('/create', isAuth, (req, res) => {
    const cube = req.body;
    cube.owner = req.user._id;
```

```
router.get('/:cubeId/edit', isAuth, async (req, res) => {

    const cube = await cubeService.getOne(req.params.cubeId).lean();

if (cube.owner != req.user._id) {
        return res.redirect('/404');
    }
```

Passing data to views per-user - displaying the user's username below the nav menu:

1) middlewares/authMiddlewares.js - res.locals can save information on the response, which can then be used by the views:

```
let decodedToken = await jwtVerify(token, secret);

            req.user = decodedToken;
            res.locals.user = decodedToken;
```

2) views/layouts/main.hbs:

```
 <ul>
            <li><img class="logo" src="/images/logo.png"></li>
```

```
            <li><a href="/">Browse</a></li>
            <li><a href="/cube/create">Add a Cube</a></li>
            <li><a href="/accessory/create">Add Accessory</a></li>
            <li><a href="/about">About</a></li>
            <li><a href="/auth/login">Login</a></li>
            <li><a href="/auth/register">Register</a></li>
            <li><a href="/logout">Logout</a></li>
        </ul>
        <h2>{{user.username}}</h2>
```

Error Handling

!!!! All async actions must be placed in try - catch clause and their errors must be handled !!!

Basic Error Handler Example:

1) middlewares/errorHandlerMiddleware.js:

```
const { errorMapper } = require('../utils/errorMapper');

exports.errorHandler = (err, req, res, next) => {

    const status = err.status || 404;

    res.status(status).render('404', {error: errorMapper(err)});

};
```

2) utils/errorMapper.js:

```
exports.errorMapper = (err) => {

    let errorMessage = err.message;

    if (err.errors) {
       errorMessage = Object.values(err.errors)[0].message;
    }

    return errorMessage;

};
```

3) index.js:

```
const {errorHandler} = require('./middlewares/errorHandlerMiddleware');

app.use(auth);
app.use(routes);
app.use(errorHandler);
```

4) controller/authController.js:

```
 try {
        const user = await authService.login(username, password);
        const token = await authService.createToken(user);

        res.cookie(SESSION_NAME, token, {httpOnly: true});

        res.redirect('/');
    } catch (error) {
        res.render('auth/login', {error: errorMapper(error)});
    }
```

5) views/layouts/main.hbs:

```
<body>

    {{#if error}}
    <div class="error-box">
        <p>{{error}}</p>
    </div>
    {{/if}}
```

6) css:

```
.error-box {
        position: fixed;
        right: 1.6em;
        display: block;
        z-index: 11111;
        top: 15%;
        text-align: center;
        border-radius: 1rem;
        color: #c71414;
        font-style: italic;
        height: auto;
        width: 420px;
        box-shadow: 0px 0px 8px 7px #c71414;
    }

    .error-box>p {
        font-size: 25px;
        font-weight: bold;
        font-family: 'Barlow', sans-serif;
        padding: 22px;
```

```
        height: auto;
    }

    .error-box {
        -moz-animation: cssAnimation 0s ease-in 10s forwards;
        /* Firefox */
        -webkit-animation: cssAnimation 0s ease-in 10s forwards;
        /* Safari and Chrome */
        -o-animation: cssAnimation 0s ease-in 10s forwards;
        /* Opera */
        animation: cssAnimation 0s ease-in 10s forwards;
        -webkit-animation-fill-mode: forwards;
        animation-fill-mode: forwards;
    }

    @keyframes cssAnimation {
        to {
            width: 0;
            height: 0;
            overflow: hidden;
            padding: 0;
        }
    }

    @-webkit-keyframes cssAnimation {
        to {
            width: 0;
            height: 0;
            visibility: hidden;
            padding: 0;
        }
    }
```