What is NodeJS

↳ prerequisite to learn

- Only system
- Your Timing + Dedication

⇒ I will teach you Basics of Javascript as well.

-> "Just start and see the Magical Result."

What is NodeJS

-> Initially, JS was used in frontend only, to enhance UI and and interaction on website. (i.e Button, dynamic colour, popup)

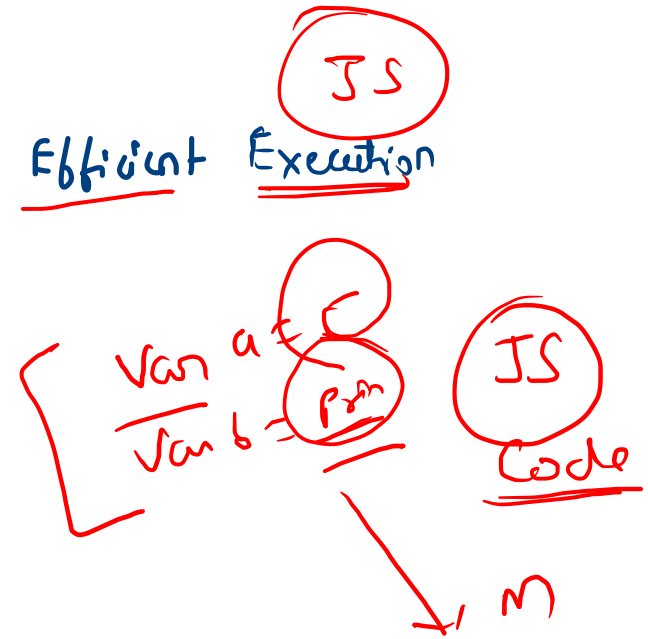
-> But Nodejs is,

- open source
- server side
- Javascript
- Runtime Enviroment → that allow to Run JS code in the server

-> Node Js allow developer to use same language in frontend and Backend

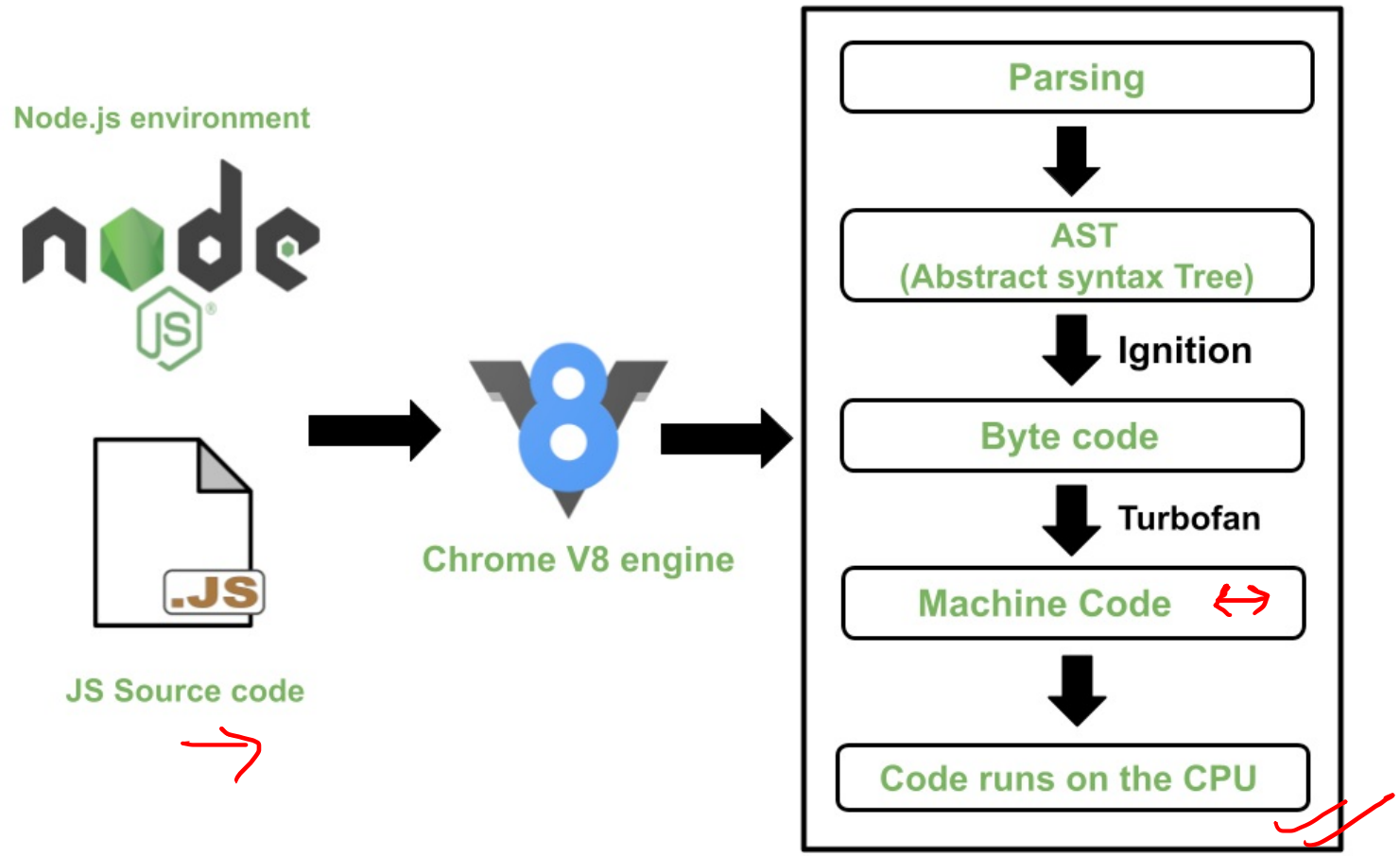
i.e client side
server side

- Node.js is built on top of "Chrome V8 Javascript Engine"
  - ↳ which is known for high performance and Efficient Execution of Js code.

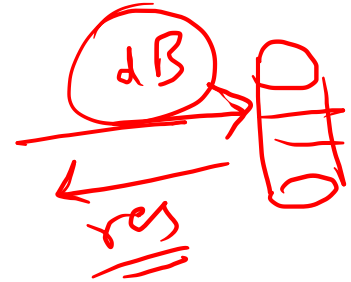


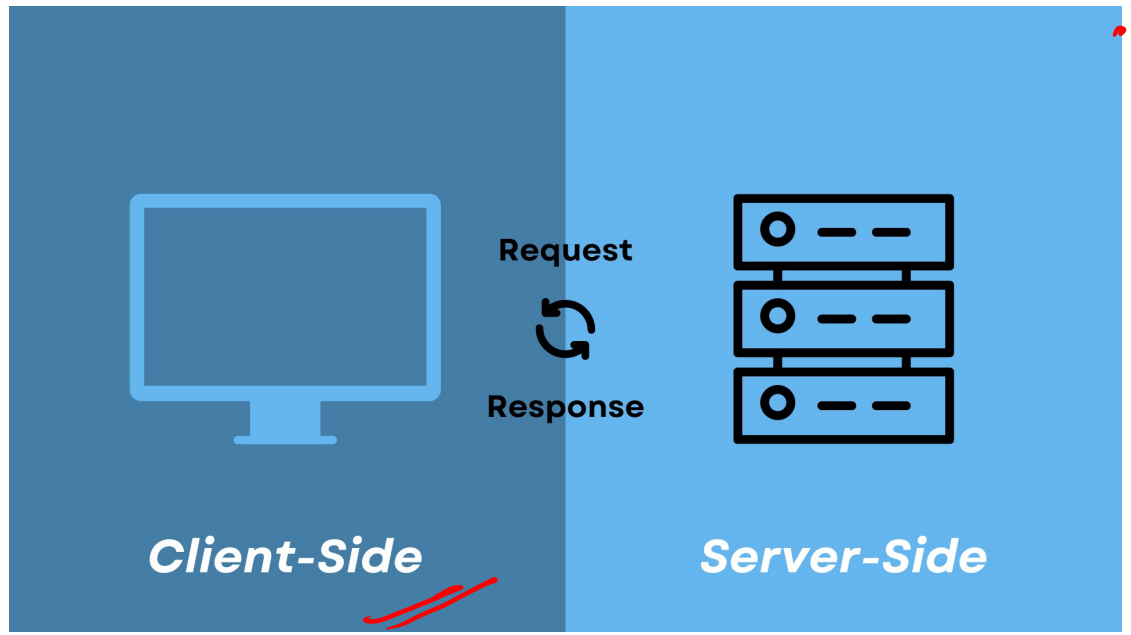
- V8 Engine
  - ↳ open source Javascript Engine developed by Google → to use in the Google chrome web browser.

Human readable JS code → V8 Engine → Machine Readable Language


Node.js environment
JS Source code
Chrome V8 engine
Parsing
AST
(Abstract syntax Tree)
Ignition
Byte code
Turbofan
Machine Code
Code runs on the CPU

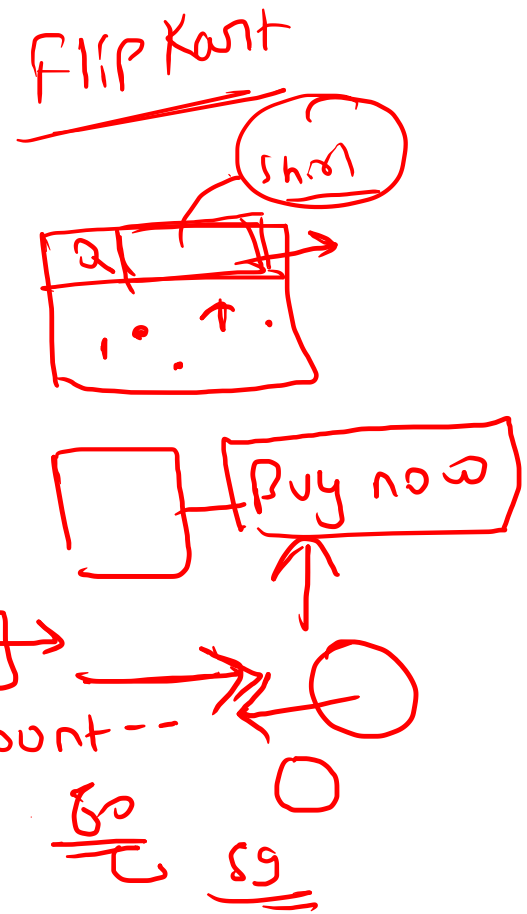
•> understand Client side & server side.

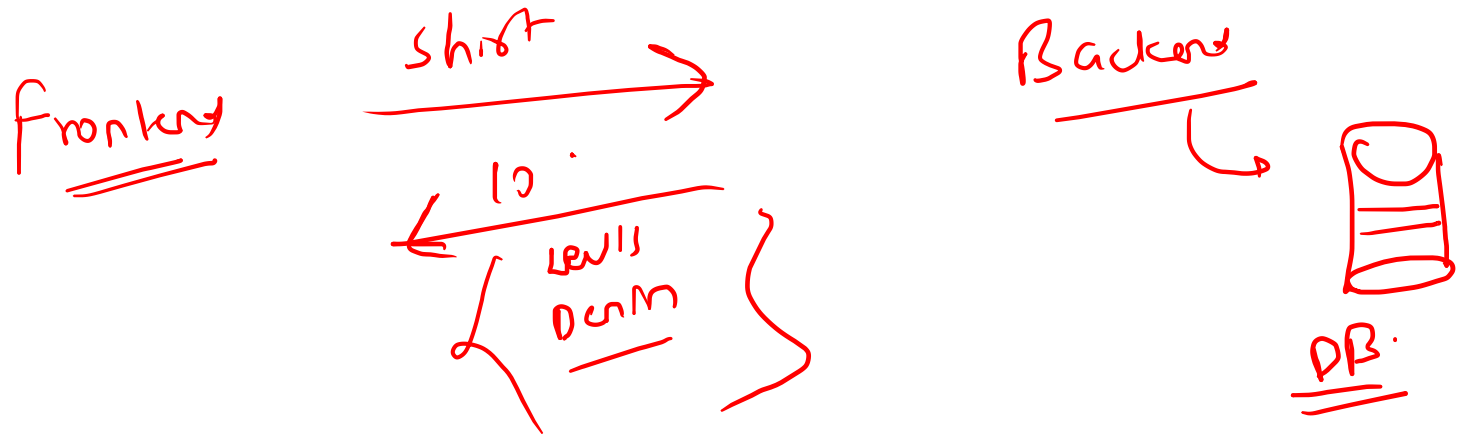




Frontend req/ Backend

res




Frontend
shift
10
levels
down
Backend
DB

Layman Example to understand Nodejs & V8 Engine

↳

client

Already serving

You

JS

Server

JS → V8 → Machine Code

tool

↳ NodeJS

Install
- NodeJS
- NPM
- Vs Code
- Postman

How to Run Javascript code ?

↳ -> I know, many of you don't know Javascript in details :

Let's start JS

```
# Var a = 5
  Var b = 6
  Var ans = a+b
  Console.log (ans)
```

-> online compiler
-> web Browser
-> Terminal
-> Vs code

Basics of JS will be using in Node JS

↳ Variable
Objects
JSON
Console
Array
Random
if
else
Loop
functions
Prompt

DAY 2

- *Learn **npm init***

- *Understand **Package.json***

- **Package.json** – ensure a list of packages with their version
- // Analogy: list of clothes you want to purchase with size and all, it also consists of other metadata like name and all.

- **Package-lock.json** — ensure detailed of every package installed with version, sub dependencies, store detailed, discount everything…
- it's like a detailed billed

- Both files work together to ensure smooth development for you and your team.

- *Understand Callback Functions*

- Let’s understand First the Functions, Like Different ways to write functions

- Then callback Functions
- It’s like a function calling different Functions

- *Core Modules of NodeJS*

- There are many built-in modules in Nodejs that we can use
- https://nodejs.org/api/

- **Learn about the ‘fs’ module**

→ It creates a file and writes the message inside

- **Learn about the ‘os’ module**
- → Learn os.userInfo()
- → Log username

- Write a username into a Txt file with the help of the ‘fs’ system

- **Console.log (‘module’)**
- → It gives the result of all modules we can do with that file in the nodeJs

- fs.appendFile(filename, message, callback);

- *Import Files from*

- Now create a notes.js
- Import file
- console.log(‘notes is starting …’)

- Module.exports.age = 5;
- module.exports.addNumber = function(a,b){}

- *Npm Package - lodash*

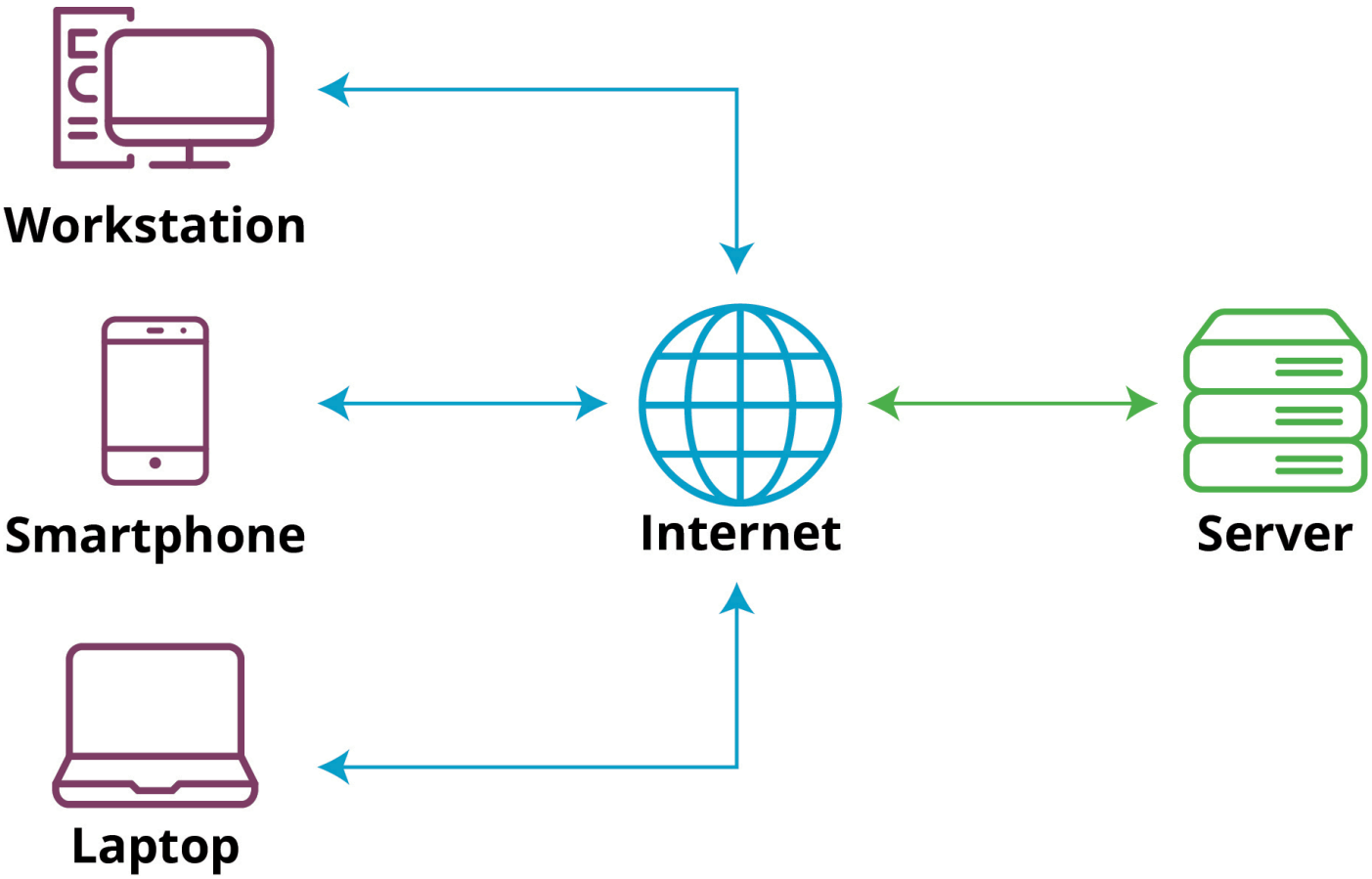
- npm i lodash
- Widely used library
- Lots of functions which helps deal with data.

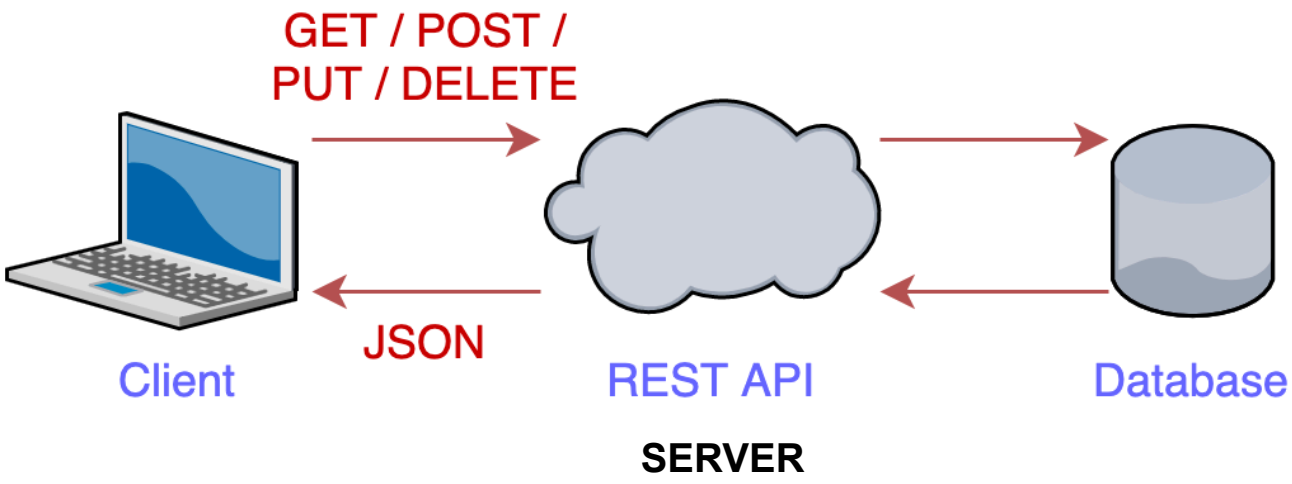
DAY 3

- *Server in Nodejs*

- **Server** – A server is a Person who communicates with clients
- Analogy → server = waiter
- Analogy → chef = database

- A server is a **computer program** that's responsible for preparing and delivering data to other computers
- web pages, images, videos, or any additional information





JSON: JavaScript Object Notation

- Imagine you're sending a message to your friend, and you want to include information like your **name, age, and a list of your favorite hobbies**.
- You can't just send the message as is,
- you need to organize the information in a way that both you and your friend understand.
- JSON is a bit like this organized format for exchanging data between computers.

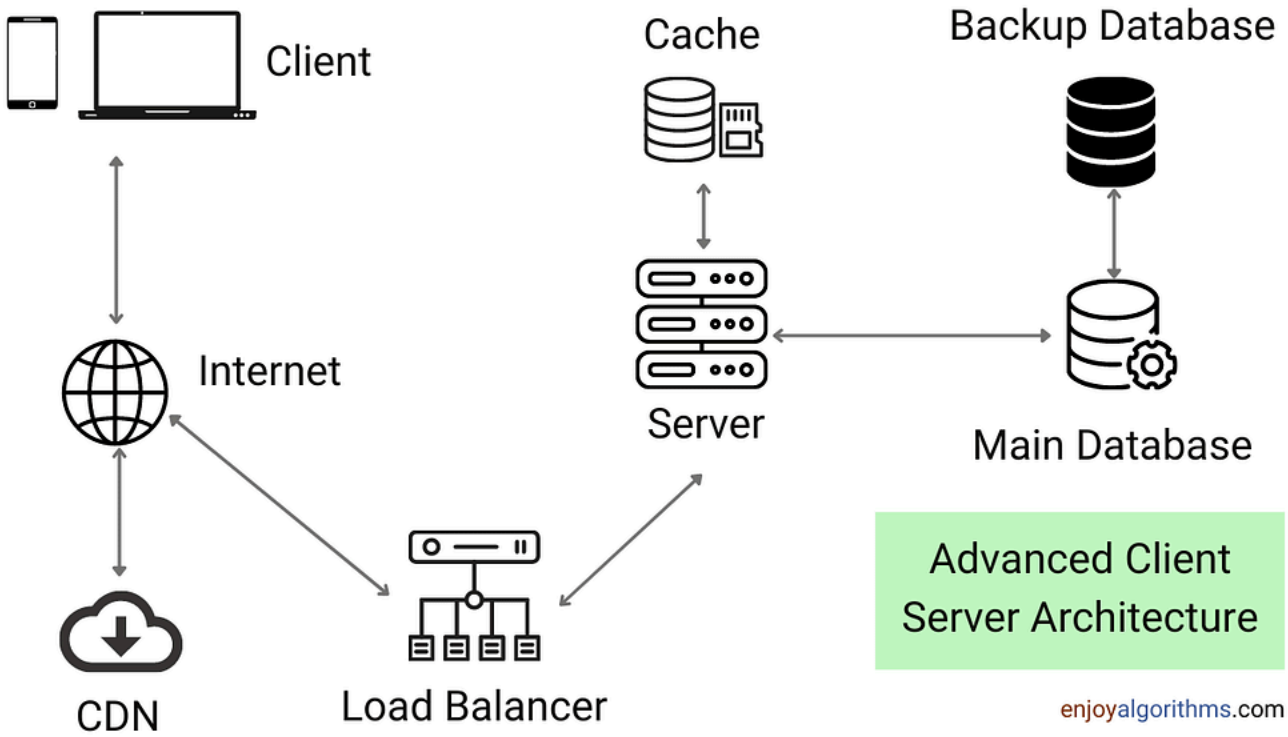
- JSON is a lightweight
- Structured and organized Data because
- in most contexts, JSON is represented as a string

```
{
  "name": "Alice",
  "age": 25,
  "hobbies": ["reading", "painting", "hiking"]
}
```

- **Inter Conversion JSON to an Object in Node.js:**

```
const jsonString = '{"name": "John", "age": 30, "city": "New York"}';
const jsonObject = JSON.parse(jsonString); // Convert JSON string to object
console.log(jsonObject.name); // Output: John

-----------------------------------------------------------

const objectToConvert = { name: "Alice", age: 25 };
const jsonStringified = JSON.stringify(objectToConvert); // Convert object
JSON string
console.log(jsonStringified); // Output: {"name": "Alice", "age":25}
```

- **ADVANCED ARCHITECTURE OF WEB FLOW**



- *What are API and Endpoints?*

- Imagine a menu card in a restaurant
- Lots of options are there, each option will give you a different order
- Now, collection of that list = Menu card = API's
- And an option in that list = Endpoint
- And the waiter only understood whatever things are written on the menu card

- *Create a server*

- Creating a server in NodeJs via **express** package

- Express.js is a popular framework for building **web applications** and **APIs** using Node.js.
- When you create an Express.js application, you're setting up the **foundation for handling incoming requests** and defining how your application **responds** to them.

- Now we are going to create a server == waiter
- Now the waiter has his own home?

In simple terms, "**localhost**" refers to your **own computer**. After creating a server in NodeJS, you can access your environment in 'localhost'

- Port Number?
- Let's suppose in a building – 100 rooms are there, for someone to reach he must know the room number right?

- *Methods to share data*

- Now, in the world of web development, we need to deal with data
- How data is sent and received between a client (like a web browser) and a server (built with Node.js)

- So there are lots of methods out there to **send or receive data** according to their needs.

- GET
- POST
- PATCH
- DELETE

- **GET**

- Imagine you want to read a book on a library shelf.
- You don't change anything
- you just want to get the information.

Similarly, the GET method is used to request data from the server.

For example, when you enter a website URL in your browser,
your browser sends a GET request to the server to fetch the web page.

- Code that we have written on the videos

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


app.get('/', function (req, res) {
    res.send('Welcome to my hotel... How i can help you ?, we
have list of menus')
})

app.get('/chicken', (req, res)=>{
    res.send('sure sir, i would love to serve chicken')
})

app.get('/idli', (req, res)=>{
    var customized_idli = {
        name: 'rava idli',
        size: '10 cm diameter,
        is_sambhar: true,
        is_chutney: false
    }
    res.send(customized_idli)
})

app.listen(3000, ()=>{
    console.log('listening on port 3000');
})
```

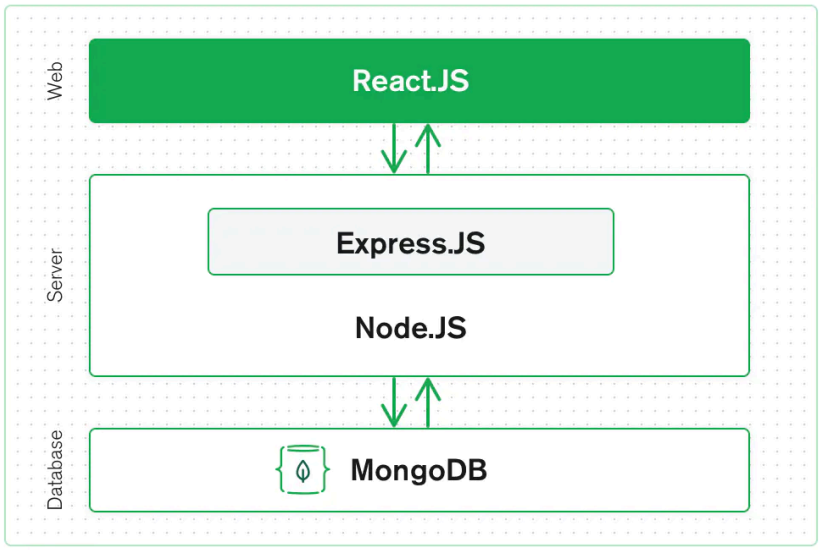
DAY 4

- *Database*

- **Web development = client + server + database**
- Ultimately, Let's suppose we are going to open Restuarant and there is lots of data around it,
    Number of chefs
    Each Person's Detail (like chef, owner, manager, waiter)
        - Name
        - Age
        - Work
        - Mobile number
        - Email
        - Address
        - salary
    Menu Details (like, drinks, Snacks, main course)
        - Name of dish
        - Price
        - Taste (like, sweet, sour, spicy)
        - Is_drink (boolean true, false)
        - Ingredients (array of data – ["wheat", "rice", "sugar"]
        - Number of sales (like 76)

- This is all Data we must have to store to run a fully functional restaurant
- So we have to deal with data

- Now There are lots of Database out there in the market, we can use according to our need
    - SQL
    - PostgreSQL
    - MongoDB
    - MariaDB
    - Oracle

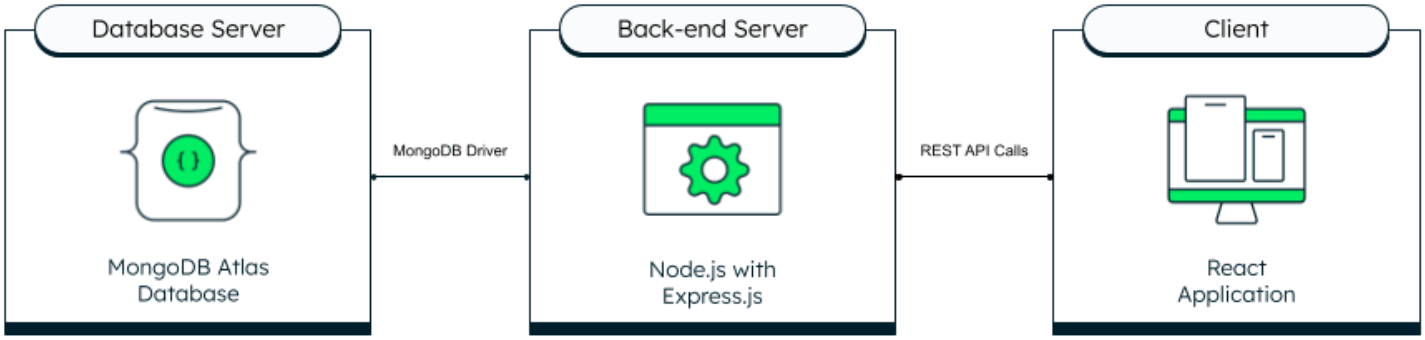


- Databases typically have their own server systems to manage and provide access to the data they store.
- These database server systems are separate from Node.js servers but work together to create dynamic and data-driven web applications

- **Node.js Server and Database Server:**

- A database server is a specialized **computer program** or system that manages databases. It stores, retrieves, and manages data efficiently
- The database server stores your application's data. When your Node.js server needs data, it sends requests to the database server, which then retrieves and sends the requested data back to the Node.js server.

- Node.js server is responsible for handling HTTP requests from clients (like web browsers) and returning responses.
- It processes these requests, communicates with the database server, and sends data to clients.



- *Setup MongoDB*

Q) So as We are creating a backend server, as same do we need to create a Database server as well?

Ans) NO

—> **Setup MongoDB locally is quite tough for most of you. But relax we are here to help you**

- Go to MongoDB's official website
- Or Google MongoDB download

For MacOS
https://stackoverflow.com/questions/65357744/how-to-install-mongodb-on-apple-m1-chip

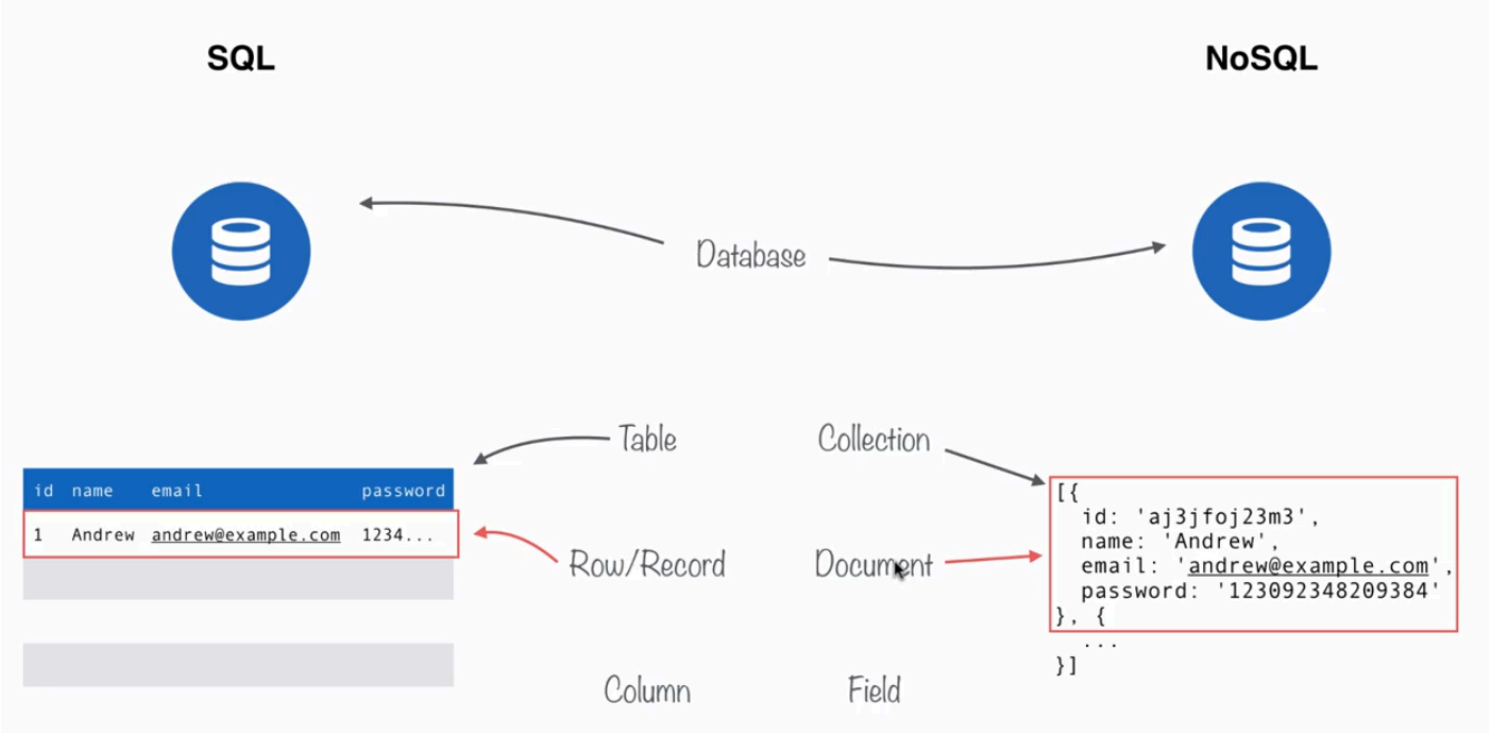
For Windows
https://www.geeksforgeeks.org/how-to-install-mongodb-on-windows/

—- Trust me Give some time to it until you will download the MongoDB in your system. There is no point to move ahead without installing MongoDB

- *MongoDB Syntax and Queries*

Now As we know there is 2 thing, MongoDB

- → Start **MongoDB Server**
- → now we can use **MongoDB shell** from where we can interact with databases



SQL	MongoDB
database	database
table	collection
column	field
row (record)	document

1. Create a Database:

In SQL:

```
CREATE DATABASE mydb;
```

In MongoDB:

```
use mydb;
```

2. Create a Table (Collection in MongoDB):

In SQL:

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  username VARCHAR(50),
  age INT
);
```

In MongoDB:

```
db.createCollection("users");
```

3. Insert Data:

```
INSERT INTO users (id, username, age)
VALUES (1, 'Alice', 25);
```

In MongoDB:

```
db.users.insertOne({ id: 1, username: 'Alice', age: 25 });
```

4. Query Data:

In SQL:

```
SELECT * FROM users WHERE age > 21;
```

In MongoDB:

```
db.users.find({ age: { $gt: 21 } });
```

5. Update Data:

In SQL:

```
UPDATE users SET age = 22 WHERE username = 'Alice';
```

In MongoDB:

```
db.users.updateOne({ username: 'Alice' }, { $set: { age: 22 }
});
```

6. Delete Data:

In SQL:

```
DELETE FROM users WHERE id = 1;
```

In MongoDB:

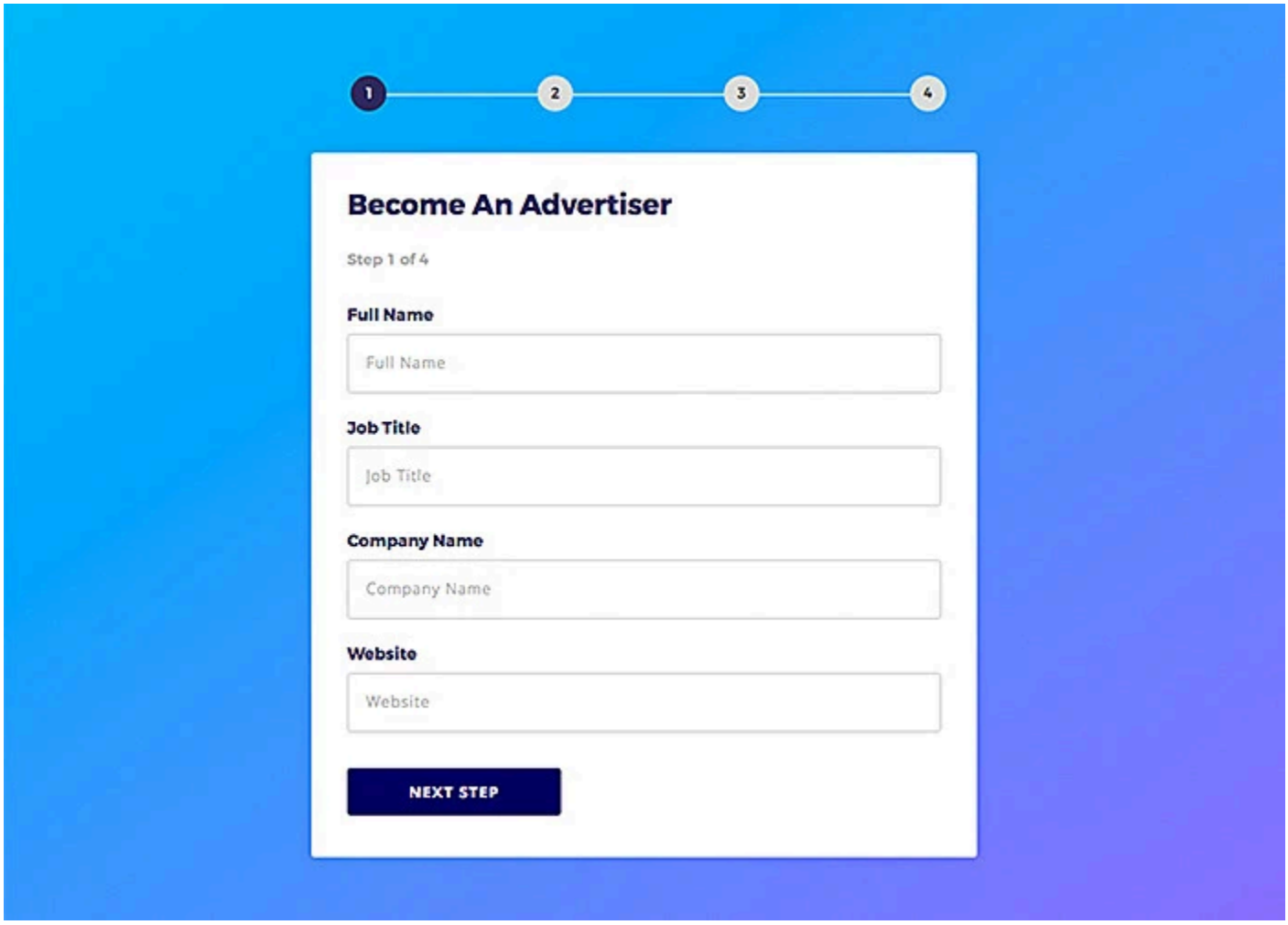
```
db.users.deleteOne({ id: 1 });
```

- *MongoDB Compass GUI*

- There are lots of Tools in the market that help to visualize data like mongoDB compass, MongoDB Robo 3T
- mongodb://127.0.0.1:27017

- *Data Desing and Postman*

- Now in order to use the database we have to integrate MongoDB with nodejs
- Now, there should be a form built on ReactJS or HTML or CSS to add chef or person details
- Now currently we don't have a such frontend thing, so we are using **Postman** for this



- Every Frontend Application, collect Data format it according to backend Requirements and then will send it to backend APIs

- Let’s suppose creating Dummy Data for Person
- Each Person's Detail (like chef, owner, manager, waiter)

```
{
      "name": "Alice",
      "age": 28,
      "work": "Chef",
      "mobile": "123-456-7890",
      "email": "alice@example.com",
      "address": "123 Main St, City",
      "salary": 60000
}
```

- Each Menu’s Detail

```
{
      "name": "Mango Smoothie",
      "price": 4.99,
      "taste": "Sweet",
      "is_drink": true,
      "ingredients": ["mango", "yogurt", "honey"],
      "num_sales": 45
 }
```

```
{
      "name": "Spicy Chicken Wings",
      "price": 9.99,
      "taste": "Spicy",
      "is_drink": false,
      "ingredients": ["chicken wings", "spices", "sauce"],
      "num_sales": 62
}
```

- *Connect MongoDB with NodeJS*

- Now, To connect MongoDB with NodeJS we need a MongoDB driver (a set of programs)

- A MongoDB driver is essential when connecting Node.js with MongoDB because **it acts as a bridge** between your Node.js application and the MongoDB database.

- MongoDB speaks its own language (protocol) to interact with the database server.
- Node.js communicates in JavaScript.
- The driver translates the **JavaScript code from Node.js** into a **format that MongoDB can understand** and vice versa.
- The driver provides a set of functions and methods that make it easier to perform common database operations from your Node.js code.
- The driver helps you handle errors that might occur during database interactions. It provides error codes, descriptions, and other details to help you troubleshoot issues.

- The most popular driver is the official MongoDB Node.js driver, also known as the `mongodb` package.

```
npm install mongodb
```

- *Mongoose*

- Now but we are going to use Mongoose, rather than `mongodb`

- Mongoose is an ***Object Data Modeling (ODM)*** library for MongoDB and Node.js
- There are lots of reasons we prefer Mongoose rather than a native official driver
- Things are a lot easier here

(Relate Real life Examples with mobiles with earphones)

- Mongoose is like a translator between your Node.js code and MongoDB. It makes working with the database smoother and easier.

- With Mongoose, you can define how your data should look, like making a blueprint for your documents. It's like saying, "In our database, each person's information will have a name, age, and email." This makes sure your data stays organized.
- Mongoose helps you make sure the data you put into the database is correct. It's like having someone check if you've written your email address correctly before sending a message.
- Very easy to query from the database

—> But if you are using `mongodb` Native Driver

- You need to write a lot of detailed instructions to make sure everything works correctly.
- Without Mongoose, your code might get messy and harder to understand.
- Since you need to handle many details yourself, it can take longer to finish your project.

In a nutshell, using Mongoose makes working with MongoDB in Node.js much simpler and smoother. It gives you tools that handle complexities for you, so you can focus on building your application without getting bogged down in technical details.

DAY 5

- *Database Connection*

- **Connect MongoDB with NodeJS**

- ***CREATE A FILE db.js IN THE ROOT FOLDER***

- The `db.js` file you've created is essentially responsible for establishing a connection between your Node.js application and your MongoDB database **using the Mongoose library**.

- In the Last Lecture, we saw that the mongoose is responsible for connection
- So let's import Mongoose Library

- *Connection Step by Step*

1. **Import Mongoose and Define the MongoDB URL:** In the `db.js` file, you first import the Mongoose library and define the URL to your MongoDB database. This URL typically follows the format `mongodb://<hostname>:<port>/<databaseName>`. In your code, you've set the URL to `'mongodb://localhost:27017/mydatabase'`, where `mydatabase` is the name of your MongoDB database.
2. **Set Up the MongoDB Connection:** Next, you call `mongoose.connect()` to establish a connection to the MongoDB database using the URL and some configuration options (`useNewUrlParser`, `useUnifiedTopology`, etc.). This step initializes the connection process but does not actually connect at this point.
3. **Access the Default Connection Object:** Mongoose maintains a default connection object representing the MongoDB connection. You retrieve this object using `mongoose.connection`, and you've stored it in the variable

`db`. This object is what you'll use to handle events and interact with the database.

4. **Define Event Listeners:** You define event listeners for the database connection using methods like `.on('connected', ...)`, `.on('error', ...)`, and `.on('disconnected', ...)`. These event listeners allow you to react to different states of the database connection.
5. **Start Listening for Events:** The code is set up to listen for events. When you call `mongoose.connect()`, Mongoose starts the connection process. If the connection is successful, the `'connected'` event is triggered, and you log a message indicating that you're connected to MongoDB. If there's an error during the connection process, the `'error'` event is triggered, and you log an error message. Similarly, the `'disconnected'` event can be useful for handling situations where the connection is lost.
6. **Export the Database Connection:** Finally, you export the `db` object, which represents the MongoDB connection, so that you can import and use it in other parts of your Node.js application.

To sum it up, the `db.js` file acts as a central module that manages the connection to your MongoDB database using Mongoose. It sets up the connection, handles connection events, and exports the connection object so that your Express.js server (or other parts of your application) can use it to interact with the database. **When your server runs, it typically requires or imports this `db.js` file to establish the database connection before handling HTTP requests.**

- *What are models or schema?*

- Models are like a blueprint of our database
- It's a representation of a specific collection in MongoDB. Like a Person
- Once you have defined a model, you can create, read, update, and delete documents in the corresponding MongoDB collection.
- Mongoose allows you to define a schema for your documents. A schema is like a blueprint that defines the structure and data types of your documents within a collection.

- Each Person's Detail (like chef, owner, manager, waiter)

```
{
      "name": "Alice",
      "age": 28,
      "work": "Chef",
      "mobile": "123-456-7890",
      "email": "alice@example.com",
      "address": "123 Main St, City",
      "salary": 60000
}
```

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

- Parameters:
- Type, required, unique, etc

- *What is body-parser*

- `bodyParser` is a middleware library for Express.js.
- It is used to parse and extract the body of incoming HTTP requests.

- When a client (e.g., a web browser or a mobile app) sends data to a server, it typically includes that data in the body of an HTTP request.
- This data can be in various formats, such as JSON, form data, or URL-encoded data. `bodyParser` helps parse and extract this data from the request so that you can work with it in your Express.js application.
- `bodyParser` processes the request body before it reaches your route handlers, making the parsed data available in the `req.body` for further processing.

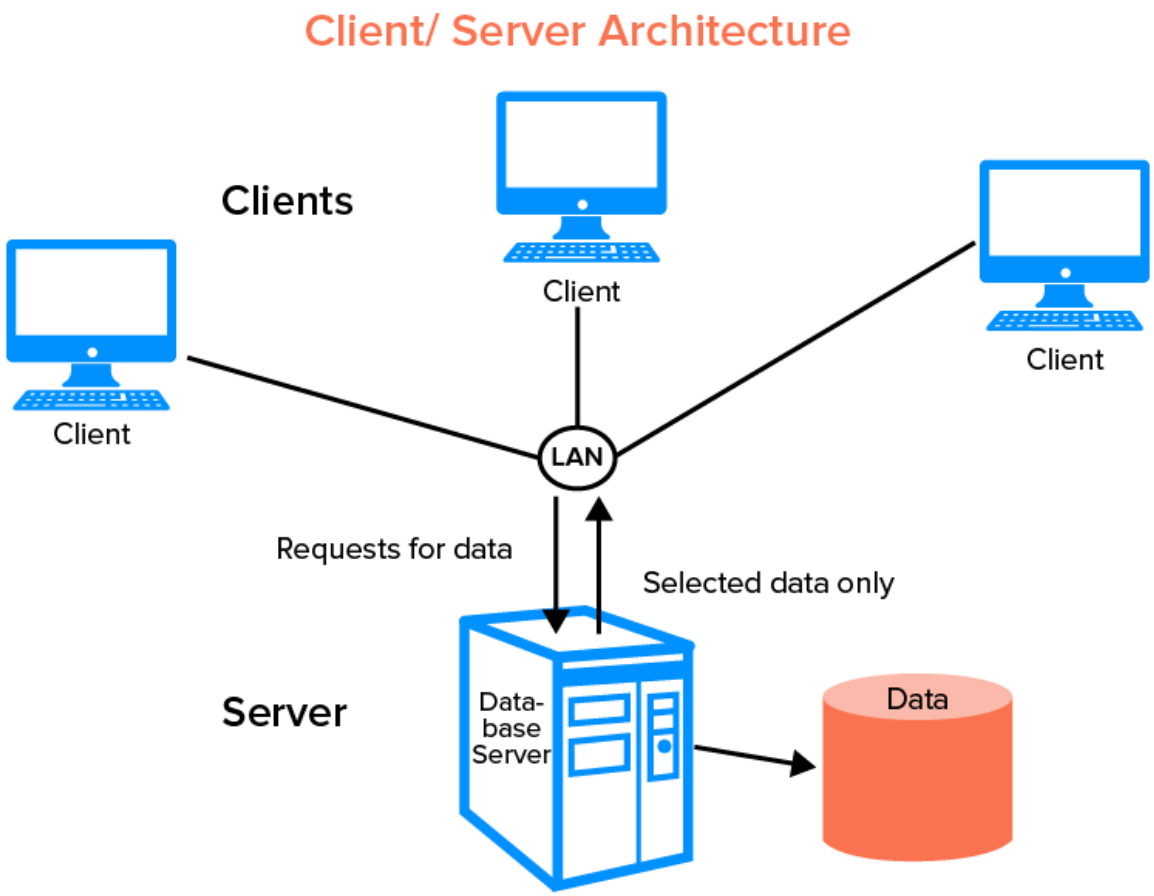
- `bodyParser.json()` automatically parses the JSON data from the request body and converts it into a JavaScript object, which is then stored in the `req.body`

- Express.js uses lots of middleware and to use middleware we use the app.use()

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

- *Send Data from Client to Server*

- we need an Endpoint where the client sends data and data needs to be saved in the database



- we need a method called POST
- Now code the POST method to add the person
- If we send the random values as well Mongoose will not save random values other than predefined schema

```
newPerson.save((error, savedPerson) => {
  if (error) {
    console.error('Error saving person:', error);
    res.status(500).json({ error: 'Internal server error' });
  } else {
    console.log('Data saved');
    res.status(201).json(savedPerson);
  }
});
```

- *Async and Await*

- Nowadays no one uses callback functions like, we used in the POST methods They look quite complex and also do not give us code readability.
- What actually callback does, callback is a function that is executed just after the execution of another main function, it means the callback will wait until its main function is not executed

- **Async and await** are features in JavaScript that make it easier to work with asynchronous code, such as network requests, file system operations, or database queries.
- Using try-and-catch block
- The `try` block contains the code for creating a new `Person` document and saving it to the database using `await newPerson.save()`.
- If an error occurs during any step, it is caught in the `catch` block, and an error response is sent with a 500 Internal Server Error status.

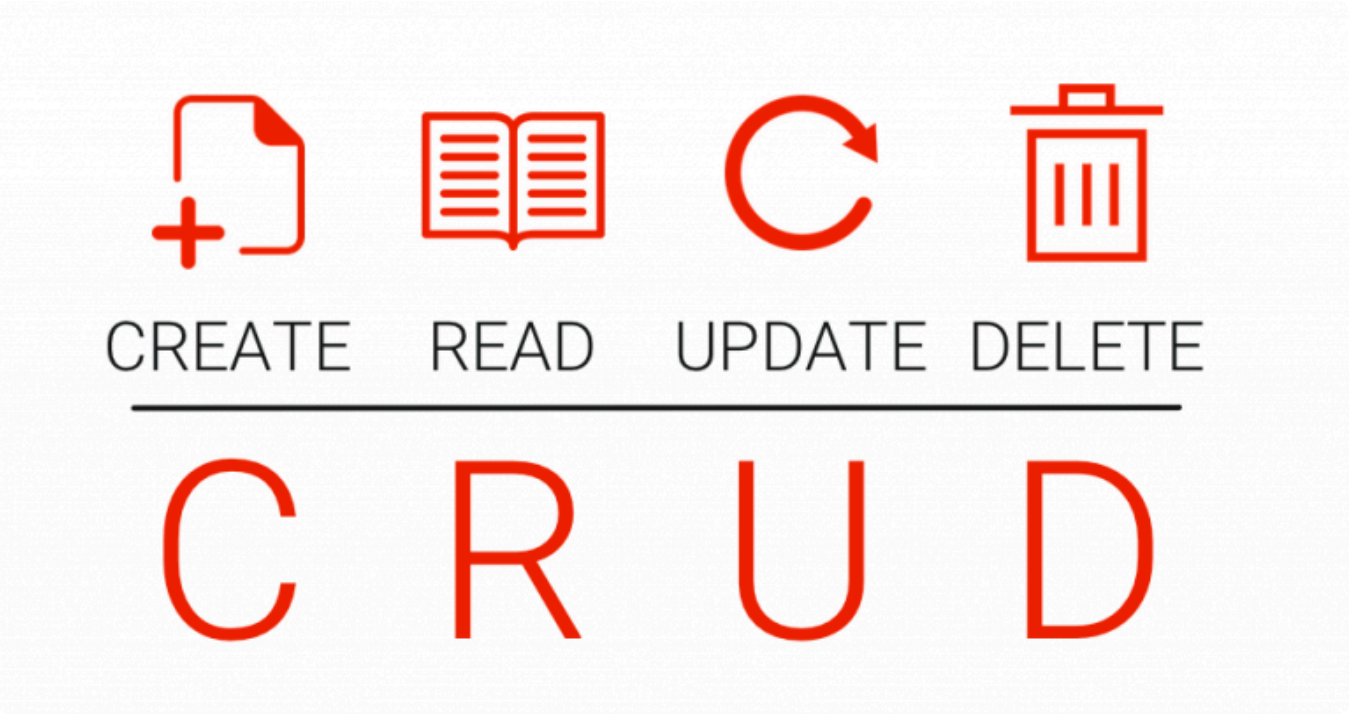
```
app.post('/person', async (req, res) => {
  try {
    const newPersonData = req.body;
    const newPerson = new Person(newPersonData);

    // Save the new person to the database using await
    const savedPerson = await newPerson.save();

    console.log('Saved person to database');
    res.status(201).json(savedPerson);
  } catch (error) {
    console.error('Error saving person:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- **Async Function (`async`):**
  - An `async` function is a function that is designed to work with asynchronous operations. You declare a function as `async` by placing the `async` keyword before the function declaration.
  - The primary purpose of an `async` function is to allow you to use the `await` keyword inside it, which simplifies working with promises and asynchronous code.
  - Inside an `async` function, you can use `await` to pause the execution of the function until a promise is resolved. This makes the code appear more synchronous and easier to read.
- **Await (`await`):**
  - The `await` keyword is used inside an `async` function to wait for the resolution of a promise. It can only be used within an `async` function.
  - When `await` is used, the function pauses at that line until the promise is resolved or rejected. This allows you to write code that appears sequential, even though it's performing asynchronous tasks.

- If the promise is resolved, the result of the promise is returned. If the promise is rejected, it throws an error that can be caught using `try...catch`.

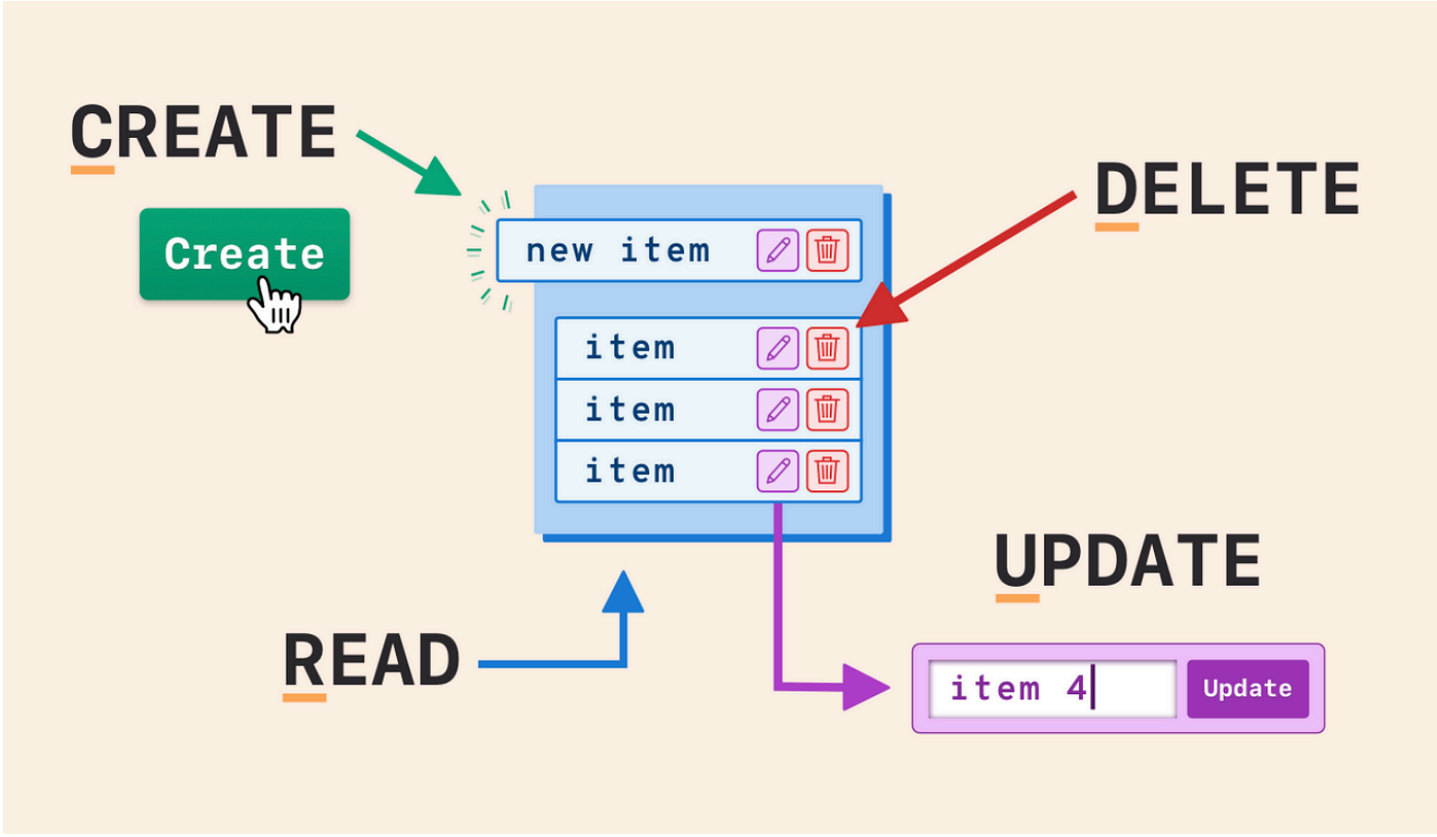
- *CRUD application*

- In any application at the core level, we are always handling the database



- Now we have seen that two methods **POST** and **GET**

		Database Operations		HTTP Methods
C	→	Create	→	POST
R	→	Read	→	GET
U	→	Update	→	PUT/PATCH
D	→	Delete	→	DELETE



- *GET Methods*

- Now Let's suppose the client wants data on all the persons
- So we need an endpoint for that **/person**

```
app.get('/person', async (req, res) => {
  try {
    // Use the Mongoose model to fetch all persons from the
database
    const persons = await Person.find();

    // Send the list of persons as a JSON response
    res.json(persons);
  } catch (error) {
    console.error('Error fetching persons:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- *Create schema for Menu*

- Now create a model for the menu

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

const menuItemSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
```

```
  price: {
    type: Number,
    required: true,
  },
  taste: {
    type: String,
    enum: ['Sweet', 'Spicy', 'Sour'],
  },
  is_drink: {
    type: Boolean,
    default: false,
  },
  ingredients: {
    type: [String],
    default: [],
  },
  num_sales: {
    type: Number,
    default: 0,
  }
});

const MenuItem = mongoose.model('MenuItem', menuItemSchema);

module.exports = MenuItem;
```

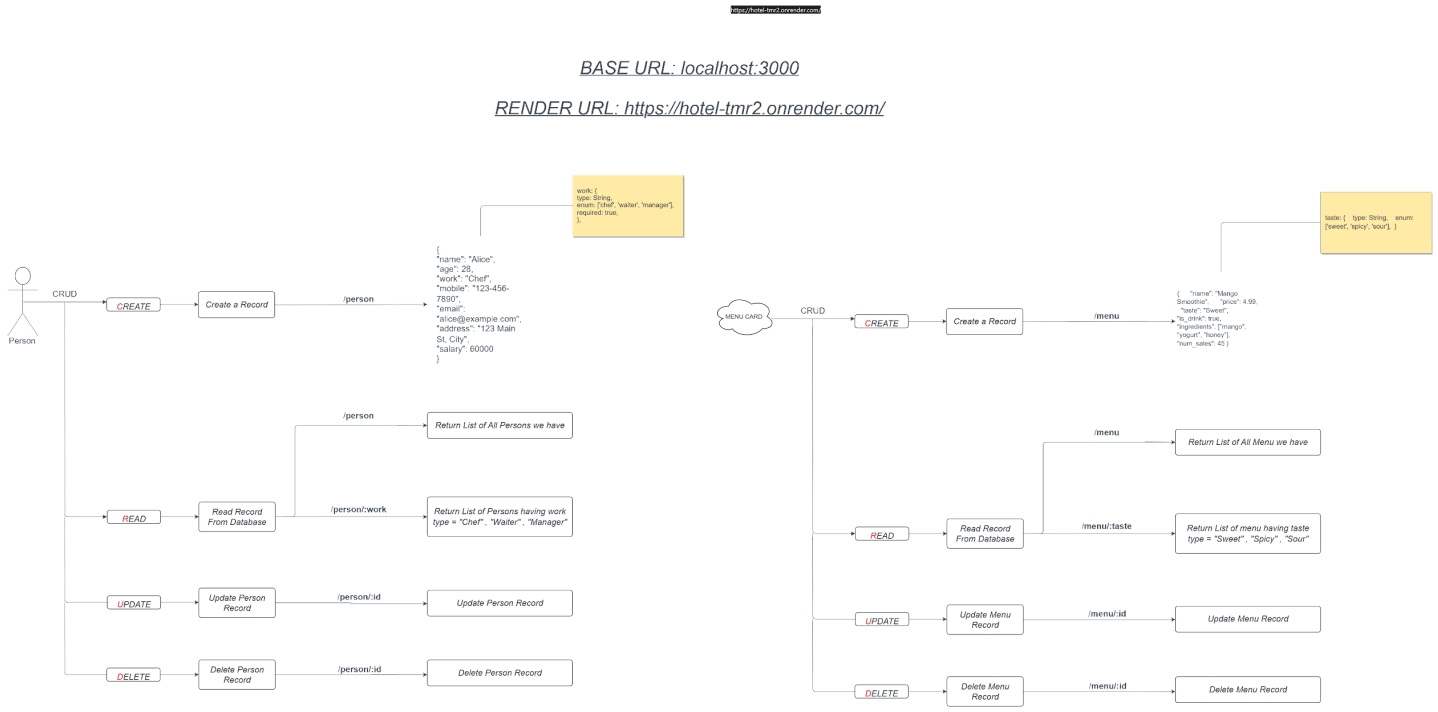
DAY 6

- *Homework Update for Menu API*

- **Task To create POST /menu and GET /menu**

- We are now creating a POST method to save menu details and it's similar to person details and the same for the GET method

- *Flow Diagram of API*



https://drive.google.com/file/d/1TswAyCgfsa04Hp6f4OP-Umg_GVkdW4eQ/view?usp=sharing

- *Parametrised API calls*

- Now if someone told you to give a list of people who are only waiters
- Then we can create an endpoint like this

- **/person/chef**
- **/person/waiter**
- **/person/manager**

- But this is not the correct method to create as many functions Here we can use parametrized endpoints
- It can be dynamically inserted into the URL when making a request to the API.

- **localhost:3000/person/:work**

→ work = [“chef”, “waiter”, “manager”]

```
app.get('/person/:work', async (req, res) => {
  try {
    const workType = req.params.work; // Extract the work type from the URL parameter

    // Assuming you already have a Person model and MongoDB connection set up
    const persons = await Person.find({ work: workType });

    // Send the list of persons with the specified work type as a JSON response
    res.json(persons);
  } catch (error) {
    console.error('Error fetching persons:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- *Express Router*

- We have lots of Endpoints in a single file server.js
- This makes bad experience in code readability as well as code handling
- Express Router is a way to modularize and organize your route handling code in an Express.js application.
- So let's create a separate file to manage endpoints /person and /menu
- Express Router is like a traffic cop for your web server
- Express Router helps you organize and manage these pages or endpoints in your web application. It's like creating separate folders for different types of tasks.

- Create a folder `routes` → `personRoutes.js`

```
const express = require('express');
const router = express.Router();

// Define routes for /person
router.get('/', (req, res) => {
  // Handle GET /person
});

router.post('/', (req, res) => {
  // Handle POST /person
});

module.exports = router;
```

- Now in **server.js,** we will use this `personRoutes`

```
// Import the router files
const personRoutes = require('./routes/personRoutes');

// Use the routers
app.use('/person', personRoutes);
```

- *Update Operation*

- We will update our person Records, and for that, we will create an endpoint from where we are able to update the record
- For Updation, we need two things
    - Which record we want to update?
    - What exactly do we want to update?
- For update, we will use the ***PUT*** method to create an endpoint
- *What is a unique identifier in a document in a collection?*
- It's **_id** which Mongodb itself gives, We will use this to find the particular record that we want to update
- —> And now we will send the data the same as we did in the POST method.

```
app.put('/person/:id', async (req, res) => {
  try {
    const personId = req.params.id; // Extract the person's ID
from the URL parameter
    const updatedPersonData = req.body; // Updated data for the
person

    // Assuming you have a Person model
    const updatedPerson = await
Person.findByIdAndUpdate(personId, updatedPersonData, {
      new: true, // Return the updated document
      runValidators: true, // Run Mongoose validation
```

```
    });

    if (!updatedPerson) {
      return res.status(404).json({ error: 'Person not found'
});
    }

    // Send the updated person data as a JSON response
    res.json(updatedPerson);
  } catch (error) {
    console.error('Error updating person:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- *Delete Operation*

- We will **Delete** our person Records, and for that we will create an endpoint from where we are able to delete the record
- For Deletion, we need one thing
    - Which record we want to update?
- For deletion, we will use the ***DELETE*** method to create an endpoint
- *What is a unique identifier in a document in a collection?*
- It's **_id** which Mongodb itself gives, We will use this to find the particular record that we want to delete

```
app.delete('/person/:id', async (req, res) => {
  try {
    const personId = req.params.id; // Extract the person's ID
from the URL parameter

    // Assuming you have a Person model
```

```
    const deletedPerson = await Person.findByIdAndRemove(personId);

    if (!deletedPerson) {
      return res.status(404).json({ error: 'Person not found' });
    }

    // Send a success message as a JSON response
    res.json({ message: 'Person deleted successfully' });
  } catch (error) {
    console.error('Error deleting person:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

DAY 7

- *Git & GitHub*

- **Git is like a time machine for your code.**

- It is a tool that keeps a record of every version of your code, so you can always go back to a previous state if something goes wrong.

- **Install Git:** If you haven't already, download and install Git on your computer. You can get it from the official Git website: https://git-scm.com/downloads
- If you want to work with git in your project →
- Run `git init` inside the root folder of your project
- This command tells Git to start tracking changes in your project folder.

```
git status
```

- After making changes to your project (e.g., writing code), you'll want to save those changes in Git.

```
git add .
```

- The `.` means "add all changes." You can replace it with specific file names if needed.

- *gitignore*

- The `.gitignore` file is a special configuration file used in Git repositories to specify files and directories that Git should ignore.
- These ignored files and directories won't be tracked by Git or included in version control.
- **Create `.gitignore` File**

```
# Ignore node_modules directory
node_modules/

# Other entries...
```

- This saves a snapshot of your project's current state.

```
git commit -m "Initial commit"
```

- If you want to collaborate with others or back up your code online, you can create a remote repository on platforms like GitHub

- **Link Your Local and Remote Repositories**
- If you created a remote repository, you can link it to your local one

```
git remote add origin https://github.com/yourusername/hotels.git
```

- **Push Changes to Remote**
- To send your local commits to the remote repository, use the git push command

```
git push -u origin master
```

- **Pull Changes**
- If you're collaborating with others, you can fetch their changes and merge them into your code using `git pull`.

- *Host MongoDB database*

- **Now we are running locally MongoDB database.**
- All data operation is performed in a local database, so let's host our database server and make our DB online presence

- MongoDB Atlas provides a Free cluster for users where you can host your database for free.
- MongoDB Atlas offers a cloud-based platform for hosting MongoDB databases
- The free tier allows developers to explore and experiment with the database without incurring any costs.
- https://www.mongodb.com/atlas/database

- Create an account for free (I already have an account)
- Show Step-by-step Process to host MongoDB Atlas

- *Dotenv*

- The `dotenv` module in Node.js is used to manage configuration variables and sensitive information in your applications.
- It's particularly useful for keeping sensitive data like API keys, database connection strings, and other environment-specific configurations separate from your code.

```
npm install dotenv
```

- **Create a `.env` File**
- This is where you'll store your environment-specific configuration variables.
- format `VAR_NAME=value`.

```
PORT=3000
API_KEY=your-api-key
DB_CONNECTION_STRING=your-db-connection-string
```

- In your server file (usually the main entry point of your application), require and configure the `dotenv` module.

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

- **Access Configuration Variables:**

```
const port = process.env.PORT || 3000; // Use 3000 as a default
if PORT is not defined
const apiKey = process.env.API_KEY;
const dbConnectionString = process.env.DB_CONNECTION_STRING;
```

- Remember to keep your `.env` file secure and never commit it to a public version control system like Git, as it may contain sensitive information. Typically, you should include the `.env` file in your project's `.gitignore` file to prevent accidental commits.

- *Test MongoDB Cluster Postman*

- Now we can test the MongoDB Cluster and check whether our data is present or not in the online DB

- *Host NodeJS Server*

- Now we are going to host our server so that our Application or Endpoints is accessible to all the users over the Internet.
- We are using localhost and our endpoints are only accessible within our computer

- We have to make it publicly available, so there are lots of company who helps us to make our application run 24*7
- Like, AWS, Google Cloud, etc. but these charge too much amount for our application
- So we are going to use some free services to host our nodeJS application, which lots of company provides for developer purposes.
- Like, Heroku, Netlify, Render, etc