**Query String in URL**

A screenshot of a computer

Description automatically generated with medium confidence

As we can see, after the “?”, we have 2 query strings, one is address=Cairo and temp=2

**Fetching API**

First, in order to get a weather, forecast from a 3rd party website and display it on the localhost, we need to first fetch this API in the backend, and process it to extract the data we need in some variable, then we link this variable such that when we call an endpoint of our localhost, we receive this data immediately.

**Why do we need to fetch the data in the backend, while we can directly fetch it in the front end directly using the 3rd party web api link?**

There are some conditions which require or prefer API fetching to be in the back-end:

1. When you need to use **security credentials(access token masalan)** to access the API and you those credentials need to stay private.
2. When the API does not permit **cross origin access** so your front-end could not directly access it.
3. When you need to do some processing on the API results and you wish to **keep the processing algorithms a trade secret** (keep in mind that all processing done in the front-end is available for any coder to see). There are no secrets in the front-end code.
4. When the front-end device may not have **appropriate resources** (CPU, memory, etc..) to process the API results. This can particularly be the case with smaller dedicated devices.
5. When you don't want the front-end device to have to **expend the battery energy** to regularly access the API. This might particularly be the case if you're regularly polling some API looking for changes. In that case, you might want the back-end to do the polling and just let the front-end know when there is something to actually do.
6. When there are reasons to do some **centralized performance management** of your access to the API such as caching of results or request throttling.
7. When all front-end devices may not have **proper network access to the API** server.

If there are no compelling benefits to putting the API access in the back-end, then it may be more scalable (for your server) to let the front-end do as much work as it can.

Steps:

1. Go to the website manual and find the api that returns us the weather info ex:

[www.weather.com/accesstoken?city=cairo](http://www.weather.com/accesstoken?city=cairo)

This returns us weather for cairo

Text

Description automatically generatedWe do this using the *request* library

**the request function takes 2 parameters like this**

request({url:theAPIURL, json:true} , (error,response)=>{

if(error)

console.log(error);

else{

let data=response.body; //response.body includes the data that we get //from the API

}

});

**How to deploy to heroku?**

1. Enter in terminal of root folder of project

heroku key:add

Then in order to load express

const express = require('express');



An example of using it

Text

Description automatically generated

**REST API**

Graphical user interface, application, Teams

Description automatically generated

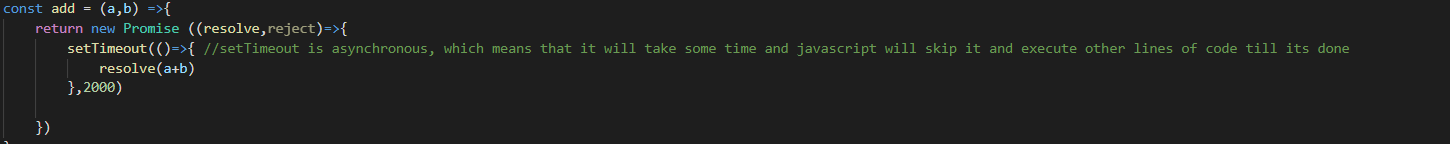
Graphical user interface, application, Teams

Description automatically generated

**Promises chaining feature:**

* Avoids having nested asynchronous functions if we have more than 1 asynchronous function

Assume we have the function add that we use to perform asynchronous operation (we learnt how to create a function with promise before)



Now to call this function and use more than 1 asynchronous method at the same time, we know one way before and it is to nest asynchronous functions as shown below.

Text

Description automatically generated

Now ,with promise chaining, we get the same result but more efficiently as shown below

Text

Description automatically generated

Result in both :

Text

Description automatically generated with medium confidence

Disadvantages of promise changing

1. It may be hard to understand what is going on ( a lot of different functions and return values)
2. Difficult to have all the values within the same scope

**As a result we have the await- async, which improves this**

**Async Await**

**Async:** returns a promise implicitly for example, if we returned “Chris” from an async function, Promise { “Chris”) is what gets returned, which as a result means that we can then use this value in the callback of the .then (value)

**Await:** Inside the async function, we add await to an asynchronous function that we call inside our async function, and it basically means that I will wait for this called asynchronous function to finish executing and pause here, then once its done, I’ll go to the next line.

An example of its usage is shown below:

Text

Description automatically generated

Output



**Pattern Chaining VS Async-Await example**

**The following is an example of a functionality that we can do by promise chaining Or Async-Await**

1. Using Promise Chaining

Text

Description automatically generated

1. Using Async

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

**Encryption and Hashing**

*Difference between hashing and encryption?*

* Using encryption, password can be reversed back but using hashing, this is one way only

We hash the password before logging in, then after logging in, we compare the hashed password with the password stored in the database. This is the most secure way.

A screenshot of a computer

Description automatically generated with medium confidence**Below is a simple program demonstration how bcrypt library works**

*Output:*

Graphical user interface, text

Description automatically generated

**Middlewares registers function to occur before or after a given event**

Updates= [“name”,”email”, “password”]

MainObject={

name: ‘Chris’,

email: ‘asdasx@aucegypt.edu’,

password : ‘123456’

}

NewObjectUpdated= {

name: ‘Chris Amgad’,

email: ‘chrisamgad@aucegypt.edu’,

password : ‘kokokaka’

}

Updates.forEach((update)=>{

console.log(update)//each update inside the Updates array individually

MainObject [update]= NewObjectUpdated[update]//update our main object values

//with the new values of the NewObjectUpdated

})

**Difference between statics and methods in mongoose**

Graphical user interface, text, application

Description automatically generated

**Steps for Accepting Tokens and Authentication**

Note: The token is an object that includes ID and timestamp. This helps the user after logging in and the token is stored for a specific time interval, for this time interval the user identity is confirmed and he will be able to perform what they need. The goal of JWT isn't to hide data, but to prove your identity to the server. Anyone can decode the token, but they can't create fake tokens because that requires the **secret key** (in our case in the example will be ‘thisismynewcourse’). The server will throw an exception when attempting to decode a fake token, since no one knows your private key.

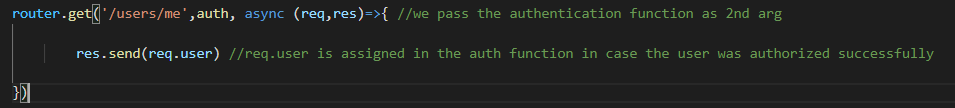
1. If the user would want to Log In or Signing Up, We would generate a token for this user.
   1. In case of Logging in
      1. We look for the user in our DB using their email and password using findOne
      2. Then we generate a token for **this** user and then update **this** user’s tokens array of objects with the newly generated token
      3. Text

         Description automatically generatedSave the user to DB
   2. In case a user tries to access an API that requires Authentication (every API except logging in and signing up)

Authenticate by accessing the request header when the user is trying do go to some API that requires authentication(every API except the logging in and signing up) and the following is done in the function:

* + - 1. Access the header request that includes the token key in the form “Bearer <Token>” so we need to remove the Bearer string.
      2. Decode the token using the jwt verify function that takes in the token and the secret key that previously encoded the token with
      3. Search within all the users in the database for the user that has the id of the token and their token inside the tokens array, and if the token was found for a specific user, this user is authenticated successfully, and they gain access to the API that they requested to access; otherwise, print on the screen an error that they need to log in again (in order to generate a new token and have it stored in the tokens array of this user)

The auth function gets passed as 2nd argument to the API in express API route handler and it always gets executed as a way to validate authentication before handling the route:



Text

Description automatically generatedauth function:

**Simple Program for running jsonwebtoken**

A picture containing text, monitor, screen, screenshot

Description automatically generated

Token-based authentication is a protocol which allows users to verify their identity, and in return receive a unique access token. During the life of the token, users then access the website or app that the token has been issued for, rather than having to re-enter credentials each time they go back to the same webpage, app, or any resource protected with that same token.

* Every operation will require authentication except login and signing up(create new user). So logging in and signing up wont require authentication
* Having a tokens array is good because for each device, a token will be generated then the user can login in on multiple devices; thus, having different tokens generated and stored in the tokens array. Then simply, once a user wants to log out from a certain device this token gets deleted

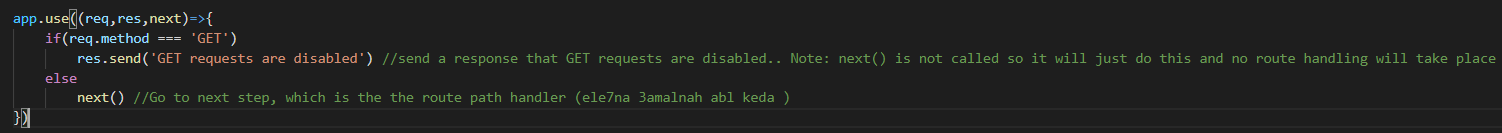
*Express middleware*

Text, timeline

Description automatically generated with medium confidence

**Example of a simple Express middleware**

Note: the middleware should be above all the app.use() functions

****

This is how we add a middleware to a specific route

Text

Description automatically generatedTimeline

Description automatically generated

Here, auth is the middleware function

**Logging in:** Adds a new token to the array tokens (array tokens represent the currently logged in sessions)

**Logging Out:** Removes the currently logged in Token from the arrays token so in auth.js, the token will no longer be found in the tokens array during the checking and so authentication would fail in case a user tried accessing an API that required running the middleware auth.js

**Assume we have a model Player**

**Below is an example of the one of the player Lional Messi**

Text

Description automatically generated

Ex:1 How do you search for an object in an array of many players using mongoose?

For example, find the player that has the award of Ballon do’r?

const BallonDorPlayer= mongoose.findOne({‘awards.award’: ‘Ballon d’Or’})

Ex2: Find the player who has received a golden boot 6 times

const player= mongoose.findOne({‘awards.award’: ‘Golden Boot’, ‘numberOfTimes’:6})

**Ex3:** Find the user that wants to log out(find the user that has the current authenticated token from the list of ALL tokens of all users)

//assume we have access to current token in req.token

const token= mongoose.findOne({‘tokens.token’: req.token })’

**Population**

Let’s say you have a Message Model, where the model has a key called *sender\_id,* and this *sender\_id* is referring to another Model, which is a User Model. Shown below is the Message Model

const messageSchema=({

key1:{7agat},

key2:{7agat},

sender\_id:{

type: mongoose.Schema.Types.ObjectId,

**ref:’User’ //Referring to User Model**

}

})

const message = mongoose.model(‘Message’, messageSchema);

Now, if we tried calling message.populate(‘sender\_id), what basically will happen is that the message will go looking for sender\_id field in message model and look for ref value, then goes to this model ref value and brings back all its data.

**Exploring how population works**

Text

Description automatically generated

**How do we link both the tasks and the users collections?**

* Since each user will have their own tasks, so every user must only have privileges on their own tasks, and no user have access to others tasks
* We can do this via 2 methods

1. In the user model, we can add a new field of type array tasks\_ids , where each index contains a taskID of type mongoose.Schema.Types.objectID, and this array basically include all the tasks ids that belong to that user
2. In the task model, we can add a new field to the task model, which is of type mongoose.Schema.Types.objectID (Id) and this refers to the Id of the user/owner of this task.

**We are going to choose the 2nd method as done in the video**

There is a better way instead of getting the owner id then searching for the ID of the owner of the task in the list of owners/users. This is more code.

The more efficient way is provided my mongoose.

1. We have a *ref*  property that we add in the task model, and we specify that we are referring to *User* model, and this reference is referenced in the owner field

Text

Description automatically generated

The ref tells Mongoose “Those docs are going to be in the \_\_\_ collection.”

1. As the user is authenticated, he creates a new task and below is the image of the new task created, when the create task API is called, Note that the owner ID was stored here in the owner as we added this functionality to store the user authenticated ID in this field

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

1. As we did in step 1, we referred to the User model from the Tasks model. We have a function called populate(), that basically tells Mongoose “Hey, I’m gonna be referencing other documents from other collections”.

So basically in our case, We can use populate to retrieve the user’s document that has the ID passed in the parameters of *populate*, an basically task has the owner field updated to include the document of this user

Text

Description automatically generated

1. The output: We get the user that is associated to this task and we can find the result in the owner field Text

   Description automatically generated

***Making Use of Query String to perform some functionalities***

Diagram

Description automatically generated

1. **Filtering using Query String**

* We are going to make use of **req.query**, which was explained before

Note: We can achieve the same thing using mongoose function find to retrieve items with specific properties but this is another way to filter using the query string

Text

Description automatically generated

As we see above, we can make use of the **match** property in the options of populate function to do this.

**2. Pagination using Query String**

Using the keywords **limit** and **skip**

**Ex:**

if limit=10 and skip=0 🡪 we get the first 10 records

if limit =10 and skip=10 🡪we skip the first 10 records and get the next 10 records

and so on…

We can perform this by adding **options** field in populate, and options include the properties **limit** and **skip**, where we assign these by what we want. For example,

Text

Description automatically generated**Output: we get only 1 record and since skip is 0, we get the 1st record only**

A picture containing text

Description automatically generated   
Graphical user interface, application

Description automatically generated with medium confidence

**3. Sorting using Query String**

We use the query parameter name **sortBy** to sort, and we need provide 2 pieces of info to it

1. *What are you sorting?*

Examples:

🡪 sort createdAt

🡪 sort updatedAt

🡪 sort age

1. Ascending or Descending?

Examples for how to use in URL

***Ex 1:***

/tasks?sortBy=createdAt:asc

OR 🡪 **we sort createdAt by ascending order**

/tasks?sortBy=createdAt\_asc

***Ex 2:***

/tasks?sortBy=age\_desc 🡪 **we sort age by descending order**

How to use in code?

Below is an example of tasks being sorted based on **createdAt** in **descending order**

We have a new key **sort** in the options, which is the 3rd arg in the populate function.

* 1 refers to sorting in ascending order
* -1 refers to sorting in descending order

Text

Description automatically generated

Text

Description automatically generated**Example in video:**

Text

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

**File Uploads**

Express doesn’t support file uploads but there’s a 3rd party library that we can install called **multer**

Install by **npm i multer**

*Configuring multer before using it in a router*

Text

Description automatically generated with medium confidence

**Uploading Image**

*Method 1: using storing binary of image*

A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

Graphical user interface, text

Description automatically generated*Method 2: having API that returns back image from URL*

Result:

Graphical user interface, application, PowerPoint

Description automatically generatedA picture containing text, city

Description automatically generated

**Auto Cropping and Image Formating using *sharp* module**

Install sharp library by npm i sharp

A screenshot of a computer

Description automatically generated with medium confidence

A picture containing text, monitor, screen, screenshot

Description automatically generated

**Sending Emails**

Using Enviorument variables is good for 2 reasons:

1. Customization: for example when we are deploying to Heroku, we specify an environment variable for the port in case we’re deploying there and the localhost if we’re deploying on local host
2. Security

First install env-cmd module by

*npm i env-cmd –save-dev*

Text

Description automatically generated

Our **dev.env** looks like this

A screenshot of a computer

Description automatically generated with medium confidence

An example of accessing an environment variable:

Text

Description automatically generated with medium confidence

Deploy

**Testing**

We will be using JEST. We are going to install it in our developers dependencies (because we don’t need Heroku to install it when deploying)

Install by ***npm i jest –save-dev***

Text

Description automatically generated

Also install supertest to test HTTP routes ***npm i supertest –save-dev***

**Chat Application using Socket.io**

First, we need to setup socket.io

1. Install socket.io using *npm install socket.io*
2. We initialize the public directory as our reference directory that includes javascript, html,css files and etc.

A screenshot of a computer

Description automatically generated with medium confidence

Note: \_\_dirname refers to where are we currently at now in this file (index.js)

Graphical user interface, text

Description automatically generated

As we can see, index.js is in the src directory inside chat-app directory so

\_\_dirname is 🡪 C:\Users\chris\Desktop\Node-Course\chat-app\src

so publicDirectoryPath variable as shown above will be

C:\Users\chris\Desktop\Node-Course\chat-app\src\..\public

* Which is pointing at the public directory

1. Now, we need to setup server file and clients

Text

Description automatically generatedThe server side:

Above is a simple code that sends to the client a count variable with an event name. This event name should be at the client as well, and when the server emits(sends) the client this event name, the client looks up this event name and execute some function

Also, the above code illustrates io.on function, which is basically waiting for any client to send to the server the event name “increment”, and if detected executes the call back function that increments the count (count++), then notifies back all users with event “countUpdated” along with the updated count variable

(Note: io.emit notifies all clients, while socket.emit notifies one client

1. Set up a client example

A picture containing text, monitor, screenshot, screen

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

**Socket.io acknowledgment**

Basic example I did

*Client Side*

A screenshot of a computer

Description automatically generated with medium confidence

*Server Side*

A screenshot of a computer

Description automatically generated with medium confidence

An example from Socket.io documentation

Text

Description automatically generated

Functions throughout the course

**In server**

1. socket.emit 🡪 sends an event to a specific client
2. io.emit 🡪 sends an event to every connected client
3. socket.on 🡪 waits for an event to be emitted from the client and once the client emits to the server an event, a function gets triggered (we write this function)
4. socket.broadcast.emit 🡪 sends an event to every connected client except this one

//for example, we want all users to have receive the message "A new user has joined" once I join/connect to the room, but I don’t get this message.

1. socket.joins
2. io.to(roomname).emit 🡪 sends an event to everybody in a specific room without sending it to other room
3. socket.broadcast.to(roomname).emit 🡪 sends an event to everyone except a specific client but its limiting to a specific chat room

**Middleware**

Note: EVERY middleware function must have the next

🡪What is the difference between next() and next(‘route’)?

Graphical user interface, text, application

Description automatically generated

Some javascript function callbacks

Text

Description automatically generated

Using usestate inside UseEffect

<https://stackoverflow.com/questions/57521132/working-of-usestate-function-inside-useeffect-function>

* look at first comment, and expand comments Graphical user interface, text, application

  Description automatically generated

Output:

*False*

*False*

*True*

*True*

Why?

Graphical user interface, text

Description automatically generated