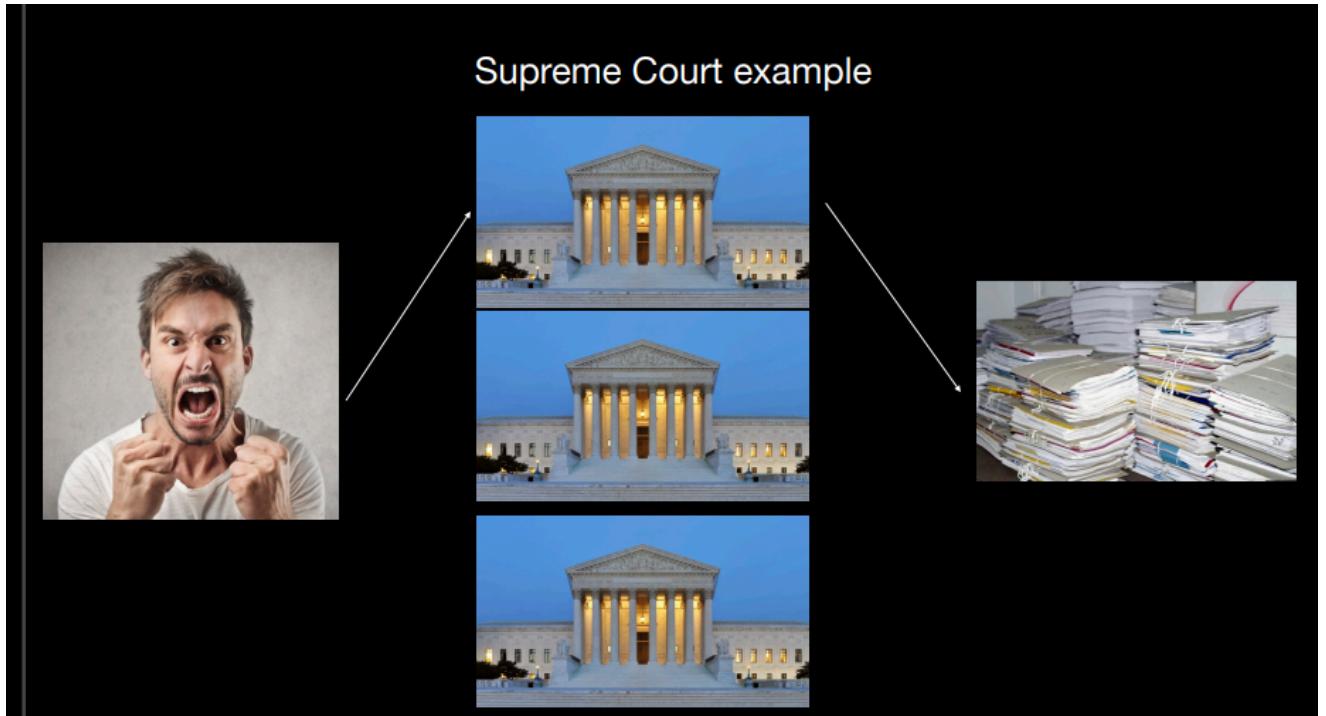


# MongoDB deep dive

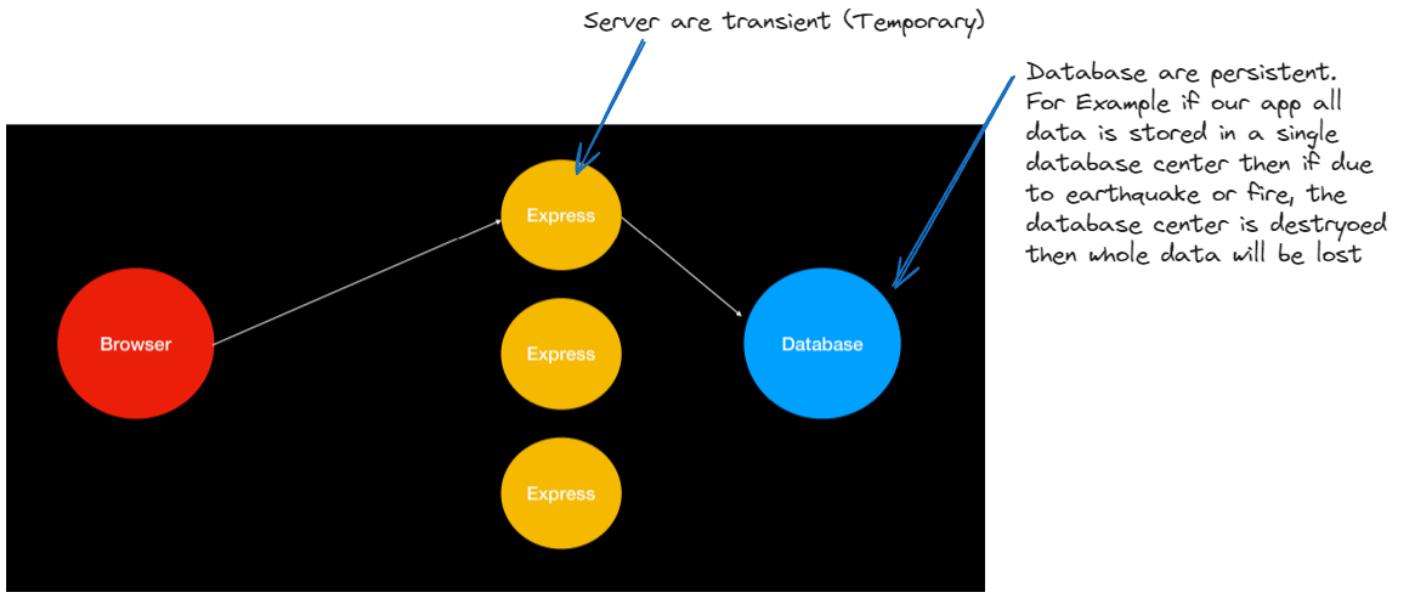
Understanding CRUD, mongoose and building an end to end authenticated app

Lets see and example:



## What is database?

It is place where data is stored persistently



Servers autoscale.

Lets see an example of LinkedIn \

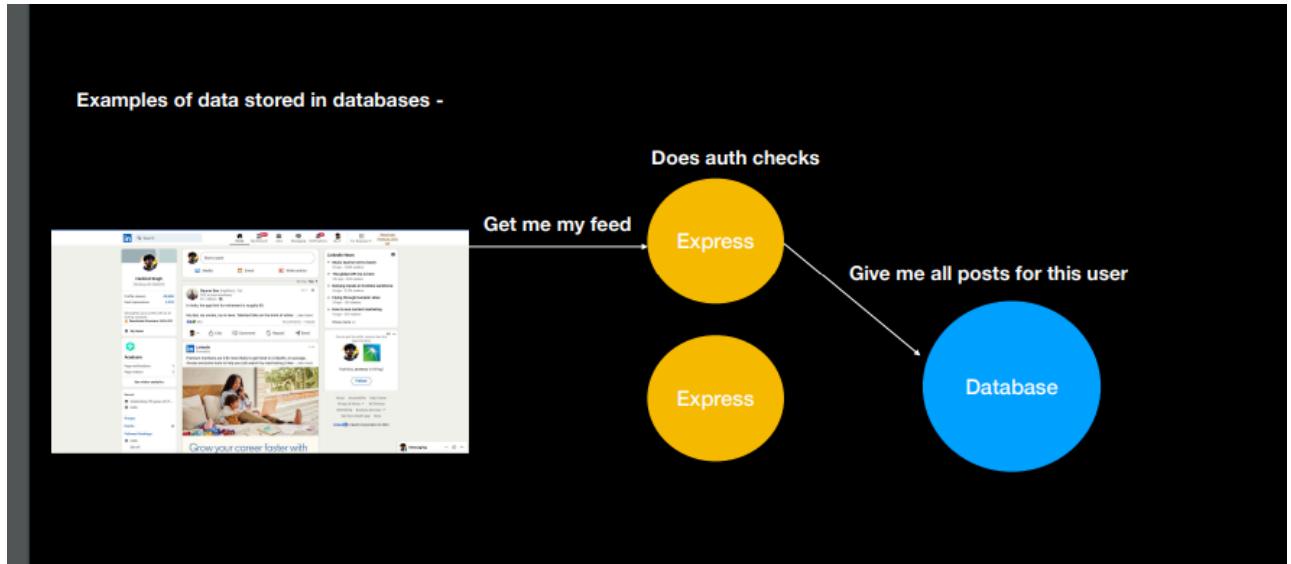
**What is a database**

Examples of data stored in databases -

**For LinkedIn**

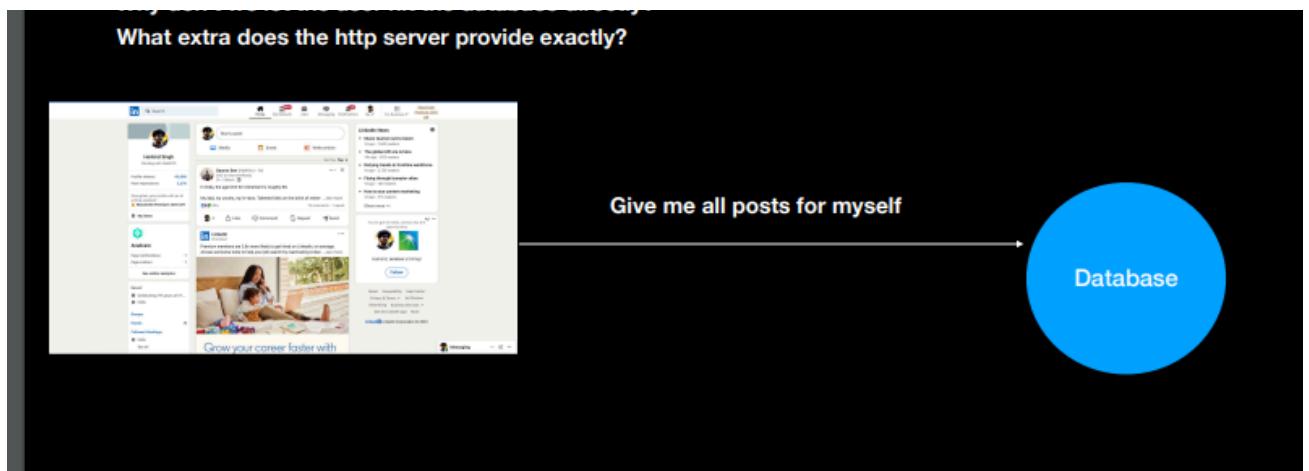
1. User data
2. Users posts
3. Users connection relationships
4. Messages

How does the request usually go by:



**Why don't we let the user hit the database directly?**

**What extra does the http server provide exactly?**



1. Database were created using protocols that browsers don't understand
2. **Databases don't have granular** (database give access to everything or nothing, Browser can ask for any data , Express server has access to all the database , hence it need to restrict the access of database (by

browser) auth check ) **access as a first-class citizen. Very hard to do user-specific access in them**

3. Some databases (firebase) let you get rid of the HTTP server and try their best to provide granola access. (hence its not compulsory to always have an HTTP Server)

Databases usually allow access to 4 primitives

1. Create Data
2. Read Data
3. Update Data
4. Delete Data

Popularly known as **CRUD**

**ODM** : Library to talk to database : examples mongoose and Prisma

Let's see the API for the mongoose library

Eventually, we'll be using prisma (which is the industry standard way of doing this)

In mongoose, first you have to define the schema

This sound counter intuitive since mongodb is schemaless?

That is true, but mongoose makes you define schema for thing like autocompletions/ Validating data before it goes in DB to make sure you're doing things right Schemaless DBs can be very dangerous, using schemas in mongo makes it slightly less Dangerous.

mongoose → Define four schema



Some database will ask upfront how the data(user table) will look like these are called SQL (Structured Query Language)

mongoDB says send me anything you want . We can store two document of different schema completely in database But SQL doesn't allow

mongoose library says first tell me what is your schema , one we tell them then we can insert all the data  
So basically if we add the data from our express server then we can follow schema strictly which we send to mongoose library  
But we still can insert random data from GUI of the MongoDB library.

Generally in our application we need strict schema and mongoose make our application more strict

Similar to TS and JS . TS provide us type safety which give us long term benefit.

## Defining Schema:

```
const UserSchema = new mongoose.Schema({
  email:String,
  password:String,
  purchasedCourses: [
    {
      type: mongoose.Schema.Types.ObjectId,
      // ObjectId : mongoDB way to give randomised ID
      ref:"Course"
    }
  ]
})
const CourseSchema = new mongoose.Schema({
  title : String,
  price : 5999
})
```

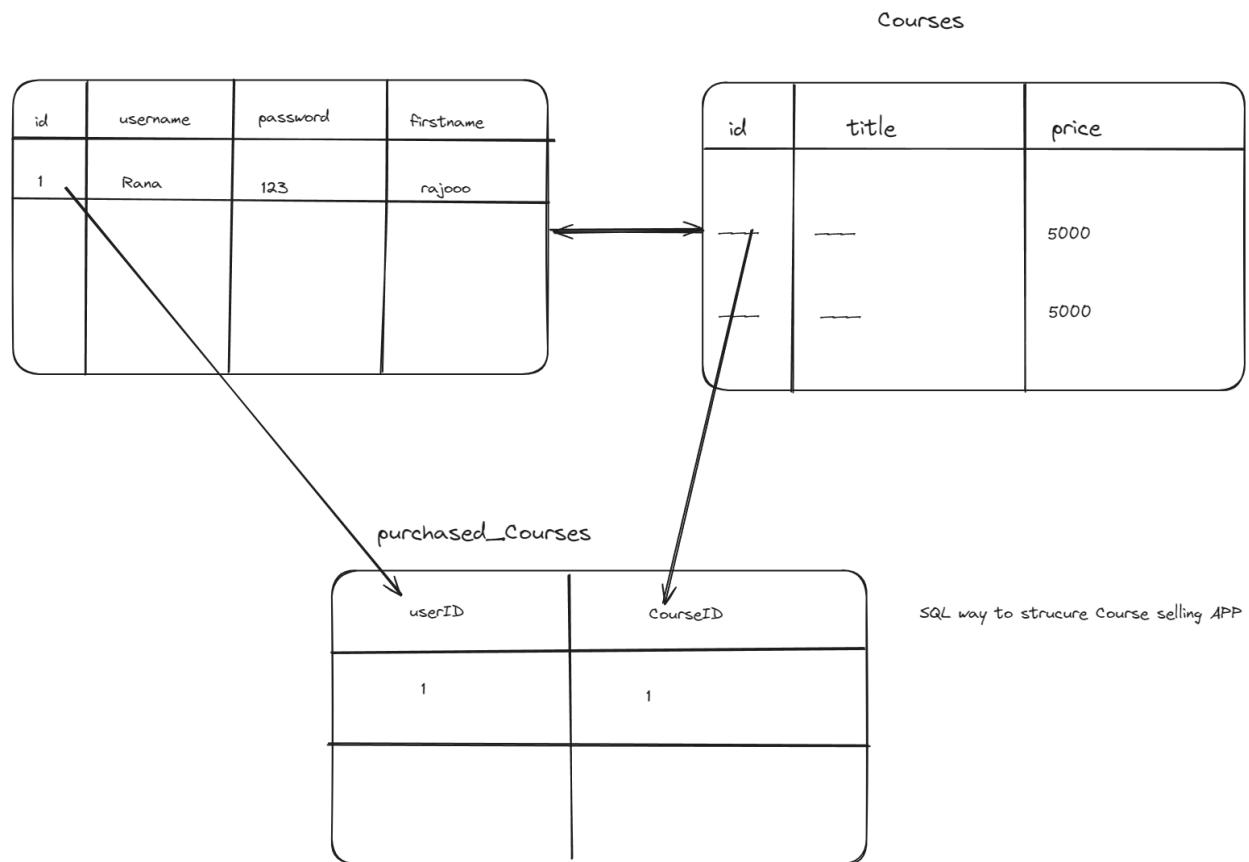
```
const User = mongoose.model('User',UserSchema);
const Course = mongoose.model('Course',CourseSchema);
```

MongoDB allow us to create complex objects For example:

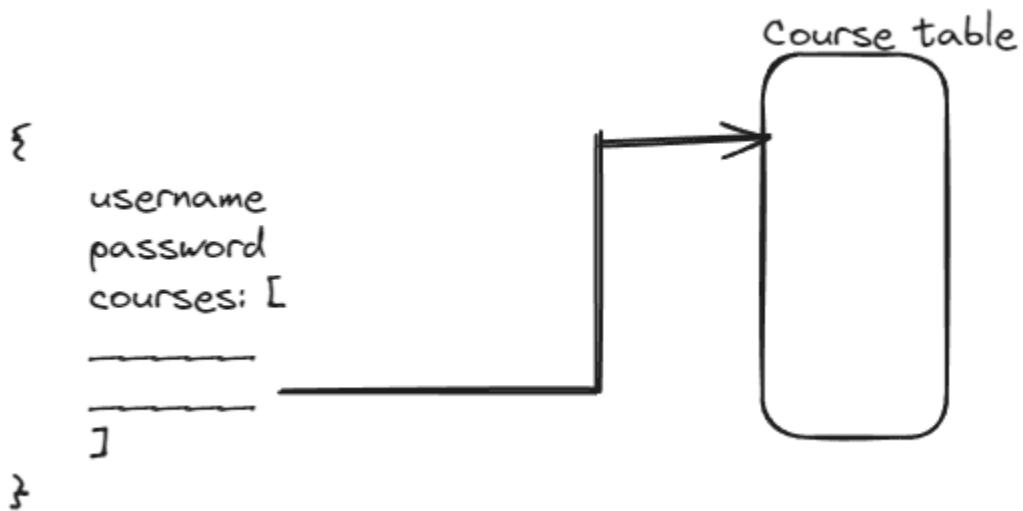
```
{  
  'id': {  
    'Sold' : 'ascasr123'  
  },  
  "name": "Thapu",  
  "courses": [  
    "0 to 0 Football fighting course",  
    "Unborn kids boxing tutorials"  
  ]  
}
```

In SQL we have table of Username, age, profession etc

The way to create relationship in MongoDB is by just creating array of things.



Another way( No SQL ) of doing thing can be:



### Create:

```
const UserSchema = new mongoose.Schema({  
  username: String,  
  password: String  
});  
  
const User = mongoose.model('User', UserSchema);  
  
User.create({  
  username: req.body.username,  
  password: req.body.password  
});
```

Entry to the UserTable

### Read:

```

const UserSchema = new mongoose.Schema({
  username: String,
  password: String
});

const User = mongoose.model('User', UserSchema);

```

→

```

User.findById("1");
User.findOne({
  username: "harkirat@gmail.com"
})
User.find([
  username: "harkirat96@gmail.com"
])

```

```

User.updateOne(
  { "id": "1" },
  { $push: { purchasedCourses: courseId } }
)

```

.findOne() : find a single entry

Syntax of pushing new course

## Update:

```

const UserSchema = new mongoose.Schema({
  username: String,
  password: String
});

const User = mongoose.model('User', UserSchema);

```

→

```

User.updateOne({
  id: "1"
}, {
  password: "newPassword"
})

User.update({}, [
  premium: true
])

```

Giving free access to our courses

## Delete:

```

const UserSchema = new mongoose.Schema({
  username: String,
  password: String
});

const User = mongoose.model('User', UserSchema);

```

→

```

User.deleteMany({})

User.deleteOne({
  username: "harkirat@gmail.com"
})

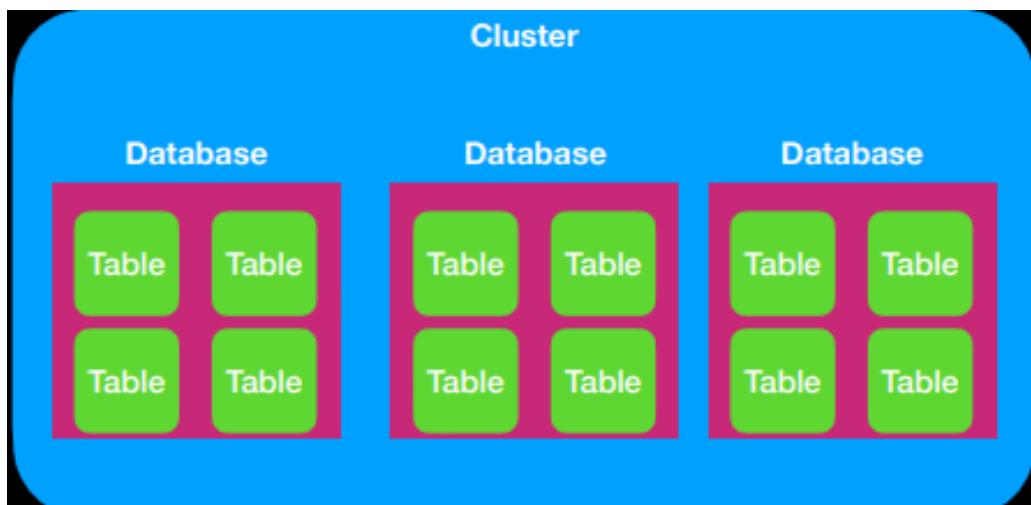
```

## User.deleteMany()

Deleting everything from the user table

## 3 Jargons to know in Databases

1. Cluster
2. Database
3. Table



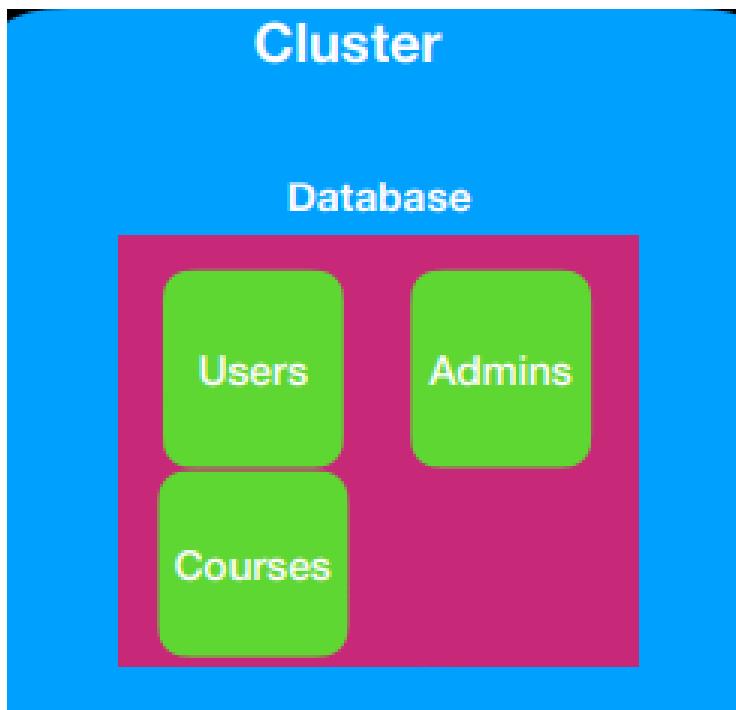
Lets see what would this mean for a simple course App:

Screenshot of a course platform showing a grid of Python-related courses:

Course Title	Rating	Reviews	Price
The Complete Python Bootcamp: From Zero to Hero!	4.8	188,710	£129
Automate the Boring Stuff with Python Programming	4.0	10,695	£529
100 Days of Codes: The Complete Python Project On-A-Dash	4.7	200,220	£529
Machine Learning AI & ML with Python & R + ChatGPT Projects	4.5	271,005	£529
Python : Master Programming and Development with 18+ Projects	4.2	33,395	£549

How learners like you are achieving their goals

Testimonial	Rating	Comments
I am proud to say that after a few months of taking this course...I passed my exam and am now an AWS Certified Cloud Practitioner	4.6	This course helped me refresh up on my product manager skills and land a job at Facebook! Thanks, guys!
One of the best courses on management and leadership I have come across so far. The advice is practical and examples highly	4.6	I highly recommend



User Table:

Users				
Id	Email	Password	Name	Age
1	<a href="mailto:harkirat@gmail.com">harkirat@gmail.com</a>	123123	harkirat	20
2	<a href="mailto:raman@gmail.com">raman@gmail.com</a>	kirat123	harkirat	22

Courses Table:

Courses			
id	Title	Description	Price
1.	Full stack	Learn Full stack	5000
2.	Web3	Learn Web3.	3999

## Purchases Table:

Purchases			
user_id	course_id	timestamp	payment_ref
1.	1	02/12/2024.	pay_123123
2.	1	02/12/2024.	pay_331213

## ASSIGNMENT

```
## Create a course selling website

### Description
You need to implement a course selling app. Make sure you setup your own
mongodb instance before starting.
mongodb+srv://admin:<password>@cluster0.9gr3ic2.mongodb.net/

It needs to support two types of users -
1. Admins
2. Users

Admins are allowed to sign up, create courses.
Users are allowed to sign up, view courses, purchase courses.
This in the real world would translate to an app like udemy.

This one doesn't use authentication the right way. We will learn how to do
that in the next assignment.
For this one, in every authenticated requests, you need to send the
username and password in the headers (and not the jwt).
This is the reason why this assignment doesn't have a sign in route.

You need to use mongodb to store all the data persistently.

## Routes
### Admin Routes:
- POST /admin/signup
```

```
Description: Creates a new admin account.
Input Body: { username: 'admin', password: 'pass' }
Output: { message: 'Admin created successfully' }
- POST /admin/courses
  Description: Creates a new course.
  Input: Headers: { 'username': 'username', 'password': 'password' },
  Body: { title: 'course title', description: 'course description', price: 100, imageLink: 'https://linktoimage.com' }
  Output: { message: 'Course created successfully', courseId: "new course id" }
- GET /admin/courses
  Description: Returns all the courses.
  Input: Headers: { 'username': 'username', 'password': 'password' }
  Output: { courses: [ { id: 1, title: 'course title', description: 'course description', price: 100, imageLink: 'https://linktoimage.com', published: true }, ... ] }

### User routes
- POST /users/signup
  Description: Creates a new user account.
  Input: { username: 'user', password: 'pass' }
  Output: { message: 'User created successfully' }
- GET /users/courses
  Description: Lists all the courses.
  Input: Headers: { 'username': 'username', 'password': 'password' }
  Output: { courses: [ { id: 1, title: 'course title', description: 'course description', price: 100, imageLink: 'https://linktoimage.com', published: true }, ... ] }
- POST /users/courses/:courseId
  Description: Purchases a course. courseId in the URL path should be replaced with the ID of the course to be purchased.
  Input: Headers: { 'username': 'username', 'password': 'password' }
  Output: { message: 'Course purchased successfully' }
- GET /users/purchasedCourses
  Description: Lists all the courses purchased by the user.
  Input: Headers: { 'username': 'username', 'password': 'password' }
  Output: { purchasedCourses: [ { id: 1, title: 'course title', description: 'course description', price: 100, imageLink: 'https://linktoimage.com', published: true }, ... ] }
```

## Goal of sign-in is to send token or jwt

We sign in to MongoDB , and connect our cluster to the MongoDBCompass  
Then we will create a new data base by the name of courses and inside it we will have a table/collection by the name of Courses.

The screenshot shows the MongoDB Compass interface. The top navigation bar includes 'My Queries', 'Databases', 'Courses', and a selected 'courses' tab. Below the navigation is a toolbar with 'Create collection', 'Refresh', 'View', 'Sort by', and a dropdown for 'Collection Name'. The main area is titled 'Courses' and displays statistics: Storage size: 4.10 kB, Documents: 0, Avg. document size: 0 B, Indexes: 1, and Total index size: 4.10 kB. On the left sidebar, under 'Databases', the 'courses' database is selected, showing its contents: 'Courses', 'local', 'sample\_mflix', 'test', and 'user\_app'.

Starting Code:

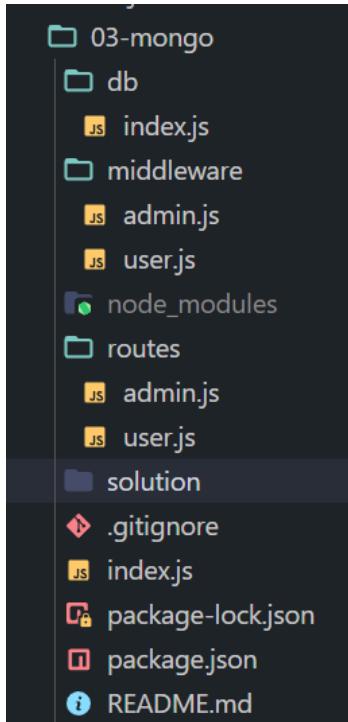
**index.js**

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const adminRouter = require("./routes/admin")
const userRouter = require("./routes/user");

// Middleware for parsing request bodies
app.use(bodyParser.json());
// all the admin go here
app.use("/admin", adminRouter)
app.use("/user", userRouter)
const PORT = 3000;
```

```
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

## File Structure



## db > index.js

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

// Connect to MongoDB
mongoose.connect('your-mongodb-url');

// Define schemas
const AdminSchema = new mongoose.Schema({
  // Schema definition here
});

const UserSchema = new mongoose.Schema({
  // Schema definition here
});
```

```
const CourseSchema = new mongoose.Schema({
    // Schema definition here
});

// tables required
const Admin = mongoose.model('Admin', AdminSchema);
const User = mongoose.model('User', UserSchema);
const Course = mongoose.model('Course', CourseSchema);

module.exports = {
    Admin,
    User,
    Course
}
```

## middleware > admin.js

```
// Middleware for handling auth
function adminMiddleware(req, res, next) {
    // Implement admin auth logic
    // You need to check the headers and validate the admin from the admin
DB. Check readme for the exact headers to be expected
}

// does the validation logic

module.exports = adminMiddleware;
```

## middleware > user.js

```
function userMiddleware(req, res, next) {
    // Implement user auth logic
    // You need to check the headers and validate the user from the user
DB. Check readme for the exact headers to be expected
}

module.exports = userMiddleware;
```

## routes > admin.js

```
const { Router } = require("express");
```

```

const adminMiddleware = require("../middleware/admin");
const router = Router();

// doesn't mean this handle the /signup endpoint
// But it handles /admin/signup
// Because in the main index.js file we have used app.use()
// Admin Routes
router.post('/signup', (req, res) => {
    // Implement admin signup logic
});

// protected by adminMiddleware
// any request for /admin/courses will reach here
router.post('/courses', adminMiddleware, (req, res) => {
    // Implement course creation logic
});

router.get('/courses', adminMiddleware, (req, res) => {
    // Implement fetching all courses logic
});

module.exports = router;

```

## routes > user.js

```

const { Router } = require("express");
const router = Router();
const userMiddleware = require("../middleware/user");
// User Routes
// one way to structure the express app
router.post('/signup', (req, res) => {
    // Implement user signup logic
});

router.get('/courses', (req, res) => {
    // Implement listing all courses logic
});

router.post('/courses/:courseId', userMiddleware, (req, res) => {
    // Implement course purchase logic
});

router.get('/purchasedCourses', userMiddleware, (req, res) => {
    // Implement fetching purchased courses logic
});

module.exports = router

```

## To run this application

- **npm install**
- **node index.js**

## Lets first define Our database schema.

### db/index.js

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

// Connect to MongoDB
mongoose.connect('mongodb+srv://admin:KmCihXn011podXRj@cluster0.9gr3ic2.mongodb.net/');

// Define schemas
const AdminSchema = new mongoose.Schema({
    // Schema definition here
    username: String,
    password: String
});

const UserSchema = new mongoose.Schema({
    // Schema definition here
    username: String,
    password: String,
    purchasedCourses: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Course'
            // it refer to course table
        }
    ]
});

const CourseSchema = new mongoose.Schema({
    // course Schema definition here
    title: String,
    description: String,
    imageLink: String,
    price: Number
});
```

```
// tables required
const Admin = mongoose.model('Admin', AdminSchema);
const User = mongoose.model('User', UserSchema);
const Course = mongoose.model('Course', CourseSchema);

module.exports = {
  Admin,
  User,
  Course
}
```

## middleware/admin.js

```
const {Admin} = require("../db");

// Middleware for handling auth
function adminMiddleware(req, res, next) {
  // Implement admin auth logic
  // You need to check the headers and validate the admin from the admin
DB.
  // Check readme for the exact headers to be expected
  const username = req.headers.username; //nishu123@gmail.com
  const password = req.headers.password;// Hakunamata
  // I have to check does this username actually exist in the admin
database
  // Step 1: I need to get the admin model
  Admin.findOne({
    username: username,
    password: password
  })//we can use async await syntax
  .then(function(value){
    if(value){ // if value exist then
      next();
      //If there exists a user with the above username and password
      inside the database then we call next or basically control reaches next,
      (inside routes/admin.js)
    }else{
      // couldn't found single user
      // possibility that username was matched but the user had
      different password
    }
  })
}

module.exports = adminMiddleware;
```

```

        res.status(403).json({
            msg: 'User doesnt exist'
        })
        // only able to access if they had their sign up
    }
})
}

// does the validation logic

module.exports = adminMiddleware;

```

## middleware/user.js

```

const { User } = require("../db");

function userMiddleware(req, res, next) {
    // Implement user auth logic
    // You need to check the headers and validate the user from the user
DB. Check readme for the exact headers to be expected
    const username = req.headers.username;
    const password = req.headers.password;

    User.findOne({
        username:username,
        password: password
    })
    .then(function(value){
        if(value){
            next();
        }else{
            res.status(403).json({
                msg:"User doesn't exist"
            })
        }
    })
}

module.exports = userMiddleware;

```

## routes/admin.js

```
const { Router } = require("express");
const adminMiddleware = require("../middleware/admin");
const { Admin } = require("../db");
const router = Router();

// doesn't mean this handle the /signup endpoint
// But it handles /admin/signup
// Because in the main index.js file we have used app.use()
// Admin Routes
router.post('/signup', async (req, res) => {
    // Implement admin signup logic
    // Readme file must be read
    const username = req.body.username;
    const password = req.body.password;
    // check if a user with this username already exist , if it exist then
    // we must stop this request

    await Admin.create({
        username: username,
        password: password
    })
    res.json({
        message: "Admin created successfully"
    })
    // .then(function() {
    //     res.json({
    //         message: 'Admin created successfully'
    //     })
    // })
    // .then() ideally we must await so that there was no network issue
    // etc.
    // .catch(function() {
    // to print the error
    // })
    // if they do not exist then we must insert something inside the
    // database.
});
// protected by adminMiddleware
```

```
// any request for /admin/courses will reach here
router.post('/courses', adminMiddleware, (req, res) => {
    // not accessible until the sign up is done
    // Implement course creation logic
});

router.get('/courses', adminMiddleware, (req, res) => {
    // Implement fetching all courses logic
});

module.exports = router;
```

## Testing it on POSTMAN

The screenshot shows the Postman application interface. At the top, there is a header with 'POST' selected, the URL 'http://localhost:3000/admin/signup', and a 'Send' button. Below the header, there are tabs for 'Params', 'Auth', 'Headers (8)', 'Body', 'Pre-req.', 'Tests', and 'Settings'. The 'Body' tab is currently active and has a 'JSON' dropdown set to 'raw'. The JSON body is defined as follows:

```
1 {  
2   "username": "abcd@gmail.com",  
3   "password": "1234567890"  
4 }
```

At the bottom of the interface, there is a status bar showing '200 OK', '354 ms', '274 B', and a 'Save Response' button.

**How can we verify where our data is stored**

**test.admins**

4 DOCUMENTS 1 INDEXES

Documents Aggregations Schema Indexes Validation

Type a query: { field: 'value' } Explain Reset Find Options

**ADD DATA** EXPORT DATA

1 - 5 of 5

**Document 1:**

```
_password: "1234"
__v: 0
```

**Document 2:**

```
_id: ObjectId('65d62f60fcb49f409475cd1a')
username: "abcd@gmail.com"
password: "1234567890"
__v: 0
```

**Document 3:**

```
_id: ObjectId('65d6322bfcb49f409475cd1c')
username: "abcd@gmail.com"
password: "1234567890"
__v: 0
```

**Document 4:**

```
_id: ObjectId('65d632d528d4c15ee9f42775')
username: "abcd@gmail.com"
password: "1234567890"
__v: 0
```

**Document 5:**

```
_id: ObjectId('65d633cd28d4c15ee9f42777')
username: "abcde@gmail.com"
password: "1234567890"
__v: 0
```

**It automatically created test with three collection/table**

**(it is default mongoDB)**

**If we want to choose Course DB we can do it by changing url**

```
mongoose.connect('mongodb+srv://admin:KmCihXn011podXRj@cluster0.9gr3ic2.mongodb.net/course_selling_app');
```

Now it will be stored in the course\_selling\_app database

Now again send a POST request from POSTMAN

The screenshot shows the MongoDB Compass interface. On the left, the sidebar lists databases: admin, course\_selling\_app, local, and sample\_mflix. The course\_selling\_app database is expanded, showing collections: admins, courses, and users. The admins collection is selected, indicated by a green background. On the right, the main area displays the 'Documents' tab. It includes a search bar, filter options, and buttons for 'ADD DATA', 'EXPORT DATA', and 'Find'. Below these are three document cards. Each card shows an ObjectId for '\_id', 'username : "abcde@gmail.com"', 'password : "1234567890"', and '\_\_v : 0'.

```

_id: ObjectId('65d634f6da3222549cf14ab')
username : "abcde@gmail.com"
password : "1234567890"
__v : 0

_id: ObjectId('65d634fcda3222549cf14ad')
username : "abcdef@gmail.com"
password : "1234567890"
__v : 0

_id: ObjectId('65d63500da3222549cf14af')
username : "abcdef@gmail.com"
password : "1234567890"
__v : 0

```

## Now we will add course adding endpoint

```

router.post('/courses', adminMiddleware,async (req, res) => {
    // not accessible until the sign up is done
    // Implement course creation logic
    // need username and password
    const title = req.body.title;
    const description = req.body.description;
    const imageLink = req.body.imageLink;
    const price = req.body.price;
    // zod should be used here
    const newCourse = await Course.create({
        title,
        // title:title
        // both same if key and value are same
        description,
        imageLink,
        price
    })
    res.json({
        // in mongoDB any data we entered is given some id randomly
        // now to know the courseId of just enter data in database we need
        to await
    })
}

```

```
        message:"Course created successfully",
        courseId: newCourse._id
    })
}) ;

```

## Lets check it on POSTMAN

In headers we pass the username and password.

And in body we send title, description, imageLink, price

### Case 1: If we enter the wrong user (user not in database admin)

The screenshot shows a POST request to `http://localhost:3000/admin/courses`. The Headers tab is selected, showing the following configuration:

Key	Value
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
username	abcd@gmail.com
password	1234567890

The Body tab shows the response received:

```
1 {  
2   "msg": "User doesn't exist"  
3 }
```

The response status is 403 Forbidden, with a time of 254 ms and a size of 269 B.

### Case 2: User do exist

POST http://localhost:3000/admin/courses

**Headers (10)**

Accept-Encoding	gzip, deflate, br
Connection	keep-alive
username	abcde@gmail.com
password	1234567890
Key	Value

**Body**

```

1   "message": "Course created successfully",
2   "courseId": "65d7672ae51b7facb3805b40"
3
4

```

200 OK 408 ms 313 B Save Response

Now let's check our database in MongoDB compass:

## course\_selling\_app.courses

Documents Aggregations Schema Indexes Validation

Filter ⏳ Type a query: { field: 'value' } Explain Reset

+ ADD DATA EXPORT DATA

1 - 1 of 1

```

_id: ObjectId('65d7672ae51b7facb3805b40')
title: "Web_3"
description: "NishuDev"
imageLink: "https://google.com/cat.png"
price: 1234
__v: 0

```

Lets work on last endpoint which return admin all its courses

```

router.get('/courses', adminMiddleware, (req, res) => {
  // Implement fetching all courses logic
}

```

```

Course.find({})
  .then(function(response) {
    res.json({
      courses: response
    })
  })
})

// router.get('/courses', adminMiddleware, async(req,res) => {
//   const response = await Course.find({})
//   res.json({
//     courses: response
//   })
// })

```

The screenshot shows a Postman interface with a GET request to `http://localhost:3000/admin/courses`. The response body is displayed in Pretty JSON format, showing an array of three course objects:

```

1  [
2    {
3      "courses": []
4      {
5        "_id": "65d7672ae51b7facb3805b40",
6        "title": "Web_3",
7        "description": "NishuDev",
8        "imageLink": "https://google.com/cat.png",
9        "price": 1234,
10       "__v": 0
11     },
12     {
13       "_id": "65d76835e51b7facb3805b43",
14       "title": "Android",
15       "description": "NishuDev 0-100",
16       "imageLink": "https://google.com/cat.png",
17       "price": 1234,
18       "__v": 0
19     },
20     {
21       "_id": "65d76842e51b7facb3805b46",
22       "title": "FullStack",
23       "description": "NishuDev 0-100",
24       "imageLink": "https://google.com/cat.png",
25     }
26   ]

```

## Let say we send the wrong user

The screenshot shows a Postman interface with a request to `http://localhost:3000/admin/courses`. The `Headers` tab is selected, displaying the following key-value pairs:

Key	Value
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
username	abcde@gmail.com
password	123456789

The response section shows a 403 Forbidden status with the message `"msg": "User doesnt exist"`.

Authentication in real world take the help of jwt.

### route/admin.js

```
const { Router } = require("express");
const adminMiddleware = require("../middleware/admin");
const { Admin, Course } = require("../db");
const router = Router();

router.post('/signup', async (req, res) => {
    // Implement admin signup logic
    // Readme file must be read
    const username = req.body.username;
    const password = req.body.password;
    // check if a user with this username already exist , if it exist then
    // we must stop this request

    await Admin.create({
        username: username,
        password: password
    })
    res.json({
        message: "Admin created successfully"
})
```

```
})

});

router.post('/courses', adminMiddleware,async (req, res) => {
    // not accessible until the sign up is done
    // Implement course creation logic
    // need username and password
    const title = req.body.title;
    const description = req.body.description;
    const imageLink = req.body.imageLink;
    const price = req.body.price;
    // zod should be used here
    const newCourse = await Course.create({
        title,
        // title:title
        // both same if key and value are same
        description,
        imageLink,
        price
    })
    res.json({
        // in mongoDB any data we entered is given some id randomly
        // now to know the courseId of just enter data in database we need
        to await
        message:"Course created succesfully",
        courseId: newCourse._id
    })
});

router.get('/courses', adminMiddleware, (req, res) => {
    // Implement fetching all courses logic
    Course.find({})
    .then(function(response) {
        res.json({
            courses: response
        })
    })
});

module.exports = router;
```

## Now working on the user endpoints

- signup endpoint

```
// User Routes
// one way to structure the express app

router.post('/signup', async(req, res) => {
    // Implement user signup logic
    // Implement admin signup logic
    // Readme file must be read
    const username = req.body.username;
    const password = req.body.password;
    // check if a user with this username already exist , if it exist then
    we must stop this request

    await User.create({
        username: username,
        password: password
    })
    res.json({
        message: "User created successfully"
    })
});
```

The screenshot displays two windows side-by-side. On the left is the MongoDB Compass interface, showing the 'course\_selling\_app' database with the 'users' collection selected. Two documents are visible in the list view. On the right is a Postman API client window, showing a successful POST request to 'http://localhost:3000/signup'. The request body is set to 'JSON' and contains the following data:

```
1 _id: ObjectId('65d76cd4fbc79af709210bbe')
2   "username": "edcba@gmail.com",
3   "password": "0987654321"
4   > purchasedCourses: Array (empty)
5   __v: 0
```

The response body is:

```
1   "message": "User created successfully"
```

Endpoint displaying all the courses published to the user.

```

router.get('/courses', async(req, res) => {
    // Implement listing all courses logic
    // this can be open endpoint , listing all the courses present just
like in udemy
    // so the question may arise why we created exactly same endpoint in
the admin , it is because in the application like udemy there is a header
called ispublished: yes/no which are only visible to the admins and list
shown to the user is filtered.
    const response = await Course.find({
        })
        res.json({
            courses:response
        })
    });

```

The screenshot shows a Postman interface with a GET request to `http://localhost:3000/user/courses`. The request body is set to `JSON` and contains:

```

1  ...
2  ...
3  ...
4  ...

```

The response is displayed in a `Pretty` JSON format:

```

1  "courses": [
2      {
3          "_id": "65d7672ae51b7facb3805b40",
4          "title": "Web_3",
5          "description": "NishuDev",
6          "imageLink": "https://google.com/cat.png",
7          "price": 1234,
8          "__v": 0
9      },
10     {
11         "_id": "65d76835e51b7facb3805b43",
12         "title": "Android",
13         "description": "NishuDev 0-100",
14         "imageLink": "https://google.com/cat.png",
15         "price": 1234,
16         ...
17     }
18 ]

```

## purchased course endpoint

In the real-world project we hit the razor pay API then the course is purchased else not.

```
router.post('/courses/:courseId', userMiddleware, async (req, res) => {
    // Implement course purchase logic
    // We don't have a purchase table
    // get back the course id which user want to buy
    // we need to extract courseId from the url
    const courseId = req.params.courseId;
    const username = req.headers.username;
    // input validation using zod
    // enough to update the user table in mongoDB
    await User.updateOne({
        username: username
    }, {
        "$push": {
            purchasedCourses: courseId
        }
    })
    // WRONG
    // purchasedCourses:{}
    //     "$push":courseId
    // }
    // if we see error we can log it
})
res.json({
    message: "purchased complete"
})
});
```

POST http://localhost:3000/user/courses/65d7672ae51b7facb3805b40

**Headers (12)**

- Authorization: 123456778
- username: edcba@gmail.com
- password: 0987654321
- authorization: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVC...

**Body**

```

1
2   "message": "purchased complete"
3

```

Type a query: { field: 'value' } Exp [51]

**DD DATA** EXPORT DATA

```

_id: ObjectId('65d76cd4fbc79af709210bbe')
username : "dcba@gmail.com"
password : "0987654321"
purchasedCourses : Array (empty)
--v : 0

_id: ObjectId('65d76ce7fbc79af709210bc0')
username : "edcba@gmail.com"
password : "0987654321"
purchasedCourses : Array (1)
  0: ObjectId('65d76835e51b7facb3805b43')
--v : 0

```

POST http://localhost:3000/user/courses/65d76835e51b7facb3805b43

**Headers (12)**

- username: edcba@gmail.com
- password: 0987654321
- authorization: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVC...

**Body**

```

1
2   "message": "purchased complete"
3

```

Now lets work on last endpoint of the users showing their purchased courses.

```

router.get('/purchasedCourses', userMiddleware,async (req, res) => {
    // Implement fetching purchased courses logic
    // This endpoint is used so that user can see their purchased courses
    // in the udemy dashboard.
    // get purchased id from the user table
    const user = await User.findOne({
        username: req.headers.username
    })
    // Create a new purchasedCourses document
    const purchasedCourses = new PurchasedCourses({
        user: user._id,
        course: req.query.course
    })
    // Save the purchasedCourses document
    await purchasedCourses.save()
    // Return the purchasedCourses document
    res.json(purchasedCourses)
})

```

```

})
// console.log(user.purchasedCourses)
// we get an array of purchasedCoursesId
const courses = await Course.find({
  courseId: {
    "$in": user.purchasedCourses
    // how to do find on mongoose on array
  }
})
res.json({
  courses: courses
})
}) ;

```

When we check on the POSTMAN there is some error lets see

The screenshot shows a Postman interface with a GET request to `http://localhost:3000/user/purchasedCourses`. The Headers tab is selected, showing the following values:

Header	Value
Authorization	123456778
username	edcba@gmail.com
password	0987654321
authorization	eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVC

The Body tab shows a JSON response with the following structure:

```

1  [
2   "courses": []
3 ]

```

The response status is 200 OK with a 360 ms duration and 248 B size.

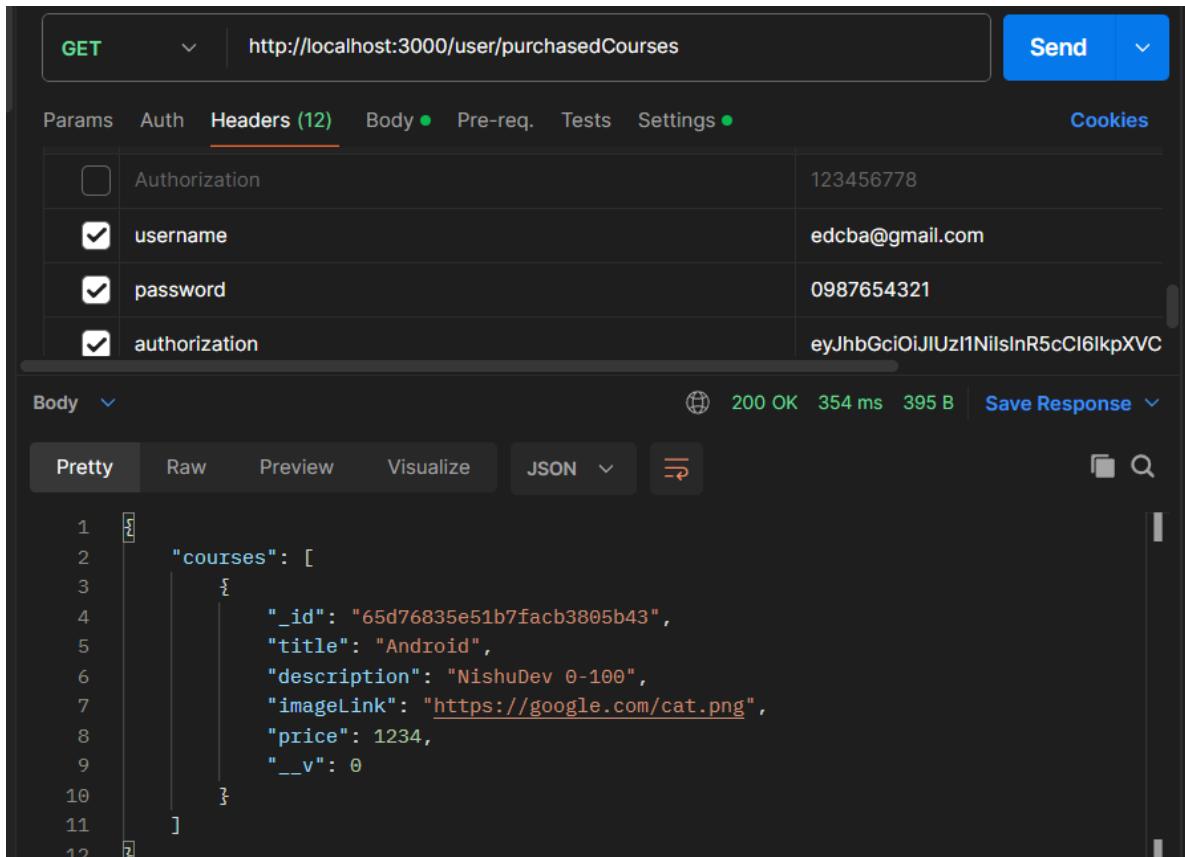
We see an empty purchasedCourse array of the given user.

```

const courses = await Course.find({
  _id: {
    "$in": user.purchasedCourses
    // how to do find on mongoose on array
  }
})

```

Now lets again test it on POSTMAN:



The screenshot shows the POSTMAN interface with a successful API call. The URL is `http://localhost:3000/user/purchasedCourses`. The Headers tab is selected, showing four entries: Authorization (value: 123456778), username (value: edcba@gmail.com), password (value: 0987654321), and authorization (value: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVC). The Body tab shows a JSON response with a single course object:

```
1 "courses": [
2   {
3     "_id": "65d76835e51b7facb3805b43",
4     "title": "Android",
5     "description": "NishuDev 0-100",
6     "imageLink": "https://google.com/cat.png",
7     "price": 1234,
8     "__v": 0
9   }
10 ]
11 ]
12 ]
```

### routes/user.js

```
const { Router } = require("express");
const router = Router();
const userMiddleware = require("../middleware/user");
const { User, Course } = require("../db");

// User Routes
// one way to structure the express app

router.post('/signup', async(req, res) => {
  // Implement user signup logic
  // Implement admin signup logic
  // Readme file must be read
  const username = req.body.username;
  const password = req.body.password;
```

```
// check if a user with this username already exist , it it exist then  
we must stop this request  
  
await User.create({  
    username: username,  
    password: password  
})  
res.json({  
    message:"User created successfully"  
})  
});  
  
router.get('/courses', async(req, res) => {  
    // Implement listing all courses logic  
    // this can be open endpoint , listing all the courses present just  
like in udemy  
    // so the question may arise why we created exactly same endpoint in  
the admin , it is because in the application like udemy there is a header  
called ispublished: yes/no which are only visible to the admins and list  
shown to the user is filtered.  
    const response = await Course.find({});  
    res.json({  
        courses:response  
    })  
});  
  
router.post('/courses/:courseId', userMiddleware,async (req, res) => {  
    // Implement course purchase logic  
    // We don't have a purchase table  
    // get back the course id which user want to buy  
    // we need to extract courseId from the url  
    const courseId = req.params.courseId;  
    const username = req.headers.username;  
    // input validatiion using zod  
    // enough to update the user table in mongoDB  
    await User.updateOne({  
        username: username  
    }, {  
        "$push":{  
            purchasedCourses: courseId  
    }  
});
```

```

        }
        // WRONG
        // purchasedCourses: {
        //     "$push": courseId
        // }
        // if we see error we can log it
    })
    res.json({
        message: "purchased complete"
    })
);

router.get('/purchasedCourses', userMiddleware,async (req, res) => {
    // Implement fetching purchased courses logic
    // This endpoint is used so that user can see their purchased courses
    // in the udemy dashboard.
    // get purchased id from the user table
    const user = await User.findOne({
        username: req.headers.username
    })
    // console.log(user.purchasedCourses)
    // we get an array of purchasedCoursesId
    const courses = await Course.find({
        _id: {
            "$in": user.purchasedCourses
            // how to do find on mongoose on array
        }
    })
    res.json({
        courses: courses
    })
);

module.exports = router

```

Bearer tells the type of token

When sending the authorization header.

## Assignment 2:

```
## Create a course-selling website

### Description

Same as the last assignment but you need to use jwts for authentication.
We have introduced the signgin endpoints for both users and admins.
For this one, in every authenticated requests, you need to send the jwt in
headers (Authorization : "Bearer <actual token>").
You need to use mongodb to store all the data persistently.

## Routes

### Admin Routes:

- POST /admin/signup
  Description: Creates a new admin account.
  Input Body: { username: 'admin', password: 'pass' }
  Output: { message: 'Admin created successfully' }
- POST /admin/signin
  Description: Logs in an admin account.
  Input Body: { username: 'admin', password: 'pass' }
  Output: { token: 'your-token' }
- POST /admin/courses
  Description: Creates a new course.
  Input: Headers: { 'Authorization': 'Bearer <your-token>' }, Body: {
    title: 'course title', description: 'course description', price: 100,
    imageLink: 'https://linktoimage.com' }
  Output: { message: 'Course created successfully', courseId: "new course
id" }
- GET /admin/courses
  Description: Returns all the courses.
  Input: Headers: { 'Authorization': 'Bearer <your-token>' }
  Output: { courses: [ { id: 1, title: 'course title', description:
    'course description', price: 100, imageLink: 'https://linktoimage.com',
    published: true }, ... ] }

### User routes
```

```

- POST /users/signup
  Description: Creates a new user account.
  Input: { username: 'user', password: 'pass' }
  Output: { message: 'User created successfully' }

- POST /users/signin
  Description: Logs in a user account.
  Input: { username: 'user', password: 'pass' }
  Output: { token: 'your-token' }

- GET /users/courses
  Description: Lists all the courses.
  Input: Headers: { 'Authorization': 'Bearer <your-token>' }
  Output: { courses: [ { id: 1, title: 'course title', description: 'course description', price: 100, imageLink: 'https://linktoimage.com', published: true }, ... ] }

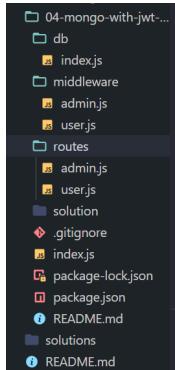
- POST /users/courses/:courseId
  Description: Purchases a course. courseId in the URL path should be replaced with the ID of the course to be purchased.
  Input: Headers: { 'Authorization': 'Bearer <your-token>' }
  Output: { message: 'Course purchased successfully' }

- GET /users/purchasedCourses
  Description: Lists all the courses purchased by the user.
  Input: Headers: { 'Authorization': 'Bearer <your-token>' }
  Output: { purchasedCourses: [ { id: 1, title: 'course title', description: 'course description', price: 100, imageLink: 'https://linktoimage.com', published: true }, ... ] }

```

## File Structure

Download from the assignment Week-3 , assignment 4



## db/index.js

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

// Connect to MongoDB
mongoose.connect('mongodb+srv://admin:KmCihXn011podXRj@cluster0.9gr3ic2.mongo.net/course_selling_app');

// Define schemas
const AdminSchema = new mongoose.Schema({
    // Schema definition here
    username: String,
    password: String
});

const UserSchema = new mongoose.Schema({
    // Schema definition here
    username: String,
    password: String,
    purchasedCourses: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Course'
            // it refer to course table
        }
    ]
});

const CourseSchema = new mongoose.Schema({
    // course Schema definition here
    title: String,
    description: String,
    imageLink: String,
    price: Number
});

// tables required
const Admin = mongoose.model('Admin', AdminSchema);
const User = mongoose.model('User', UserSchema);
const Course = mongoose.model('Course', CourseSchema);

module.exports = {
    Admin,
```

```
User,  
Course  
}
```

## Now we will work on middleware

Lets first set up the project

cd on the working directory

npm install

npm install jsonwebtoken

node index.js

### middleware/admin.js

```
const jwt = require("jsonwebtoken")  
const secret = require("../index");  
  
// Middleware for handling auth  
function adminMiddleware(req, res, next) {  
    // Implement admin auth logic  
    // You need to check the headers and validate the admin from the admin  
DB. Check readme for the exact headers to be expected  
    // While accessing headers everything get converted to the lowercase  
    const token = req.headers.authorization;  
    // we could have seen this by console.log(req.autho...)  
    // Bearer asfgsddhfdsafdgdsasdfsadGDF get back token from string  
    const words = token.split(" ");  
    const jwtToken = words[1];  
    // from here we do the jwt verification  
    // there should be some global secret which used to generate the token  
and verify the token  
    const decodedValue = jwt.verify(jwtToken, secret);  
    if( decodedValue.username){  
        // where does this username come from??  
        next();  
    }else{  
        res.status(403).json({  
            msg:"You are not authmeticated"  
        })  
    }  
}
```

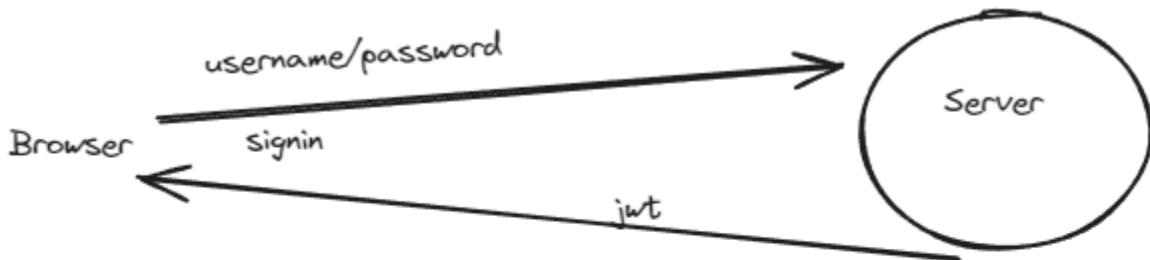
```

    }
}

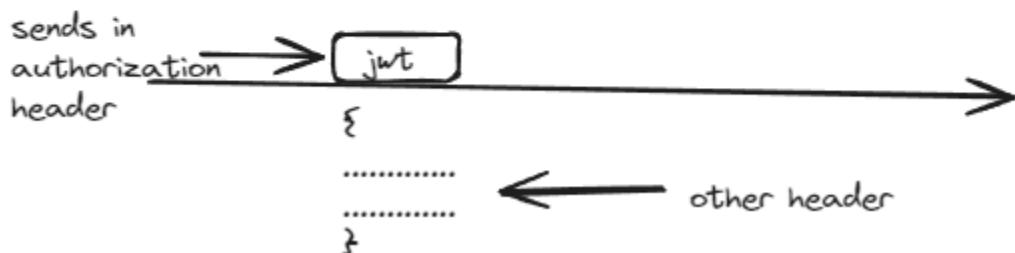
module.exports = adminMiddleware;

```

## What are we doing



Now whatever authenticated request browser need to send



Jwt has encoded version of the username .

Never send password in jwt.

```

if( decodedValue.username) {
    // where does this username come from??
    next();
}

```

Middleware uses this to extract the username from the jwt

**middleware/admin.js**

```

const jwt = require("jsonwebtoken")
const secret = require("../index");

// Middleware for handling auth

```

```

function adminMiddleware(req, res, next) {
    // Implement admin auth logic
    // You need to check the headers and validate the admin from the admin
DB. Check readme for the exact headers to be expected
    // While accessing headers everything get converted to the lowercase
    const token = req.headers.authorization;
    // we could have seen this by console.log(req.autho..)
    // Bearer asfgsddhfdafdgdsasdfsadGDF get back token from string
    const words = token.split(" ");
    const jwtToken = words[1];
    // from here we do the jwt verification
    // there should be some global secret which used to generate the token
and verify the token
    const decodedValue = jwt.verify(jwtToken, secret);
    // why aren't we checking in the database that whether the user exist
in our database or not?
    // This is reason why jwt is powerfull, it save us a database call.
    // It save us 1DB call
    // It checks in inmemory that whether the jwt was created by me or
not.
    // We just have to send jwt once the user sign only one time , this
jwt which was send is enough to verify that user exist or not.
    if( decodedValue.username){
        // where does this username come from??
        // if decoded.username exist then tell them that user exist
        next();
    }else{
        res.status(403).json({
            msg:"You are not authmeticated"
        })
    }
}

module.exports = adminMiddleware;

```

## middleware/user.js

```

function userMiddleware(req, res, next) {
    const token = req.headers.authorization;
    const words = token.split(" ");

```

```

const jwtToken = words[1];
const decodedValue = jwt.verify(jwtToken, secret);
// ideally we should send the type of user
// username, type: "admin" | "user"
// and check type in if block (This is called authorization)
if( decodedValue.username){
    next();
} else{
    res.status(403).json({
        msg:"You are not authmeticated"
    })
}
}

module.exports = userMiddleware;

```

## Let work on admin routes

- First we work on the signup endpoint

```

// admin routes
router.post('/signup',async (req, res) => {
    // Implement admin signup logic
    // Readme file must be read
    const username = req.body.username;
    const password = req.body.password;
    // check if a user with this username already exist , it it exist then
    // we must stop this request

    await Admin.create({
        username: username,
        password: password
    })
    res.json({
        message:"Admin created succesfully"
    })
});

```

POST <http://localhost:3000/admin/signup>

Params Auth Headers (12) Body ● Pre-req. Tests Settings ● Cookies

raw JSON Beautify

```

1 {
2   "username": "abcd@gmail.com",
3   "password": "1234567890"
4 }
```

Body 200 OK 188 ms 274 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "message": "Admin created successfully"
3 }
```

## Database:

My Queries

Performance

Databases

course\_selling\_app2.admins

Documents Aggregations Schema Indexes Validation

Filter Type a query: { field: 'value' } or [Generate query](#)

[ADD DATA](#) [EXPORT DATA](#) [UPDATE](#) [DELETE](#)

<code>_id: ObjectId('65d82f173c522274147ee508')</code>
<code>username : "abcd@gmail.com"</code>
<code>password : "1234567890"</code>
<code>__v : 0</code>

- signin endpoint

```

router.post('/signin',async (req, res) => {
  // Implement admin signup logic
  // Input Body: { username: 'admin', password: 'pass' }
  // Output: { token: 'your-token' }
  const username = req.body.username;
  const password = req.body.password;

  const user = await User.find({
    username,
    password
  })
})
```

```
})
if(user) {
    const token = jwt.sign({
        username
    },JWT_SECRET);
    res.json({
        token
    })
} else{
    res.status(411).json({
        message:"Incorrect email and password"
    })
}
});
```

Lets see whether it works or not.

We get an error.

Basically, we are creating a circular dependencies

index.js import admin route and admin import index.js

To solve this we will create a config.js file.

```
module.exports = {  
    JWT_SECRET : "thapa_server"
```

```

}
// making index.js return nothing

```

And make changes in rest of file where we import JWT\_SECRET see the final solution

Now the signin endpoint works:

The screenshot shows a Postman interface. The top bar has 'POST' selected and the URL 'http://localhost:3000/admin/signin'. Below the URL, there are tabs for 'Params', 'Auth', 'Headers (12)', 'Body' (which is active), 'Pre-req.', 'Tests', and 'Settings'. The 'Body' tab is set to 'JSON'. The request body contains the following JSON:

```

1  {
2    "username": "abcd@gmail.com",
3    "password": "1234567890"
4  }

```

Below the request, the response details are shown: '200 OK', '298 ms', '391 B', and a 'Save Response' button. The response body is displayed in 'Pretty' format:

```

1  {
2    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ1c2Vybmc6ImFiY2RAZ21haWwUY29tIiwiaWF0IjoxNzA4NjY3NTQ2fQ.
DtAYPdr01mwbkxgkCQ3g280Jf2ocX7J5sc3bD3bHZ8"
3  }

```

**Let implement the course making endpoint**

```

router.post('/courses', adminMiddleware, async (req, res) => {
  const title = req.body.title;
  const description = req.body.description;
  const imageLink = req.body.imageLink;
  const price = req.body.price;
  const newCourse = await Course.create({
    title,
    description,
    imageLink,
    price
  })
  res.json({
    message: "Course created successfully",
    courseId: newCourse._id
  })
});

```

POST http://localhost:3000/admin/courses

Body (JSON)

```
1 {  
2   "title": "FullStack_0to100",  
3   "description": "Full course by thapa",  
4   "price": 4999,  
5   "imageLink": "https://google.com/courseBhaiKa.png"  
6 }
```

Body

Pretty

```
1 {  
2   "message": "Course created successfully",  
3   "courseId": "65d8348ac9fcb6b326ef41b7"  
4 }
```

## In Headers we pass the Authorization

POST http://localhost:3000/admin/courses

Headers (12)

Key	Value
Authorization	123456778
username	edcba@gmail.com
password	0987654321
authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6Ikç

Body

Pretty

```
1 {  
2   "message": "Course created successfully",  
3   "courseId": "65d834fbc9fcb6b326ef41b9"  
4 }
```

The screenshot shows the MongoDB Compass interface. On the left, there's a sidebar with a tree view of databases and collections. The 'courses' collection under the 'course\_selling\_app2' database is selected and highlighted in green. The main area displays two documents from this collection:

```
_id: ObjectId('65d8348ac9fcbb6b326ef41b7')
title : "FullStack_0to100"
description : "Full course by thapa"
imageLink : "https://google.com/courseBhaiKa.png"
price : 4999
__v : 0

_id: ObjectId('65d834fbc9fcbb6b326ef41b9')
title : "Web3"
description : "Ethereum course by thapa"
imageLink : "https://google.com/courseBhaiKa.png"
price : 4999
__v : 0
```

## Lets implement the getting the purchased course list endpoint

```
router.get('/courses', adminMiddleware, (req, res) => {
    // Implement fetching all courses logic
    Course.find({})
        .then(function(response) {
            res.json({
                courses: response
            })
        })
    })
});
```

The screenshot shows a Postman request for `http://localhost:3000/admin/courses`. The Headers tab is selected, showing two entries: `password` (checkbox unchecked) and `authorization` (checkbox checked, value: `Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6Ik`). The Body tab shows a JSON response:

```
1 "courses": [
2   {
3     "_id": "65d8348ac9fcb6b326ef41b7",
4     "title": "FullStack_0to100",
5     "description": "Full course by thapa",
6     "imagelink": "https://google.com/courseBhaiKa.png",
7     "price": 4999,
8     "__v": 0
9   },
10  {
11    "_id": "65d834fbc9fcb6b326ef41b9",
12    "title": "Web3",
13    "description": "Ethereum course by thapa",
14    "imagelink": "https://google.com/courseBhaiKa.png",
15    "price": 4999,
16  }
]
```

## Suppose the token is wrong

The screenshot shows a Postman request for `http://localhost:3000/admin/courses`. The Headers tab is selected, showing three entries: `username` (checkbox checked, value: `euclida@gmail.com`), `password` (checkbox unchecked), and `authorization` (checkbox checked, value: `Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6Ik`). The Body tab shows an error response:

```
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Error</title>
</head>
<body>
  <pre>JsonWebTokenError: invalid token<br> &ampnbsp &ampnbspat module.exports [as verify]
  (C:\Users\NTC\Desktop\Dev\Week4\week-3\04-mongo-with-jwt-auth\node_modules\jsonwebtoken\verify.js:82:17)<br> &ampnbsp &ampnbspat adminMiddleware
  (C:\Users\NTC\Desktop\Dev\Week4\week-3\04-mongo-with-jwt-auth\middleware\admin.js:16:30)<br> &ampnbsp &ampnbspat Layer.handle [as handle_request]
  (C:\Users\NTC\Desktop\Dev\Week4\week-3\04-mongo-with-jwt-auth\node_modules\express\lib\router\layer.js:95:5)<br> &ampnbsp &ampnbspat next
```

We can handle it by doing try/ catch in admin middleware

### routes/admin.js

```
const {Router} = require("express")
const adminMiddleware = require("../middleware/admin");
const {Admin,Course} = require("../db")
const {JWT_SECRET} = require("../config")
const router = Router();
const jwt = require("jsonwebtoken")

// admin routes
router.post('/signup',async (req, res) => {
    // Implement admin signup logic
    // Readme file must be read
    const username = req.body.username;
    const password = req.body.password;
    // check if a user with this username already exist , if it exist then
    we must stop this request

    await Admin.create({
        username: username,
        password: password
    })
    res.json({
        message:"Admin created successfully"
    })
});

router.post('/signin',async (req, res) => {
    // Implement admin signin logic
    // Input Body: { username: 'admin', password: 'pass' }
    // Output: { token: 'your-token' }
    const username = req.body.username;
    const password = req.body.password;

    const user = await Admin.find({
        username,
        password
    })
})
```

```
if(user) {
    const token = jwt.sign({
        username
    },JWT_SECRET);
    res.json({
        token
    })
} else{
    res.status(411).json({
        message:"Incorrect email and password"
    })
}
};

router.post('/courses', adminMiddleware,async (req, res) => {
    const title = req.body.title;
    const description = req.body.description;
    const imageLink = req.body.imageLink;
    const price = req.body.price;
    // zod should be used here
    const newCourse = await Course.create({
        title,
        description,
        imageLink,
        price
    })
    res.json({
        message:"Course created succesfully",
        courseId: newCourse._id
    })
});

router.get('/courses', adminMiddleware, (req, res) => {
    // Implement fetching all courses logic
    Course.find({})
    .then(function(response){
        res.json({
            courses: response
        })
    })
});
```

```
}) ;

module.exports = router;
```

## Now implementing User endpoints

Mainly there is one endpoint with some change

Purchasing course endpoint

Here we are not passing username in the headers , because we are passing authorization in the headers . Because if we change the username then , different user will access courses with the same jwt token

Hence better approach is to keep the username in req object

```
if( decodedValue.username) {
    req.username = decodedValue.username;
    next();
} else{
    res.status(403).json({
        msg:"You are not authmeticated"
    })
}
```

Middleware suppose to do

- End the request
- Forward the request
- Pass data along to the next middleware

```
router.post('/courses/:courseId', userMiddleware, (req, res) => {
    // Implement course purchase logic
    const username = req.username;
    // without user explicitly sending their username we were able to
get the username
});
```