

Databases

First off, why do we need *databases*? Can't we just use a simple `.csv` file and import it, read it's contents and write to it?

To begin with, *Database is an organised collection of data*. Lets say you're managing Instagram's database. You'd be managing over 1.628 billion user's data.

Now, you need to get the users data, store about so many reels and images and you'd have millions of users using Instagram at once.

- Databases let you *query data* . That means you can ask the database, for example you can ask the database about a user's profile, their daily time spent on the app, number of followers they have. ~~didn't mean to be creepy here but yes~~
- When we have so many user using the app, databases can rapidly look up data and pick up the data and give it to you in milliseconds. *speed*
- Now, we have many people uploading images and reels, this means that many concurrent users can write to the database making it scalable. This means that data can be duplicated and spread over multiple servers since you have so so many users, allowing data to *duplicate, isolate* itself.

WITH ALL THIS HAPPENING, I DON'T THINK `.csv` IS A GOOD IDEA ANYMORE `--`

MongoDB

- It is an open source , document-oriented database.

Wait a sec, what does document-oriented mean?

It means that your data in the database is stored as *documents* and are written using *JSON* like formats.

Info

Instagram doesn't use MongoDB to store its data, it uses PostgreSQL and the main database cluster contains 12 replicas in different zones. We just will be using Instagram to explain and give examples wherever required.

Coming back our Instagram example,

- all the data would be stored as *documents* which are written in *JSON*, for example a document for each user. Sounds like a lot of document but it isn't much, you'll understand as we progress.
- A document is analogous to a table's row.
- Let me introduce you to two such documents.

```
{
  "userId": "6969",
  "userName": "Penguin",
  "fullName": "Penguin Doe",
  "email": "penguin69@example.com",
  "age": 18,
  "profilePictureUrl":
  "https://example.com/path/to/profile/picture.jpg",
  "bio": "Hello, I'm a linux enthusiast using a Mac Book Air!",
  "followersCount": 500,
  "followingCount": 200,
  "postsCount": 1000,
  "location": "Ig Bangalore"
}
```

```
{
  "userId": "5000",
  "userName": "Shreya",
  "fullName": "Shreya Gurram",
  "email": "shreya05@example.com",
  "phone": "+91 69696 96969",
  "age": 19,
  "followersCount": 100,
  "followingCount": 50,
  "postsCount": 20
}
```

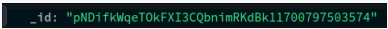
Here, I have the data of 2 users, notice how Penguin has a *bio*, *location* and a *profilePictureUrl* while Shreya doesn't have them in her document or JSON object. Shreya has *phone number* that penguin doesn't have. This means that:

1. This is a **NoSQL** database i.e. data doesn't have tables or rows and has no relation.
 2. It is stored with **Dynamic Schema** i.e. there is no fixed structure either and it hasn't been defined either and *every document can have different data* like we saw in the example above. This allows *flexibility* and *adaptability* as the user can add data to the JSON object anytime. I am not limited by the fields I can have, like I would be in an SQL table.
- Now these documents are stored in groups called *collections*
 - collections are analogous to tables in SQL. They contain a bunch of documents, or table rows. The only thing is that every row can have any number of fields.

Info

NoSQL stands for Not Only SQL

Although MongoDB isn't a relational database, here's an analogy you could use to understand and remember.

Relational DB	MongoDB	Analogy
Table	Collection	Analogous to multiple user's data (Shreya, Penguin, Millions of other users)
Row/Tuple	Document	Analogous to one user's data (Penguin's data)
Column	A property inside a document, aka field	In the above example, userName, fullName, etc are fields
Primary Key	MongoDB automatically assigns a property called <i>_id</i> to every document if you don't specify it. This is a unique ID. Every collection must have a unique ID	Here's the <i>_id</i> field MongoDB made for me for the above collections 
Table Join	Multiple documents combined together, called embedded documents	

These databases can **fetch**, **get** data really quickly because they are stored in a very similar fashion like JSON called *BSON*.
BSON stands for Binary JSON. This eliminates the Object Relational Mapping (ORM) layer that relational databases use.

Info

JSON adds some "extra" information to documents, like length of strings. This makes traversal faster and also designed to be fast to encode and decode.

All devices understand binary so why not just store every thing in binary under the hood!

Setting Up MongoDB

- You might have heard of 2 things here: MongoDB Atlas and MongoDB Compass
- Your data needs a place to live. This could be a server on the internet or your own computer. When you install MongoDB server, you create this database locally on your system. Then you connect to this database either via a *terminal* through a program called *Mongosh* (Mongo Shell), or via a GUI program called *MongoDB Compass*. It let's you navigate your data and that's why it's called a compass.
- MongoDB also offers a service to host your database for you. That means that MongoDB (the company) takes care of putting your data on a server. This service is called *MongoDB Atlas*. You can then connect to it via either MongoDB Compass or *Mongosh*.
- Atlas gives you a URL to connect to it along with a username and password.
- If you're hosting it locally, your local installation should give you a username and password and you'd connect to it via the `localhost` url.

Note

- You can use `Node.js` to manipulate MongoDB by running `npm install mongodb` in your terminal and requiring it in your script `const { MongoClient } = require('mongodb');` Node.js and writing javascript on the server is explained in the previous chapter
- You could also use a module called `mongoose` that provides *data modelling capabilities*. This just means that it offers you the ability to place constraints over what fields are allowed in a particular collection. It's kinda like placing an artificial table like restrictions.

Important

NODE JS IS NOT A LANGUAGE. JUST A RUNTIME ENVIRONMENT THAT ALLOWS YOU TO WRITE JAVASCRIPT ON A SERVER, RATHER THAN ON THE BROWSER. JAVASCRIPT IS THE LANGUAGE.

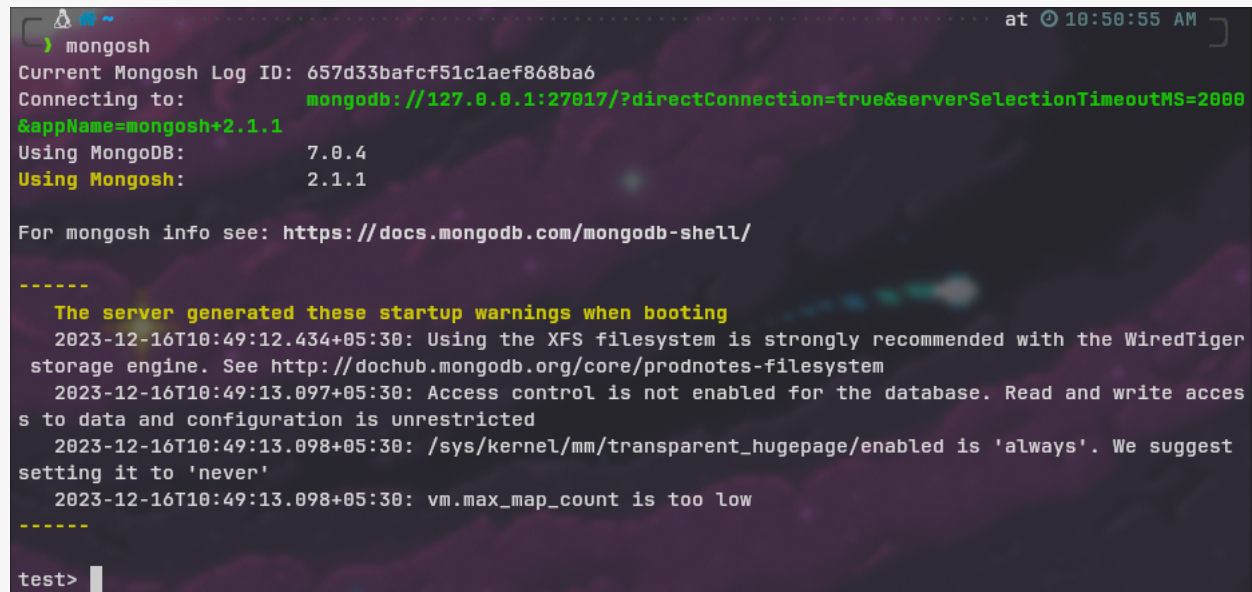
Alr, enough of theory, lets get our hands dirty.

Data Operations

- I will be explain the operations for both terminal and Node js. I'll go through the operations one by one, and show how you do it in the terminal, and in Node.js.
- This is because the syntax for these are very similar and it'd be better to do it together rather than having it in two different places
- Lets begin with making a database called **holyInstagramDB** and a collection called **usersData** which will store the data of all the users.

[info]

Here, the **terminal** means the shell given to you by mongosh. That looks something like this:



```
mongosh
Current MongoDB Log ID: 657d33bafcf51c1aef868ba6
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
&appName=mongosh+2.1.1
Using MongoDB:      7.0.4
Using Mongosh:      2.1.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-12-16T10:49:12.434+05:30: Using the XFS filesystem is strongly recommended with the WiredTiger
storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2023-12-16T10:49:13.097+05:30: Access control is not enabled for the database. Read and write acces
s to data and configuration is unrestricted
2023-12-16T10:49:13.098+05:30: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest
setting it to 'never'
2023-12-16T10:49:13.098+05:30: vm.max_map_count is too low
-----
test> |
```

Create

You can create the database and a collection using the following commands.

Terminal

- Database - a group of collections:

```
use holyInstagramDB
```

This will create a database if it doesn't exist else if it exists it will switch to the database.

- Collection - a group of documents:

```
db.createCollection("usersData")
```

This will create a collection called `usersData` if it doesn't exist in the database `holyInstagramDB`.

- A collection isn't really created in MongoDB until you insert data into it.
- You can also directly insert data into a collection by just naming it.
- Then what's the use of `.createCollection()`?
- It allows you to make rules for your collection, create capped collections, etc. It also pre-allocates data to make further writes faster.

Node.js

- Database - a group of collections:

JS

```
const { MongoClient } = require('mongodb');
const uri = "mongodb://localhost:27017"; // I'm hosting the
mongodb instance locally, so my url is localhost. If you were
hosting it on atlas, this would be given to you by atlas.
const client = new MongoClient(uri);
client.connect().then(() => {
  console.log("Connected successfully to server");
  const database = client.db("holyInstagramDB");
})
```

This will create a database if it doesn't exist or switch to the database if it exists.

- Collection - a group of documents:

```
const { MongoClient } = require('mongodb');
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri); // create a new Mongo
Client instance. MongoClient is a class provided by mongodb
client.connect().then(() => {
  console.log("Connected successfully to server");
  const database = client.db("holyInstagramDB");
  const users = database.collection("usersData"); // this is
  our collection. We can later refer to this collection by just
  doing users.something
})
```

This will create a collection called `usersData` in the database `holyInstagramDB` if it doesn't exist

`client.connect()` returns a promise and we use `.then()` to handle the promise. Now, instead of using multiple `.then()` which is visually hard to follow. We can use `async` and `await` to make it easier to follow.

`async` is used to declare an asynchronous function. An asynchronous function when called will be asynchronous. You can use an `await` statement only inside an `async` function. When you `await` inside an async function, it waits for that particular statement to complete. The `await` keyword is used to pause the execution of the function until the Promise is resolved or rejected.

```
const { MongoClient } = require('mongodb');
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);
async function connect_to_db () => {
  await client.connect();
  const database = client.db("holyInstagramDB");
  const users = database.collection("usersData");
  return [database, users];
}
```

So here we have a function `connect_to_db` which is an asynchronous function. We use `await` inside it to wait for the connection to complete before we get the database and collection objects.

Insert

You can insert one or many data into the collection using the following commands.

Terminal

1. Insert one data into the collection:

```
SHELL
db.usersData.insertOne({
  "userId": "133927",
  "userName": "Jane",
  "fullName": "Jane Doe",
  "email": "janedoeeee@gmail.com",
  "age": 27,
  "bio": "I like eating Black Forest Cakes!",
  "followersCount": 100,
  "followingCount": 50,
  "postsCount": 20
});
```

Here, we are inserting one data or one JSON object into the collection `usersData` in the database `holyInstagramDB`.

So, now we have 3 users in the collection and a million more to go. I cant keep inserting one by one, so lets insert many at once.

2. Insert many data into the collection:


```
# this is of the syntax db.collection.insertMany([<array of
documents>])
db.usersData.insertMany([
  {
    "userId": "133927",
    "userName": "Jane",
    "fullName": "Jane Doe",
    "email": "janedoeeee@gmail.com",
    "age": 27,
    "bio": "I like eating Black Forest Cakes!",
    "followersCount": 100,
    "followingCount": 50,
    "postsCount": 20
  },
  {
    "userId": "493627",
    "userName": "John",
    "fullName": "John Doe",
    "email": "iamjohndoeeee@outlook.com",
    "age": 18,
    "profilePictureUrl":
"https://example.com/path/to/profile/picture.jpg",
    "bio": "Hello, I hate eating Black Forest Cakes!",
    "followersCount": 500,
    "followingCount": 200,
    "postsCount": 1000,
    "location": "Mars City, Mars"
  }
]);
```

Here, we are inserting many data or many JSON objects into the collection `usersData` in the database `hollyInstagramDB`.

Node.js

1. Insert one data into the collection:

```
// syntax
// <collectionObject>.insertOne(<document>)
// this code snippet goes inside the .then() callback function.
// OR if you're using async await, after calling the function to
// connect to db.
// you basically need to have access to the collection object
// we just created at collection = db.collection("userData");
```

```
users.insertOne({
  "userId": "133927",
  "userName": "Jane",
  "fullName": "Jane Doe",
  "email": "janedoeeee@gmail.com",
  "age": 27,
  "bio": "I like eating Black Forest Cakes!",
  "followersCount": 100,
  "followingCount": 50,
  "postsCount": 20
});
```

2. Insert many data into the collection:

```
// syntax
// collectionObject.insertMany(<array of documents>))

users.insertMany([
  {
    "userId": "133927",
    "userName": "Jane",
    "fullName": "Jane Doe",
    "email": "janedoeeee@gmail.com",
    "age": 27,
    "bio": "I like eating Black Forest Cakes!",
    "followersCount": 100,
    "followingCount": 50,
    "postsCount": 20
  },
  {
    "userId": "493627",
    "userName": "John",
    "fullName": "John Doe",
    "email": "iamjohndoeeee@outlook.com",
    "age": 18,
    "profilePictureUrl":
    "https://example.com/path/to/profile/picture.jpg",
    "bio": "Hello, I hate eating Black Forest Cakes!",
    "followersCount": 500,
    "followingCount": 200,
    "postsCount": 1000,
    "location": "Mars City, Mars"
  }
]);
```

Read

You read multiple collections or one collection using the following commands.

Terminal

1. Read all the collections from the database:

```
db.usersData.find({})
```

This will read all the data from the collection `usersData` in the database `holyInstagramDB`.

2. Read a specific collection from the database:

SHELL

```
db.usersData.find({"userName": "Penguin"})  
# this is of the syntax db.collection.find(query, projection)
```

This will read all the data from the collection `usersData` in the database `holyInstagramDB` where the `userName` is `Penguin`.

Node.js

1. Read all the collections from the database:

JS

```
const { MongoClient } = require('mongodb');  
const uri = "mongodb://localhost:27017";  
const client = new MongoClient(uri);  
client.connect().then(() => {  
  console.log("Connected successfully to server");  
  const database = client.db("holyInstagramDB");  
  const users = database.collection("usersData");  
  users.find() // this finds all the data in the collection  
})
```

This will read all the data from the collection `usersData` in the database `holyInstagramDB`.

2. Read a specific collection from the database:

```
users.find({"userName": "Penguin"});  
// this finds all the data in the collection where the userName  
is Penguin  
// this is of the syntax users.find(query, projection)  
// the projection defines what fields are shown in your output.  
You can pass an object like {fullName: 1, _id: 0} to show only  
fullName.
```

Update

Terminal

1. Update one data in the collection:

Now, I don't like the fact that Penguin has a Mac Book Air, so I want to update his bio. Lets do that!!

```
# this is of the syntax db.collection.updateOne(filter, update,  
options)  
db.usersData.updateOne({"userName": "Penguin"}, {$set: {"bio":  
"Hello, I'm a linux enthusiast and I like cupcakes!"}})
```

Here, we are updating one data or one JSON object in the collection **usersData** in the database **holyInstagramDB** where the **userName** is **Penguin**.

\$set is a MongoDB operator that sets the value of a field in a document. There are other operators like **\$in**, **\$gt** (greater than), etc.

2. Update many data in the collection:

Now, I don't like John Doe and Jane Doe. I want to block them. I won't update them individually since I can update the same value for both of them at once using the following command.

```
# this is of the syntax db.collection.updateMany(filter,
update, options)
db.usersData.updateMany({"userName": {$in: ["John", "Jane"]}},
{$set: {"blocked": true}})
```

Here, we are updating many data or many JSON objects in the collection **usersData** in the database **holyInstagramDB** where the **userName** is **John** or **Jane**.

Node.js

1. Update one data in the collection:

Lets see how we can update Penguin's bio using Node.js

JS

```
// this is of the syntax users.updateOne(filter, update,
options)
users.updateOne({"userName": "Penguin"}, {$set: {"bio": "Hello,
I'm a linux enthusiast and I like cupcakes!"}});
```

2. Update many data in the collection:

Lets block John and Jane using Node.js

JS

```
// this is of the syntax users.updateMany(filter, update,
options)
users.updateMany({"userName": {$in: ["John", "Jane"]}}, {$set:
{"blocked": true}});
```

FUN STUFF: Update Or Insert? Introducing **upsert**

Now, lets say I want to update Samira's bio but I don't know if she exists in the database or not.

Upsert is a combination of update and insert. If the document exists, it updates it else it inserts it.

SHELL

```
# this is of the syntax db.collection.updateOne(filter, update, options)
db.usersData.updateOne({"userName": "Samira"}, {$set: {"bio": "Money money money aha!"}, {"followersCount": "3000"}, {"followingCount": "2000"}}, {upsert: true})
```

JS

```
// this is of the syntax users.updateOne(filter, update, options)
users.updateOne({"userName": "Samira"}, {$set: {"bio": "Money money money aha!"}, {"followersCount": "3000"}, {"followingCount": "2000"}}, {upsert: true});
```

Note

upsert is a boolean value that is false by default. If set to true, creates a new document when no document matches the query criteria.

Delete

Terminal

1. Delete one data in the collection:

Imma be evil now, I want to delete Samira's data from the database.

SHELL

```
# this is of the syntax db.collection.deleteOne(filter, options)
db.usersData.deleteOne({"userName": "Samira"})
```

2. Delete many data in the collection:

You know what imma be more evil. I want to delete all the blocked users from the database i.e Jane's and John's data.

SHELL

```
# this is of the syntax db.collection.deleteMany(filter, options)
db.usersData.deleteMany({"blocked": true})
```

This looks for the users who have the field `blocked` set to `true` and deletes them.

Node.js

1. Delete one data in the collection:

Deleting Samira's data!!!

JS

```
// this is of the syntax users.deleteOne(filter, options)
users.deleteOne({"userName": "Samira"});
```

2. Delete many data in the collection:

Deleting all the blocked user's data from the database!

JS

```
// this is of the syntax users.deleteMany(filter, options)
users.deleteMany({"blocked": true});
```

Drop

Till now, only you and I knew how I could access our *Holy Instagram's Database* and make changes so conveniently. And this was ultra ultra creepy. Now, everyone knows our secret and I'm forced to delete the database.

I won't sit and delete one collection at a time, I'll just drop the entire database.

SHELL

```
db.dropDatabase()
```

JS

```
database.dropDatabase();
```


This will drop the database `holyInstagramDB` and all the collections in it.

Note

Be careful when you use this command. It is irreversible and you cannot undo it.

Here's the end of our Holy Instagram Database 🙄 (not really)

Node.js Recap

Honestly Node.js feels like the elephant in the room that we haven't talked about. Node.js is an environment that lets you run JavaScript code outside of a web browser. Ashten Period.

Now, how does Node.js actually work? When does it know what to do when you run `node filename.js` in your terminal?

Node.js is a single threaded app. This is explained in detail in unit 3 and this is just a recap.

Node.js does not do multiple things at the same time, it is single threaded. BUT, we Node.js kinda cheats around this by implementing an event-response kind of architecture. Imagine you want to send a database request. It sends the request and then does other things. Then it waits for the request to emit an event. Then it reacts accordingly.

Event Emitter Class

Woah woah woah wait a sec WHAT ARE EVENTS?!

Think of an event as a signal that something has happened.

Now this something that has happened can be an user interaction, a system notification, or anything that your program needs to respond to.

- Lets get back to Instagram, you find a friend and you want to follow them so you click on the follow request button. Now this is an event, you have sent a signal to Instagram's server and now it needs to do something with the signal. On getting the signal Instagram's server will respond to your signal and send you a notification that a follow request has been sent and your friend would get a follow request from you.
- Another example would be when you click on the submit button after filling a form. This is an event, you have sent a signal to the program and now it needs to respond. It would respond by save all the data that you've filled in the form and save it in the backend database.

So *event emitters* are objects that emit named events. And there is a *listener* function that listens to the event and responds to it.

Event Emitter Class is a class that has methods to emit these events and listen to them. It is part of the events module. Lemme explain with an example

Note

EventEmitter is a class and not an object. So, you need to create an instance of the class to use it.

Lets make a **onclick** event emitter class that emits an event when the user clicks on a button.

Here's the HTML code for the button

HTML

```
<button id="submitButton">Submit</button>
```

Here's the JavaScript code for the event emitter class

JS

```
const EventEmitter = require('events'); // importing event
module
const emitter = new EventEmitter(); // creating an instance of
the event emitter class

const button = document.getElementById('submitButton'); //
getting the submit button element

button.addEventListener('click', () => { // adding a click
event listener to the button which console logs the following
message when clicked!
  console.log("button has been clicked. going to save data and
gonna sell it hehe!");
});
```

Notice how I used `addEventListener`, instead I could also use `.on` to listen to the event of submit button being clicked.

JS

```
button.on('click', () => { // adding a click event listener to
  the button which console logs the following message when
  clicked!
  console.log("button has been clicked. going to save data and
  gonna sell it hehe!");
});
```

Info

The generalised syntax is `.eventName('event', listener)` where listener is a function that is called when the event is emitted.

More examples from the event emitter class are:

- `.once(event, listener)` : this listens to the event only once and then removes the listener
- `.removeListener(event, listener)` : this removes the listener for the event
- `.removeAllListeners([event])` : this removes all the listeners for the event
- `.setMaxListeners(n)` : this sets the maximum number of listeners for the event
- `.listeners(event)` : this returns an array of listeners for the event
- `.emit(event, [arg1], [arg2], [...])` : this emits the event with the arguments arg1, arg2, etc.

Single Page Application and React Routing

- We are back to React. *why...* PES people can't decide one order and teach.
- First they do react in the previous unit, then Node.js, then Mongo, then again React??

Single Page Application (SPA) is a web application that loads a single HTML page once and then the contents of the pages are dynamically changed as per the user interaction with the web app. This means that the page doesn't reload when the user interacts with the web app.

This saves a lot of time and bandwidth since the page doesn't reload every time the user interacts with the web app.

Previous `<a>` tags were used to navigate between pages. So, when a link is clicked the server would get a request and the server would respond. This is how the web worked before SPAs came into the picture.

Now this feels slow since the server has to respond to every request and the page has to reload every time the user interacts with the web app.

Guess whatt **PESUACADEMY IS A SPA!** (but a very poorly built one, I know you would have gotten a whole rant about PES from Anurag in the previous unit so I'll spare you from that!)

The first screenshot shows the 'My Courses' page. The URL is `pesuacademy.com/Academy/s/studentProfilePESU`. The page displays a table of courses for Semester 3:

Course Code	Course Title	Course Type	Status	Action
UE22CS223A	Unix Shell Programming	EC	Enrolled	
UE22CS241A	Statistics for Data Science	CC	Enrolled	
UE22CS242A	Web Technologies	CC	Enrolled	
UE22CS243A	Automata Formal Languages and Logic	CC	Enrolled	
UE22CS251A	Digital Design and Computer Organization	CC	Enrolled	
UE22CS252A	Data Structures and Its Applications	CC	Enrolled	

The second screenshot shows the 'Student Grievance Redressal System' page. The URL remains `pesuacademy.com/Academy/s/studentProfilePESU`. The page displays a table with columns: Sl.No, Ticket Number, Created Date, Problem Type, Closed date, Status, and Actions. The table is empty, showing 'No data available in table'.

The third screenshot shows the 'My Courses > UE22CS242A: Web Technologies' page. The URL remains `pesuacademy.com/Academy/s/studentProfilePESU`. The page displays a table with columns: Units, Introduction, Objectives, Outcomes, Syllabus, Outline, and References. The table is empty, showing 'No data available in table'.

Notice how the URL doesn't change when I click on *My Course*, *Student Grievances Redressal System* and when I navigate to Web Technologies and click on Unit 4. This is because the page doesn't reload when I click on these links.

How we do this in react? We use React Router!

With *React Router* we can achieve client side routing(CSR) in our react app.

Info

Client Side Rendering vs Server Side Rendering

Server Side Rendering is when the server renders the HTML page and sends it to the client. On every user interaction, the server needs to send the fully-rendered HTML of the new data to the user. This is time taking and slow and inefficient and takes up power as the server.

Client Side Rendering is when the server sends the HTML page with the JavaScript files. The JavaScript files are then executed on the client side (all browsers have their own JIT Compiler) and the page is rendered on the client side. On every user interaction, the server doesn't need to send the fully-rendered HTML of the new data to the user.

We gotta import a couple things like `BrowserRouter`, `Route`, `Switch` and `Link` from `react-router-dom` to use React Router.

JS

```
import { BrowserRouter, Route, Switch, Link } from 'react-router-dom';
```

- `<BrowserRouter>` is a component that would wrap all the components and pages that we want to render in our app.
- `<Route>` is a component that renders a component or page when the path matches the URL.
- `<Link>` is a component that is used to navigate between pages. It is similar to `<a>` tag but it doesn't reload the page when clicked.

I know it sounds really complex, don't freak out. Let's get back to our Instagram example.

So we have a bunch of pages in our Instagram app like *Home*, *Profile*, *Explore* and *Settings*. We need to render these pages when the user clicks on them.

We have an `App.js` file that renders the `Home` page when the user opens the app. From here, we need to route to other pages.

```

import React from 'react';
import { BrowserRouter, Route, Switch, Link } from 'react-router-dom'; // importing the required

// importing all the required page
import Home from './pages/Home';
import Profile from './pages/Profile';
import Explore from './pages/Explore';
import Settings from './pages/Settings';
import Login from './pages/Login';

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Link to ="/">Home</Link>
        <Link to ="/profile">Profile</Link>
        <Link to ="/explore">Explore</Link>
        <Route path="/" component={Home} />
        <Route path="/profile" component={Profile} />
        <Route path="/explore" component={Explore} />
        <Route path="/settings" component={Settings} />
      </Switch>
    </BrowserRouter>
  );
}

export default App;

```

You might be confused about why we have `<Link>` and `<Route>` components. Let me explain.

`<Link>` is used to create links basically a replacement of `<a>` tag.

Now, `<Route>` has a `path` attribute which is the URL path and the current location's path matches the URL, then the required component will be rendered. For Example, when the URL path matches "/", the `Home` component will be displayed or rendered.

But wait, what if the user enters a URL that doesn't exist in our app?

We need to handle that too. We can use the `<Switch>` component to handle that. `<Switch>` renders the first child `<Route>`. In our case it would render the `Home` component.

Styled Link

- You can use a Styled Links to wrap your components in some CSS.
- You would never be using this in real life, it's the old way of doing things.
- Here's how you do it:

```
// App.js
import {StyledLink} from './styles.js';
<Router>
  <StyledLink to="/">Home</StyledLink>
  // other routes here
</Router>

// styles.js
import styled from 'styled-components' // a module to make
styled components

import {NavLink} from 'react-router-dom' // link in the
navigation bar at the top

// we make it a default export so that it can be imported in
App.js
export default const StyledLink = styled(NavLink)`
  padding: 10px;
  // some other css here
`;
```

Aightyy that's a lot of info, Lets get to something simple and easy. Presenting Web Services!

Web Services

These are services that are available over the internet. This services can be shared by many different applications. These services are built on top of the HTTP protocol.

Web services provide a standardized method for exchanging data between different applications or systems over the internet which follow standard internet protocols.

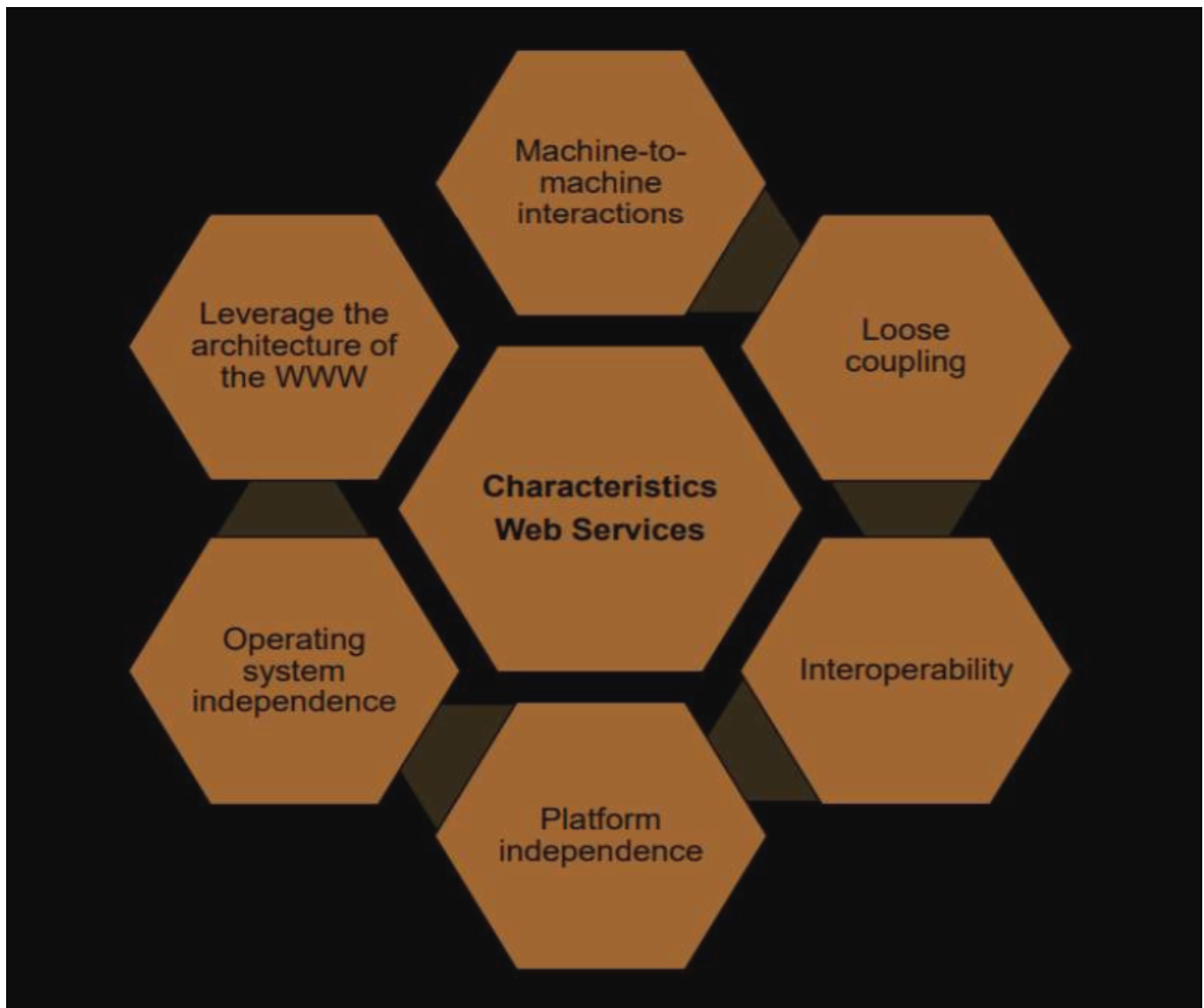
For example, you can use the weather API to the weather data of your location and display it on your app. The weather API is a web service that is available over the internet and can be used by many different applications.

From this example, some of the important features of web services are:

- It is designed for application to application since you can use the API to render weather data however you want.
- It is interoperable i.e. it can be used by any application that can understand the request and response format of the API.
- It is platform independent i.e. it can be used by any application irrespective of the platform it is running on.
- It is language independent i.e. it can be used by any application irrespective of the language it is written in.

This lets us *integrate applications* making it *flexible, adaptable* and they are *inexpensive*. One of the type of web services is *RESTful web services*. It treats data as resources.

This image is from the slides btw, viewer discretion.



REST API

REST API stands for *REpresentational State Transfer Application Programming Interface*. It is an architectural style for building web services. This architecture is all about CSR (Client Side Rendering) and it is stateless i.e. the server doesn't store any state about the client session or doesn't store any client data on the server side. This makes the server less complex and more scalable.

Every resource in a REST API is a unique URL. For example, the URL for a user's profile would be <https://instagram.com/penguin> and the URL for a user's post would be <https://instagram.com/penguin/posts/1234>.

The communication between the client and server in a REST API is done through HTTP requests. Let's get back to our Instagram example.

- I wanna see Jane's profile, REST API would use a **GET** request to retrieve Jane's record from the database.
- I wanna update my bio, REST API would use a **PUT** request to update my bio in the database.
- I wanna delete my account, REST API would use a **DELETE** request to delete my record from the database.
- You are creating a new spam account, REST API would use a **POST** request to create a new record in the database.

Remember that question in ISA-2 asking about RESTful DESIGN CONSTRAINTS? Here's the answer to it!

1. **Client-Server Architecture** : The client and server are independent of each other and can be developed and deployed separately. This makes the system more scalable and flexible.
2. **Stateless** : The server doesn't store any state about the client session or doesn't store any client data on the server side. This makes the server less complex and more scalable.
3. **Cacheable** : The client side must be cacheable which means that it should be able to store copies of the frequently accessed data, making it efficient. Think of this as the cached memory in your laptop.
4. **Uniform Interface** : This means that the client and server must be able to understand each other's requests and responses which means we'd need a uniform interface.
5. **Layered System** : This means that the client and server can be deployed on different machines and can be scaled independently.
6. **Code on Demand** : This means that the server can extend the functionality of the client by sending code to the client.

To satisfy all the above constraints we need have the following considerations:

- **Resource Identification** : Every resource in a REST API is an unique URL. For example, the URL for a user's profile would be <https://instagram.com/penguin> and for Penguin's posts would be <https://instagram.com/penguin/posts/>.
- **Resource Manipulation** : The client can manipulate or change the resource by sending a request to the server and the server needs to give an appropriate HTTP status code. For example, the client can send a request to the server to update Penguin's bio.
- **Security and Authentication** : The server needs to authenticate the client before sending the response. For example, if Penguin wants to change his bio, we'd need to authenticate him before he could change his bio.
- **Designing URI** : The URI should be simple like <https://instagram.com/penguin> and not like something like <https://instagram.com/696969penguiningmail> which

makes no sense.

Recap to HTTP METHODS

HTTP Method	Example	Explanation
GET	<code>GET /users</code>	Retrieves a list of users
POST	<code>POST /users</code>	Creates a new user
PUT	<code>PUT /users/1</code>	Updates user with id 1
DELETE	<code>DELETE /users/1</code>	Deletes user with id 1

Recap to HTTP STATUS CODES

HTTP Status Code	Explanation
200 OK	The request was successful
201 Created	The request was successful and a resource was created
400 Bad Request	The request was invalid
401 Unauthorized	The request did not include an authentication token or the authentication token was expired
403 Forbidden	The client did not have permission to access the requested resource
404 Not Found	The requested resource was not found
500	Internal Server Error

Express JS

Express JS is a web application framework for Node.js. It is used to build web applications and APIs. There are many modules available on [npm](#) which can be directly used in Express.

Why should we use Express JS? ~~bc its in syllabus~~

1. You can develop web apps and API's quickly.
2. It is scalable because you can use it to create mobile apps, web apps, single page apps, multi page apps.
3. It has various templating engines like Pug, EJS, Handlebars, etc. ~~this is lil bit useless yet it's there in our syllabus!~~
4. It has a large community and is open source.

5. Can easily be integrated with MongoDB. (We studied this all the way at the beginning of this unit, remember?)
6. *Middleware*, This is a fun thing, I would be explaining this in detail soon.

Setting Up Express JS

You can install Express JS using the following command in your terminal in your Node.js application.

SHELL

```
npm install express
```

Routing

Routing in simple words means to give the client what they're after for from the server. For example if the client wants to see Penguin's profile, the server should give the client Penguin's profile and not Shreya's profile. This is routing.

This is really important in web apps cause you don't want to end up giving the client wrong data.

Express JS has a built in router which is used to define routes in your app. It is a middleware and can be used to perform multiple tasks like validation, authentication, etc. This route has some specifications:

1. HTTP Method : GET, POST, PUT, DELETE, etc.
2. Path : Path specification that matches the URI. For example, `/users` or `/users/:id`
3. Route Handler : Function that is executed when the route is matched.

Let's start with setting routes in our Express JS app. We have an `app.js` file which is the entry point of our app. We need to import the `express` module and create an instance of the `express` class.

JS

```
const express = require('express'); // importing express module
const app = express(); // creating an instance of the express
class
```

Now, we need to define a route handler for the root path `/` which is the home page of our app. We can do this using the `app.get()` method.

```
app.get('/', (req, res) => {
  res.send('Hello World!');// sending a response to the client
  console.log("Hallo, handling the get request!");
});
```

Info

Note that this is entirely different from React Router. That's a whole different topic from frontend that PES randomly inserted in between.

The route handler callback function takes two parameter, **req** and **res**.

req is the request object that contains information about the HTTP request that raised the event. Here, it is the GET request.

res is the response object that contains information about the HTTP response that the app sends when it receives an HTTP request. Here, it is the response to the GET request, which is **Hello World!** and console logs **Hello, handling the get request!**.

Let's see another example of a route handler for the **/users** path.

```
app.get('/users', (req, res) => {
  res.send('Hello Users!');// sending a response to the client
  console.log("Hallo, handling the get request at /users!");
});
```

When the client sends a GET request to the **/users** path, The server first matches the URL to the appropriate route handler. Then the server fires the callback function associated with that route, that responds with **Hello Users!** and console logs **Hallo, handling the get request at /users!**.

Info

The path can take Regex and doesnt have to be specific. For example, **/users/12*** would match **/users/123**, **/users/1234**, **/users/12345**, etc.

Note

If you don't want to specify the HTTP method, you can use `app.all()` method which handles all the HTTP methods.

Let's see an example of `app.post()` method which handles the POST request.

JS

```
app.post('/users', (req, res) => {  
  res.send('Hello Users!');// sending a response to the client  
  console.log("Hallo, handling the post request at /users!");  
});
```

Here, the POST request is handled at the `/users` path.

Your entire program will look something like this:

```
const express = require('express'); // importing express module
const app = express(); // creating an instance of the express
class
```

```
app.get('/users', (req, res) => {
  res.send('Hello Users!');// sending a response to the client
  console.log("Hallo, handling the get request at /users!");
});
```

```
app.post('/users_post', (req, res) => {
  res.send('Hello Users!');// sending a response to the client
  console.log("Hallo, handling the post request at
/users_post!");
});
```

```
// after you're done defining the routes, you have to call
app.listen(). It takes in a port number and a callback function
that is called when the app starts listening. Now you can make
requests to http://localhost:3000/users,
http://localhost:3000/users_post etc
app.listen(3000, () => {
  console.log("App is listening on port 3000");
})
```

Route Parameters `req.params` & Dynamic Routes

Route parameters are used to extract the values from the URL.

For example, we want to get a particular follower of a user. The URL would be `/users/:id/followers/:followerId`.

Here, `:id` and `:followerId` are route parameters. We can access these route parameters using `req.params`.

For example, below, we send a request to the following url:

`/users/Penguin/followers/1234`

```
app.get('/users/:id/followers/:followerId', (req, res) => { //
  notice how the user id and follower id are route parameters and
  are not explicitly specified
  res.send(`Hello ${req.params.id}!`); // sending a response to
  the client, this would send Hello Penguin! if the user id is
  Penguin
  console.log(`Hallo, handling the get request at
  /users/${req.params.id}/followers/${req.params.followerId}!`);
  // this would console log Hallo, handling the get request at
  /users/Penguin/followers/1234!
});
```

`${req.params}` basically means request parameters. So, `${req.params.id}` would give the user id and `${req.params.followerId}` would give the follower id.

Middleware

Middleware is a function that has access to the request and response object.

They are a chain of functions that are executed one after the other when a request is made to your express.js server.

Middleware functions can be used to perform the following tasks:

1. Execute any code.
2. Make changes to the request and the response objects.
3. End the request-response cycle.
4. Call the next middleware function in the stack. `next()`

For example if the client wants to update their bio, the server needs to authenticate the client before updating the bio. This is done using *middleware*.

So we have a *PUT* request to update the bio, the middleware authenticates the client and then the bio is updated. See how the middleware came in between the request(change in bio) and the response(updated bio).

Here's the syntax for middleware functions:


```
app.use((req, res, next) => {  
  console.log("This is a middleware function!");  
  next();  
});
```

Here, `next()` is used to call the next middleware function in the chain of middleware functions. If `next()` is not called, the request will be left hanging.

Info

An Express application can use the following types of middleware:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

General example of middleware:

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

app.use((req, res, next) => {
  console.log("This is a middleware function!");
  // start authentication, here we can put code to authenticate
  the user!
  next();
});

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

app.get('/users', (req, res) => {
  res.send('Hello Users!');
});

app.listen(3000, () => {
  console.log("Server is listening on port 3000!");
});
```

In this case, the middleware function logs a message to the console and then calls the `next()` function.

Info

The order in the chain of middleware functions called is the same as how we add them in our program. So, for the above example, the middleware function that console logs is called first for all routes, then the `app.get()` is used at the last.

Application Level Middleware

Application level middleware is used to perform operations that are common to all routes, like authentication, logging and so on. In our example, our Instagram app would have an authentication middleware that would authenticate the user before updating their bio.

Authentication is out of scope of this course and would require you to know a whole bunch of other things. So, we'd go for a simpler example like displaying the current date.

JS

```
const express = require('express');
const app = express();
app.use((req, res, next) => {
  console.log("This is a middleware function!");
  req.requestTime = Date.now();
  console.log("Today's date is " + req.requestTime);
  next();
});
```

Here, we are using the `Date.now()` method to get the current date and time and storing it in the `req` object. We can access this date and time using `req.requestTime`.

Now, let's see which HTTP Method would Express JS use to get the user id from the URL of our Instagram app.

JS

```
app.use('/users/:id', (req, res, next) => {
  console.log("This is a middleware function!");
  console.log("The HTTP Method used is " + req.method);
  next();
});
```

Here, we are using the `req.method` to get the HTTP Method used by the client.

Router Level Middleware

Router level middleware is used to perform the same thing as application level middleware but it uses the router object.

- These are usually used when you want the middleware to work on a specific route.
- Although application level middleware can do this too, it's recommended to use router level middleware to achieve this.

```
const express = require('express');
const app = express();
const router = express.Router();
router.use("/users", (req, res, next) => { // would only run at
/users route. If you skip that parameter, it'll run on all
routes.
  console.log("This is a middleware function!");
  console.log("Today's date is " + Date.now());
  console.log("The HTTP Method used is " + req.method);
  console.log("Using Router Level Middleware!");
  next();
});
```

Notice how we aren't using `app.use()` but `router.use()`.

Error Handling Middleware

Error handling middleware is used to handle errors that occur during the execution of the request-response cycle. This only runs if any errors arise. For example, if the client sends a request to update their bio but the database isn't responding, the server would send an error message to the client.

It is very similar to the above two middleware but has an extra parameter `err`.

```
const express = require('express');
const app = express();
app.use((err, req, res, next) => { // Notice how err is the
first parameter
  console.log("This is a middleware function!");
  console.log("Using Error Handling Middleware! If you see
this, something went wrong!" + err);
  next();
});
```

Here, we are using the `err` parameter to get the error message.

Built-in Middleware

Express JS has a bunch of built-in middleware that can be used to perform various tasks. Some of them are:

1. `express.json()` : This is used to parse incoming requests with JSON payloads.
2. `express.urlencoded()` : This is used to parse incoming requests with URL payloads.
3. `express.static()` : This is used to serve static files like Instagram's logo, profile pictures.

JSON Payload

Written in JSON format, usually used to send data to the server. Like, when you fill a form and hit the submit, your data would be sent to the server in JSON format as a JSON payload.

URL Payload

URL payload is sent as part of the URL and is typically used for GET requests, where data is being retrieved from a server. It is a key-value pair.

```
http://example.com/api/resource?param1=value1&param2=value2
```

Here, `param1=value1` and `param2=value2` are the URL payloads.

Third Party Middleware

These middlewares aren't built in Express JS and need to be installed using `npm`. For example, `body-parser` is a third party middleware that is used to parse incoming form data. The default behaviour of HTML forms isn't to send data in the form of JSON. It uses its own format. We use `body-parsers` to parse this into json.

We can install `body-parser` using the following command in our terminal.

```
npm install body-parser
```

SHELL

Now, we need to import `body-parser` in our `app.js` file.

```
const express = require('express');  
const bodyParser = require('body-parser');  
app.use(bodyParser.json());
```

For making a JSON parser, we need to use `bodyParser.json()` and for making a URL parser, we need to use `bodyParser.urlencoded()`.

Template Engines

- Template engines are used when you want to make some kind of quick frontend when you're a backend developer
Template engines are used to create an HTML template with dynamic data.
So, what this does is, at runtime, the engine converts your template to HTML and sends it.

Now, Express JS doesn't have a built in template engine. There are many template engines available on `npm` like Pug, EJS, Handlebars, etc. We'd be using Pug for this example.

Form Data

Pug

Pug used to be called as Jade, I have no idea why they changed it.
Do an `npm install pug` in your terminal to install Pug.

Let's make a simple form using Pug.

This is `form.pug` file's contents:

- This is the template to generate your form.

```

html
  head
    title Form
  body
    form(action="/form", method="POST")
      div // div tag doesnt need to be closed, Pug would
automatically close it like how Javascript puts a semicolon at
the end of every line
        label(for="name") Name:
        input(type="text", name="name")
      br
      div
        label(for="email") Email:
        input(type="email", name="email")
      br
      div
        label(for="password") Password:
        input(type="password", name="password")
      br
      div
        label(for="ProfilePicture") Profile Picture:
        input(type="file", name="profilePicture")
      div
        input(type="submit", value="Submit")

```

Notice how this is very similar to HTML.

Now, we need to import Pug in our `app.js` file and set the view engine to Pug.

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

var bodyParser = require('body-parser');

// first we need to read the form data.
app.get('/form', (req, res) => {
  res.render('form');
});

app.set('view engine', 'pug'); // setting the view engine to pug
app.set('views', './views'); // setting the views directory

// now we need to parse the form data.
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// now we need to send the form data to the server.
app.post('/form', (req, res) => {
  console.log(req.body);
  res.send("Form Submitted!");
});

app.listen(3000, () => {
  console.log("Server is listening on port 3000!");
});
```

`req.body` contains the form data. We can access the form data using `req.body.name`, `req.body.email`, `req.body.password`.

Now, we need to create a `views` folder in our project directory and put the `form.pug` file in it.

If you run the server and go to `localhost:3000/form`, you'd see the form that we created using Pug.

Now, if you fill the form and click on submit, you'd see the form data in the terminal.

File Upload

File upload is a lil hard to do in Express JS due to its asynchronous nature.

There are many middleware like `multer`, `connect` that can be used to upload files in Express JS.

Start with installing `file-upload` using the following command in your terminal.

SHELL

```
npm install file-upload
```

When you upload a file, it would be accessible in `req.files`. It is very similar to `req.body` which contains the form data.

Now, we need to import `file-upload` in our `app.js` file.

I'm going to upload a file called `mydp.png` to that input field.

In my Express server request, I can access this file using `req.files.profilePicture`.

JS

```
app.post('/upload', (req, res) => {  
  console.log(req.files.profilePicture); // the uploaded file  
  would be console logged  
  console.log(req.files.profilePicture.name); // the name of  
  the uploaded file would be console logged  
  console.log(req.files.profilePicture.mimetype); // the  
  mimetype of the uploaded file would be console logged  
  console.log(req.files.profilePicture.data); // the data of  
  the uploaded file, this is a buffer representaiton of your data  
  i.e your data in form of an array of bits.  
  console.log(req.files.profilePicture.size); // the size of  
  the uploaded file would be console logged  
  res.send("Form Submitted!");  
});
```

Info

Create a folder called `uploads` in your project directory. This is where the uploaded

files would be stored.

If you write some HTML in your `index.html` file for the file upload button. Here's some simple HTML for it:

HTML

```
<h1>File Upload</h1>
<form action="/upload" method="POST" enctype="multipart/form-
data">
  <input type="file" name="profilePicture">
  <input type="submit" value="Submit">
</form>
```

Now, if you run the server and go to `localhost:3000/upload`, you'd see the file upload button. Now, if you upload a file and click on submit, you'd see the file data in the terminal.

Let's see some code for `index.js` file that would upload the file to the `uploads` folder.

```
const express = require('express');
const app = express();
app.use(upload());

app.post('/upload', (req, res) => {
  console.log(req.files.profilePicture); // the uploaded file
  would be console logged
  console.log(req.files.profilePicture.name); // the name of
  the uploaded file would be console logged
  console.log(req.files.profilePicture.mimetype); // the
  mimetype of the uploaded file would be console logged
  console.log(req.files.profilePicture.data); // the data of
  the uploaded file, this is a buffer representaiton of your data
  i.e your data in form of an array of bits.
  console.log(req.files.profilePicture.size); // the size of
  the uploaded file would be console logged

  // moving file to the uploads folder
  var file = req.files.profilePicture;
  var fileName = req.files.profilePicture.name;
  file.mv(`./uploads/${fileName}`, (err) => {
    if(err) {
      console.log(err);
      res.send("Error Occured!");
    }
    res.send("File Uploaded!");
  });
  res.send("Form Submitted!");
});

app.listen(3000, () => {
  console.log("Server is listening on port 3000!");
});
```

Here, we are using the `mv()` method to move the file to the `uploads` folder.

With this we are done with Unit 4! 🎉

Made with ♥ by Shreya Gurram, Reviewed by Anurag Rao

© Shreya Gurram 2023